$\tau\acute{\epsilon}\chi\nu\eta$ and Quest-Oriented Learning

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Sarah Ruth Matzko
August 2007

Accepted by:
Dr. Robert Geist, Committee Chair
Dr. Timothy Davis
Dr. Pradip Srimani
Dr. Andrew Duchowski

المنارة للاستشارات
www.manaraa.com

UMI Number: 3274327

UMI®

UMI Microform 3274327

Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

المنارة للاستشارات

www.manaraa.com

Abstract

A new approach for teaching undergraduate computer science courses is presented. A general teaching approach that is its basis is also described. Summaries and guides to several introductory courses are provided. Results from the use of the curriculum are presented, and other applications of the approach are suggested.

للاستشارات المنارة

www.manaraa.com

Acknowledgments

I would like to thank the members of the curriculum committee for allowing the $\tau\acute{\epsilon}\chi\nu\eta$ approach to be used with Clemson University students. Also, I thank the supportive members of the Clemson University Computer Science faculty for their willingness to adopt the $\tau\acute{\epsilon}\chi\nu\eta$ approach and for their constructive feedback. I greatly appreciate the dedication of the members of my committee to the success of $\tau\acute{\epsilon}\chi\nu\eta$ and their creative ideas and advice. I thank my family for believing I would complete my degree and Nick, whose certainty in my abilities convinced me that I would succeed.

This work was supported in part by the CISE Directorate of the U.S. National Science Foundation under award EIA-0305318.

www.manaraa.com

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Acknowledgments . . . . **iii**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

List of Algorithms . . . . **ix**

1 Introduction **1**

2 Background **2**
- 2.1 Digital Production Arts Degree 2
- 2.2 Raytracing in a Second-Year Course 3

3 Curriculum **9**
- 3.1 Educational Goal 9
- 3.2 Problem 10
- 3.3 Curriculum Structure 11
- 3.4 Educational Approach 12

4 Related Work **19**
- 4.1 Semester-Long Projects in Introductory Courses 19
- 4.2 Graphics in Introductory Courses 20

5 Implementation **22**
- 5.1 Course Phases 22
- 5.2 Supporting Features 34
- 5.3 Introductory Language Selection 41

6 Adaptations to Other Environments **47**
- 6.1 Adaptation to Small Colleges 47
- 6.2 Adaptation to an Upper-Level Course 49

7 Results and Evaluation **53**
- 7.1 The Original, Second-Year Raytracing Course (215) 53
- 7.2 Computer Science I 59
- 7.3 Computer Science II 69
- 7.4 Second-Year Data Structures Course (212) 74

المنارة للاستشارات

www.manaraa.com

7.5 Second-Year, Tools and Techniques for Software Development (215) 77
7.6 Retention 80
7.7 Observations 81

Appendices 83
A CS1 Guide 84
B CS2 Guide 136
C Algorithms and Data Structures Course Guide 225
D Tools and Techniques for Software Development Guide 230

Bibliography 273

المنارة للاستشارات
www.manaraa.com

List of Tables

7.1 2005 101 Comparison 68
7.2 2006-2007 101 Walker-Fraser Surveys 69
7.3 2006 Student Perceptions of 102 72
7.4 2006 102 Skills Comparison 73
7.5 2006 102 Comparison 73
7.6 2007 Walker-Fraser Surveys of raytracing Courses 74
7.7 2007 215 Walker-Fraser Results 80
7.8 Retention 81

المنارة للاستشارات
www.manaraa.com

List of Figures

2.1 2002 CPSC 215 Phase II Student Renderings. 7

6.1 Fully rendered image of the Civil War-era Hunley Submarine 51

7.1 2002 CPSC 215 Example Student Renderings. 54
7.2 2004 CPSC 215 Example Student Renderings. 56
7.3 2003-2004 Pilot 215 Course Relevance 57
7.4 2003-2004 Pilot 215 Graphics Interest 57
7.5 2003-2004 Pilot 215 Perceived Skill Development 58
7.6 2005 CPSC 101 Phase 1 60
7.7 2005 CPSC 101 Phase 1: Images with Enlarged Detail 61
7.8 2006 Covenant CPSC 101 Phase 1 62
7.9 2005 CPSC 101 Phase 2 63
7.10 2005 CPSC 101 Phase 3 64
7.11 2005 CPSC 101 Phase 3: Convolution Filters 65
7.12 2006 Covenant CPSC 101 Phase 3 66
7.13 2006 Covenant CPSC 101 Phase 3 67
7.14 2005 CPSC 101 Phase 4 Color Transfer 67
7.15 CPSC 101 Phase 4 Color Transfer Sources 68
7.16 2006 CPSC 102 Student Images 70
7.17 2007 Covenant College CS2 Images 71
7.18 2006 CPSC 212 Phase 1: Tangent plan estimation 75
7.19 2006 CPSC 212 Phases 2-4 75
7.20 2006-2007 CPSC 212-215 Comparisons 76
7.21 2007 CPSC 215 Student GUI Checkers Games. 77
7.22 2007 CPSC 215 Student GUI Chess Games. 78
7.23 2007 Features in Chess Game by Seagers, Musselman, and Squires 79
24 Demonstration of the left-handed and right-handed coordinate systems 142
25 Gradient sky 147
26 Ray-sphere intersection 151
27 Blue sky and filled circles 157
28 Ray scene projection 158
29 Blue sky, circles, and floor 161
30 Checker algorithm 162
31 Sky, circles, and checkered floor 163
32 Shadows on a bright scene 185
33 Angle to the light 188
34 Lighting with diffuse component 191
35 Light attenuation with distance 194
36 Vector bounce illustration 196
37 Scene with reflectivity 201

www.manaraa.com

38 Anti-aliased image 204
39 Intersection tests 206
40 Scene with boxes 211
41 Checkerboard 242
42 Checkerboard with immobile pieces 247
43 Checkerboard with inner circle 249
44 Checkerboard with 3D pieces 250
45 Checkerboard with 3D, anti-aliased pieces 251

المنارة للاستشارات
www.manaraa.com

List of Algorithms

2.1 Fundamental Object Structure . . . 5
2.2 Pseudo-code for raytracing . . . 6
.1 Output of a single-pixel image file . . . 99
.2 800 by 600 image creation . . . 101
.3 Complete program to read in and print back out a file . . . 103
.4 Complete function for skipping over comments and white space . . . 106
.5 Main function that returns an integer . . . 107
.6 The complete function for reading the ASCII header . . . 109
.7 Header file . . . 110
.8 Invocation of the header-reading function and error testing . . . 110
.9 Modification of the image to exclude blue . . . 112
.10 Image grayscaling . . . 113
.11 Adding scan lines . . . 114
.12 Fade program . . . 115
.13 Monochrome program . . . 117
.14 Declaration of an array of unsigned characters to hold image data . . . 118
.15 Reading of the entire block of image data . . . 119
.16 Rotate 90 degrees . . . 120
.17 Applying the sharpen filter . . . 122
.18 Output of the entire block of image data . . . 122
.19 Dynamic memory allocation . . . 123
.20 3 by 3 multiplication . . . 124
.21 Character and float conversion . . . 125
.22 Character and float conversion, continued. . . . 126
.23 RGB to LMS and LMS to RGB . . . 126
.24 LMS to CIELAB and CIELAB to LMS . . . 127
.25 RGB to CIELAB and CIELAB to RGB . . . 127
.26 Image reading utilities . . . 128
.27 CIELAB conversion main function . . . 129
.28 Reading file name from command line . . . 130
.29 File reading with file handle . . . 131
.30 Log (base 10) and Power of Ten functions . . . 132
.31 Conversion with minimized skewing . . . 132
.32 File information structure . . . 133
.33 Mean and standard deviation . . . 134
.34 Image scaling . . . 134
.35 Color transfer main function . . . 135
.36 Stub raytrace method to fill the image data array with red pixels . . . 138
.37 Function for printing a PPM format image to standard out . . . 139
.38 Main function for invoking the appropriate functions to produce an image file . . . 139

المنارة للاستشارات
www.manaraa.com

.39 Main function with parameters for accepting command-line arguments . . . 140
.40 Declaration of the scene structure . . . 141
.41 Declaration of the sky structure . . . 143
.42 Scene set up function . . . 144
.43 Conversion from pixel coordinates to world coordinates . . . 145
.44 Function to compute a specified pixel's values . . . 145
.45 Dereferencing of function pointer . . . 145
.46 Conversion of the pixel's channel values to the [0,255] range . . . 146
.47 Complete program for creating a gradient sky pattern . . . 146
.48 Addition of sphere's color function . . . 148
.49 Consolidation of shared attributes in the object structure . . . 148
.50 Modification of color functions to match new object structure . . . 149
.51 Square macro . . . 151
.52 Point subtraction function . . . 152
.53 Intersection structure . . . 152
.54 Sphere intersection function . . . 153
.55 Sky intersection function . . . 154
.56 Signature information added to the intersection function pointer declaration . . . 155
.57 Specification of the objects in the scene . . . 156
.58 Addition of nearest object search . . . 157
.59 Specification of the floor structure . . . 158
.60 Addition of floor structure to the geometry union . . . 158
.61 Floor's color function . . . 158
.62 Floor intersection function . . . 159
.63 Addition of the floor to the scene . . . 160
.64 Two colors in the floor structure . . . 160
.65 Specification of the floor colors . . . 161
.66 Functional floor texture . . . 163
.67 If-not-defined preprocessor directive . . . 164
.68 Point class . . . 165
.69 Point class implementation . . . 167
.70 Color class definition . . . 168
.71 Beginning of Color class implementation . . . 168
.72 Definition of intersection structure . . . 169
.73 Definition of purely virtual Object methods to be overridden by child classes . . . 169
.74 Scene class header file . . . 170
.75 Definition of the Sky class . . . 171
.76 Sky class code file . . . 171
.77 Sphere class definition . . . 172
.78 Sphere class . . . 173
.79 Floor class definition . . . 174
.80 Floor class implementation . . . 174
.81 Initialization of Objects in the Scene . . . 175
.82 Scene class's first intersected method and coordinate computation method . . . 175
.83 Raytracer class definition . . . 176
.84 Raytracer implementation . . . 177
.85 Raytracer main function and trace method . . . 178
.86 Makefile . . . 178
.87 Sphere class definition . . . 179
.88 Light class definition . . . 179
.89 Scene class definition . . . 180

www.manaraa.com

.90 Addition of Lights to the Scene 181
.91 Method to iteratively return the next light visible from a given point 182
.92 Addition of a diffuse color method in the Object class 182
.93 Creation of diffuse color methods in child classes 183
.94 Addition of boolean method specifying whether lighting affects this object 183
.95 Color add-to operator overloading 184
.96 Addition of diffuse lighting to color computation 184
.97 Point distance, division and unit vector 186
.98 Definition of Sky intersection 187
.99 Purely virtual normal computation method in Object class 188
.100 Normal computation 189
.101 Color scaling method 189
.102 Raytracer's pixel trace method 190
.103 Sphere class normal computation 192
.104 Declaration of the weight that distance has in this Raytracer 192
.105 Addition of a distance-traveled-so-far parameter 192
.106 Raytracer::trace_pixel update 193
.107 Addition of initial distance traveled argument to the pixel trace invocation. 194
.108 Point arithmetic methods 195
.109 Vector bounce function 196
.110 Addition of reflectivity attributes to Scene Objects 197
.111 Definition of objects to have reflective components 198
.112 Separation of diffuse color computation 199
.113 Computation of specular reflectivity 199
.114 Color scaling methods 200
.115 Update of trace_pixel 201
.116 Pseudo-random jitter method 202
.117 Raytracer's anti-aliasing trace 203
.118 Invocation of anti-aliasing trace from Raytracer loop 204
.119 Box class definition 205
.120 Box constructor and ambient color methods 206
.121 Initialization of Box intersection computation variables 207
.122 Box class normal computation 208
.123 Addition of boxes to the scene 209
.124 Linked list node class 210
.125 Linked list node add after method 210
.126 Linked list iterator class 212
.127 Linked list iterator methods 212
.128 Iterator method for getting the next Object 212
.129 Iterator dereferencing and incrementing 212
.130 Beginning of Linked list code 213
.131 Remainder of linked list class 213
.132 Creation of linked list functions 214
.133 Scene objects in a linked list 215
.134 First intersection method with a linked list 216
.135 Next light method with a linked list 216
.136 Scene specification file 218
.137 friend function for reading in a Point 219
.138 Friend function for reading in a color 219
.139 Sphere constructor for reading a sphere 220
.140 Box input constructor 221

المنارة للاستشارات
www.manaraa.com

.141 Floor input constructor . . . 221
.142 Sky input constructor . . . 222
.143 Light constructor for reading input . . . 222
.144 Added object reading method . . . 223
.145 Scene object reading . . . 223
.146 Raytracer scene name parameter . . . 224
.147 Main function that accepts an input file . . . 224
.148 Simple program . . . 235
.149 Beginning of the GameSquare class . . . 239
.150 Draw method for creating the square in the appropriate location . . . 239
.151 CheckerBoard class . . . 241
.152 Piece class instance variables and accessor/mutator methods . . . 243
.153 Piece's draw method . . . 244
.154 Square class . . . 245
.155 CheckerBoard Piece placement method . . . 245
.156 Updated CheckerBoard constructor . . . 246
.157 Beginning of Checkers class . . . 246
.158 Piece placement method . . . 247
.159 Checkers class instantiation . . . 247
.160 Beginning of the Piece class draw method . . . 248
.161 End of draw method . . . 248
.162 Beginning of updated draw method . . . 249
.163 Gradient Paint . . . 249
.164 Creation of inset circle . . . 250
.165 Addition of anti-aliasing . . . 251
.166 Number across accessor method . . . 252
.167 Piece draw method with center location specified . . . 253
.168 Simplified Piece draw method . . . 253
.169 Piece play method . . . 254
.170 Addition of reference to moving piece . . . 254
.171 Updated CheckerBoard's paint method . . . 254
.172 Extension of the MouseAdapter . . . 255
.173 Mouse pressed event . . . 255
.174 Mouse dragged event . . . 256
.175 Mouse released method . . . 256
.176 Addition of listener . . . 257
.177 Player class . . . 258
.178 Addition of a Player to the Piece class . . . 259
.179 Updated color accessor method . . . 259
.180 Updated Piece draw method . . . 260
.181 Move validation . . . 260
.182 Piece's make play move with validation . . . 260
.183 Piece placement with associated players . . . 261
.184 Sample exception handling . . . 267

المنارة للاستشارات
www.manaraa.com

Chapter 1

Introduction

The $\tau\acute{\epsilon}\chi\nu\eta$ (pronounced "TEKnee") project is a zero-based re-design of the undergraduate curriculum in computer science to incorporate more artistic components. $\tau\acute{\epsilon}\chi\nu\eta$ is the Greek word for art. It shares its root with $\tau\epsilon\chi\nu o\lambda o\gamma\acute{\iota}\alpha$, the Greek word for technology. The project name comes out of an effort to reunite art with computer science in order to broaden computer science education [15].

Inspiration from the success of the Digital Production Arts program and the following pilot course led to the National Science Foundation grant titled "$\tau\acute{\epsilon}\chi\nu\eta$." Research under the grant has led to 9 papers thus far, covering the success of the pilot course [15], the introductory course [51], [50], the laboratory approach [49], the second (CS2) course [17], the first year trial phase [16], the second-year data structures course [22], the application of $\tau\acute{\epsilon}\chi\nu\eta$ to an upper-level course [14], and the adaptation of $\tau\acute{\epsilon}\chi\nu\eta$ to a different institution [33].

The background of $\tau\acute{\epsilon}\chi\nu\eta$ is covered in Chapter 2. The curriculum structure and generalized educational approach are discussed in Chapter 3. Chapter 4 covers similar approaches in the literature, and Chapter 5 fleshes out the details of the implementation of $\tau\acute{\epsilon}\chi\nu\eta$. Applications of $\tau\acute{\epsilon}\chi\nu\eta$ to other settings are explained in Chapter 6, and results and evaluation may be found in Chapter 7. Detailed guides for those who wish to try $\tau\acute{\epsilon}\chi\nu\eta$ courses in their own settings are available in the appendices.

المنارة للاستشارات
www.manaraa.com

Chapter 2

Background

The path to creating the $\tau\acute{\epsilon}\chi\nu\eta$ approach began by observations made from the Digital Production Arts degree program and from the first trial course taught with a large-scale, graphical problem. From the beginning, $\tau\acute{\epsilon}\chi\nu\eta$ demonstrated signs of success which led to its large-scale adoption.

2.1 Digital Production Arts Degree

In 1999, Clemson University established a graduate degree program that bridged the arts and the sciences. The Master of Fine Arts in Digital Production Arts is a two-year program that is aimed at producing digital artists who carry solid foundations in computer science and thus can expect to find employment in the rapidly expanding special effects industry for video, film, and interactive gaming. Students in the program are required to complete graduate-level work in both the arts and computer science. Graduates have found employment in many of the top studios, e.g., Industrial Light & Magic (Lucasfilms), Rhythm & Hues, BlueSky Studios, DreamWorks, Tippett Studios, and Pixar. This program effected a significant change in the Clemson undergraduate degree program enrollment. The faculty witnessed a shift of undergraduate majors from the B.S. degree in Computer Science to the B.A. degree in Computer Science with an elected minor in Art. Whether these undergraduate students ultimately pursue the DPA program or not, it is our position that this shift to a more balanced educational experience is of substantial benefit to the students and to society.

An interesting and initially unexpected result of the DPA program was the demographic of the students enrolled. The problem of under-representation of women and minorities in computing programs is well known and, nationwide, shows little sign of amelioration. The DPA program had an initial enrollment

المنارة للاستشارات
www.manaraa.com

of 32% women and 16% African American, both well above the averages for more conventional graduate programs in computing, including those at Clemson. A natural conjecture was that a DPA-based re-design of the computer science program would effect similar enrollment shifts, and the new curriculum might merit widespread adoption on this basis as well as on the bases of enhanced problem-solving skills of the students and enhanced enthusiasm of all participants.

Consequently, the experiences and observations from the Digital Production Arts program led to the investigational application of art and graphics research into required computing courses. Instruction in these courses was and continues to be strictly oriented toward large-scale problem-solving using problems that are visual in nature.

2.2 Raytracing in a Second-Year Course

The trial course for this new, graphical approach was CPSC 215, Tools and Techniques for Software Development, a second-year course. The intent of the course at the time was instruction in programming methodology using C and C++. To meet this end, the trial course [15] was taught through the large-scale problem of constructing a raytracing system for rendering synthetic images. The trial course's success led to the National Science Foundation grant titled $\tau\acute{\epsilon}\chi\nu\eta$.

2.2.1 Trial Course Design

The project chosen for the trial course was raytracing. Raytracing is a technique for synthesizing images by following hypothetical photon paths [18], [27], [56]. A raytracing system models a virtual viewer looking upon a collection of geometrically-specified objects in Euclidean three-space. A virtual rectangular viewing screen is interposed between the viewer and the objects. The screen is oriented so that a vector that is normal to the screen and based at the center of the screen passes through the eye point of the viewer. The screen is considered a two-dimensional lattice of equally spaced points representing pixels or sub-pixels of the final projected image. The raytracing algorithm creates the image by firing a virtual photon from the viewpoint through each lattice point. If the photon hits an object, it may bounce and hit additional objects. The color assigned to the lattice point is a weighted sum of the colors of all objects hit by the photon. Many commercial rendering systems for special effects are based upon raytracing [42]. In addition to being an interesting, real-world problem, the development of a raytracing system provides an ideal mechanism for exposing the student to the object-oriented (OO) paradigm. The system implementation can be initiated in an

المنارة للاستشارات
www.manaraa.com

imperative style, but it quickly becomes apparent that the most reasonable way to represent the interactions of photons or rays with different types of geometric objects is to associate functions for calculating ray-object intersection points and surface normal vectors with each type of object. The overall design draws, in a very natural way, into the object-oriented paradigm. The benefits of inheritance and polymorphism are clear from the onset of their introduction. Because the systems naturally grow large and complex very quickly, techniques for partitioning, testing, and large-scale development are well-received.

2.2.1.1 Phase I - Fundamentals

Students who entered CPSC 215 at the time of this course typically had completed the CS I and II courses using Java and had had little or no exposure to the C language or to basic concepts of computer graphics. Three weeks of the fifteen-week course, including both in-class lecture and assigned projects, were devoted to fundamentals of the C language, the standard library, and their use in the representation, storage, and retrieval of image data. These topics were introduced in the context of several assigned 2D image transformation problems, including converting color images to grayscale; digital half toning (converting grayscale to black-white); "colorizing" grayscale movie frames; and reformatting standard television images to display on High Definition monitors. Guidance, in terms of pseudocode algorithms and code fragments, was provided in class to assist students in solving the assigned problems.

2.2.1.2 Phase II - Raytracing Structure

In the next five weeks of the class, the raytracing problem was described and the key elements required in its solution were introduced, including the use of structures, unions, pointers, and recursion. A breadth-first approach with repeated refinements was employed. In this way the students were quickly able to render simple images and then refine them using more sophisticated treatments. Key to success was a carefully defined structure to represent the "objects" in the scene. A typical example is shown in Code 2.1. Note the extensive use of function pointers. These function pointers lead to the object-oriented paradigm by allowing a geometric shape in the image to have associated functions by storing pointers to those functions.

A color is specified as a triple of red, green, and blue (RGB) intensities in the range [0,255]. The functions `ambient()` and `diffuse()` return coefficients representing the degree to which the surface of the object reflects red, green, and blue components of incident light. These functions typically return constant values, but the functional representation supports procedural textures such as a checkerboard floor. An object's `hits()` function is responsible for determining if and where a given ray intersects this object. The

www.manaraa.com

```
struct object {
   struct color (*ambient)();
   struct color (*diffuse)();
   struct color mirror; /* weight on specular */
   void (*get normal)();
   int (*hits)();
   union {
      struct ball ball;
      struct floor floor;
   } config;
struct object *next;
```

Algorithm 2.1: Fundamental Object Structure

`get_normal()` function returns a unit vector normal to the surface at any point. In Phase II, only two types of objects, a sphere (called “ball”) and an infinite horizontal plane (called “floor”), were defined, and colors were limited to grayscale.

A call to trace a ray from a virtual eye point through a pixel begins with an iteration over the object list to find (via `hits()`) the object whose ray-object intersection is closest to the virtual eye point. The color of that pixel is set to the weighted sum of ambient, diffuse, and specular illumination components. The ambient component is a constant unless a procedural texture is in use. The diffuse component is proportional to the cosine of the angle between the surface normal at the intersection point and a vector pointing toward the scene light source. The specular (reflective) component is computed recursively. The incident ray is reflected about the surface normal, and a recursive call to raytrace is made with the intersection point as the new virtual eye and the reflected ray as the new ray direction. The returned value from the recursion is the specular component. Pseudocode for the raytracing algorithm is shown in Algorithm 2.2.

Requirements for the first raytracing project were flexible:

1. The inclusion of at least one light source, two spheres, a checkerboard planar surface, and a sky.

2. The illustration of shadows, diffuse and specular illumination, and anti-aliasing through sub-pixel sampling.

3. The production of an image of at least 1024x768 pixels with an aspect ratio 4:3, with no ratio-induced distortions.

Some impressive images resulted from this phase: Figure 2.1.

المنارة للاستشارات
www.manaraa.com

```
color_t raytrace (ray_t ray, float ray depth){
   if (ray depth > max depth) return(black);
   best distance = +∞;
   for (each object in the scene){
      compute ray-object intersection point, pt;
      if (distance(pt,viewpoint)< best distance){
         record object and pt;
         update best distance;
      }
   }
   if (no object intersected) return(background color);
   add best distance to ray depth;
   set color to ambient color for this object;
   get normal for this object at pt;
   for (each light in the scene){
      if (pt not in shadow){
         compute diffuse component for this pt/light;
         add diffuse color to color;
      }
   }
   if (object has a specular component){
      compute reflected ray;
      reflected color = raytrace (reflected ray, ray depth);
      add reflected color to color;
   }
   return(color);
}
```

Algorithm 2.2: Pseudo-code for raytracing

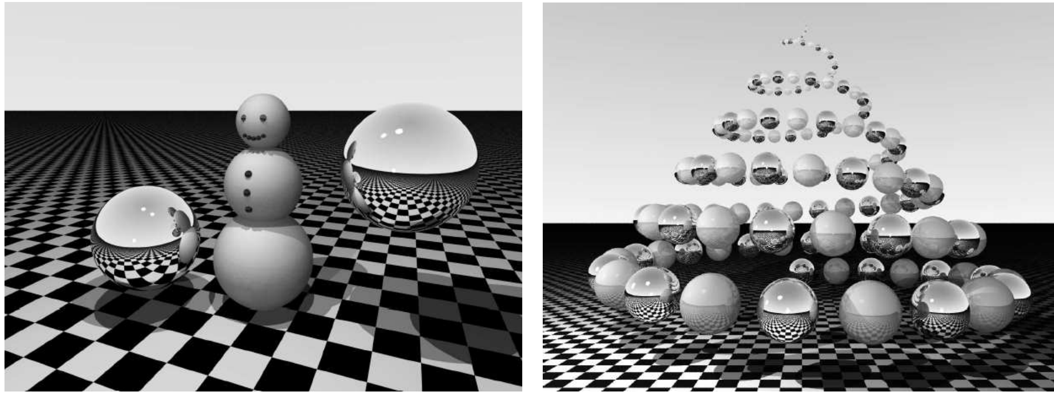
2.2.1.3 Phase III - Raytracing Refinements

In the next four weeks of the course, the design of the raytracing system was extended to include refraction, stereographic projection, and new object types. New types included boxes, quadrics, and surfaces of revolution. Algorithms for the defining functions, `hits()` and `get_normal()`, were derived in class, but the implementation was left to the students. These additions to the task naturally generated discussions of new tools and techniques including dynamic memory allocation, non-trivial linking, and modular program design. With all associated functions grouped in separate files, students found "makefiles" to offer a welcome relief rather than a burdensome extension. Requirements for the second project were again brief and flexible:

1. The inclusion of at least 3 light sources, 2 boxes, 2 spheres, a planar surface, and a sky (unless the scene is completely enclosed by a sphere or box).

2. The illustration of shadows, diffuse and specular illumination, and anti-aliasing through sub-pixel sam-

المنارة للاستشارات
www.manaraa.com



(a) By student S. Duckworth

(b) By student T. Nguyen

Figure 2.1: 2002 CPSC 215 Phase II Student Renderings.

pling.

3. The production of a color image of at least 1024x576 pixels with an aspect ratio 16:9, with no ratio-induced distortions.

2.2.1.4 Phase IV - Scene Specification Language

To permit complex scenes and animations to be defined using external, file-based specifications, a week was spent on additional I/O techniques and the design of an integrated parser. The scene specification language suggested was a highly simplified synthesis of several current formats [66]. Students were free to extend the language, and several did. In the final two weeks, source code simplifications available through the use of C++ were discussed. Because OO is a design paradigm, not a language, and because the students had a large OO design in front of them, the transition was not viewed as a major one. In particular, simplifications available in vector operations (spatial and color) through operator overloading and the advantages of derived classes were easily described. The use of C++ for the final raytracing project was optional, and several took that option.

2.2.2 Discussion

The impressive images and encouraging evaluations of this course and all other $\tau\acute{\epsilon}\chi\nu\eta$ results are in Chapter 7. Some concerns about the size of the problem for a three-hour course were raised and later addressed by moving the raytracer to a four-hour course (Computer Science II) with its prerequisite course

www.manaraa.com

introducing image creation and manipulation. Another possible solution is the use of teamwork. Strongest supporters of cooperative learning [36], [76] admit to drawbacks, and thus any teamwork should be structured in such a way as to encourage peer learning and discourage weaker students from "hiding" in large, strong teams without genuinely getting involved.

While increased motivation was an expected result from the new course, a surprising aspect was the extent of extra work the students actually performed, in many cases more than our graduate students who also write raytracing programs for an advanced graphics course. Several of the undergraduates investigated advanced techniques, such as new object geometries, texturing, and 3D stereograms, on their own. These features were far beyond the requirements for the assigned projects.

2.2.3 Conclusions

A benefit of the $\tau\acute{\epsilon}\chi\nu\eta$ project is the opportunity for faculty to engage undergraduates with discussions of the research that carries their enthusiasm. We conjecture that the benefits to the students will arise not only from the problem-solving orientation of the instruction and the exposure to the vitality of real research problems but also from a newly induced vitality in the instructors.

The positive experiences (discussed more in Section 7) from this course and the observations from the DPA program led to introducing students to computer science using the $\tau\acute{\epsilon}\chi\nu\eta$ approach. While raytracing may seem like a complex topic for second-year students, in fact first-year students now perform raytracing in the four-hour, second-semester CPSC102 course. Unlike the trial CPSC215 course, students in CPSC102 have experience in C and in image processing, and as a result, the class is equipped from the first day to focus on appropriate data structures and algorithms for processing three-dimensional geometry.

المنارة للاستشارات
www.manaraa.com

Chapter 3

Curriculum

3.1 Educational Goal

Research under $\tau\acute{\epsilon}\chi\nu\eta$ led to the synthesis of a new direction in computer science education that draws upon educational theory and epistemology to address problems commonly observed in undergraduate computing programs, in particular, high drop rates, lack of faculty interest in introductory courses, slow development of students, and lack of participation by women and minorities. This new direction has led to the design and implementation of a curriculum in computing that embodies these principles.

Before discussing a new curriculum for computer science, we must identify the goal of the curriculum. The obvious answer is to produce good computer scientists. Unfortunately, this stated goal is rather vague, owing to the use of the word "good." However, it is generally agreed that a good computer scientist is skilled in computational problem solving: writing, reading, analyzing, testing, and debugging computer programs and algorithms. Less obvious but important skills include the ability to work with and learn from other computer scientists, the creativity to conceive new research directions, as well as the ability to pursue such directions. To generalize, a computer scientist must operate at the top of Bloom's taxonomy: analysis, synthesis, and evaluation. Analysis is introduced immediately in computer science with pattern seeking, organization of code, and understanding of hidden parts. Evaluation is important in data structures for comparing approaches and choosing the appropriate solution for a given problem. Finally, synthesis is employed in research by using old ideas to create new ones, by generalizing from given facts, and by predicting and drawing conclusions based on data.

www.manaraa.com

3.2 Problem

The decision to redesign any curriculum naturally flows from the identification of particular problems with current approaches. With the goal and definition of computer science education in mind, what issues does this curriculum redesign address? While this approach likely has flaws of its own, the issues it seeks to address are illustrated by the following symptoms: 1) high drop rates, 2) lack of faculty interest in teaching lower-level courses, and 3) slow development of students into good computer scientists.

For each issue, the symptom will not be alleviated without addressing the greater cause. For example, what causes high drop rates? Lack of programming experience, poor math skills, and poor environment have all been linked to the lack of retention in Computer Science [55]. The most obvious cause is that some students do not enjoy or are not capable of doing computational problem solving. Since professors wish for students to be happy with their fields, it should be considered a positive occurrence when such students transfer to majors better suited for them. Unfortunately, not all students who believe they cannot do (or do not like) computer science base their decisions on valid evidence. Often, first-year students, especially students with no programming experience, feel overwhelmed and alone in the midst of vast amounts of new information. The plight of these so-called "at risk" students [55] is exacerbated when faculty discourage student interaction to prevent cheating. This lack of socialization has been linked to drops [68], since the isolation leads some students to believe that they are the only ones struggling and afraid to ask for help, only to fall further and further behind. Other students jump into programming with alacrity, only to become frustrated by toy programming assignments that do not solve any useful problems and do not produce impressive results for the amount of time invested. In this point-and-click generation, students can be easily discouraged by the basic beginnings used to form the foundation of later education. The problems of both of these groups, the overwhelmed and the frustrated, need to be addressed within a single educational environment.

Lack of faculty interest in lower-level courses can arise from multiple sources. First, faculty, like students, become bored by toy programs, but devising more interesting assignments to reinforce computer science concepts is difficult, especially for early programming courses. Additionally, students in lower-level courses often require a taxing amount of out-of-class support. Finally, lower-level courses seldom benefit the instructor's current research but instead draw needed time and energy away from his or her research interests. All of these problems should be taken into account when a curriculum is prepared.

Finally, slow (or non-existent) student development toward becoming good computer scientists is partially a symptom of the problems with student support. Toy programs, cut-and-paste code, and poor

www.manaraa.com

motivation promote laziness and apathy in students, harming the educational environment. When the students are engaged and excited about learning, their academic development will greatly improve. An additional factor is that not all attributes of good computer scientists are currently emphasized in typical curricula. Areas such as teamwork, creativity, and research skills should receive more attention in order to encourage breadth in computer science as well as depth.

3.3 Curriculum Structure

With the stated issues in mind (student isolation, frustration, boredom, and faculty burden), the goal of $\tau\acute{\epsilon}\chi\nu\eta$ (and more broadly, Quest-Oriented Learning) is to *create a focused, social learning environment that motivates, educates, and broadens students at all levels without undue burden to faculty*. With the proper environment and problem set, students can become excited and inspired to seek the most out of a course without a burden on the instructor to manufacture the desire in the students.

Each $\tau\acute{\epsilon}\chi\nu\eta$ course is structured by a large, semester-long graphics project. The graphics project is selected to fit concepts appropriate for the course level and content and is broken into phases. From the first day, the target project is the focus of each phase of the project and each new piece of information. From the first phase, students are not provided with starter code or advanced libraries, since they are building mental models of computing. In line with constructivism, no advanced concepts such as objects and abstraction are introduced before students can fully understand fundamental building blocks such as looping, branches, and memory allocation [51]. Research indicates that not only are students experiencing difficulty grasping complex abstract topics, but basic programming structures are receiving weakened coverage to afford time to explore object-oriented constructs. Students experience difficulty understanding variables, parameters, and loops. "Why should we believe that they can construct a viable model of an object such as a radio button?" [6].

The choice of graphics as the central focus of the projects is not arbitrary. Graphics problems afford special educational experiences not offered by other studies. Besides the obvious visual feedback afforded by graphics projects, these projects also leave room for creative expression, demonstrate the need for advanced mathematics, meet students' expectations of visual components, prepare students for the increasingly visual world, and tap into areas of problem-solving otherwise untouched. Creative expression is emphasized by the fact that some graphical results are technically impressive but artistically boring. The visual component present in computer graphics adds a dimension to the evaluation of programs. "Unlike other areas of computer

المنارة للاستشارات
www.manaraa.com

science, algorithms must be considered not only for time and memory usage, but for their visual effect" [77]. This opening for variation among "correct" answers lets the students hone their own creative skills, and link left-brain and right-brain activity, the scientific with the aesthetic.

Graphics projects reveal the necessity of advanced mathematics, such as calculus, linear algebra, and trigonometry, through exploration of the relationship between mathematics and the form, location, orientation, and motion of objects in the physical world. Without tangible problem focus, such topics in mathematics often seem irrelevant to students' lives.

Graphics meet the expectations of a generation of students accustomed to video games and hand-held computers. Graphics projects create the interest that Graphical User Interfaces are meant to create [62], without introducing the complexity of object-oriented programming and abstraction. Since first-year students are best suited for concrete knowledge [59], it is best to avoid beginning CS with abstraction. See Chapter 5 for an in-depth discussion of the selection of the type of the introductory language.

Cunningham exposes the visual aspects of problem solving that are often overlooked by the omission of required graphics education, including mental visualization, kinesthetic (movement) analogies, aesthetics, physical modeling, and the ability to visually communicate [12]. He points out that "visual communication is simply not part of their general education pattern. We use little visual communication in our teaching except for the diagrams that help students see the patterns of our processes; our students use almost no visual communication in expressing their learning." Cunningham makes the case that graphics courses teach and reinforce problem solving skills, and similarly we maintain that problem solving skills are taught well through graphics. Thus, Cunningham's approach is to emphasize problem solving in graphics courses, while we emphasize placing graphics in problem solving courses.

3.4 Educational Approach

3.4.1 Foundation

The educational foundation of the $\tau\acute{\epsilon}\chi\nu\eta$ curriculum draws from the work of Piaget in developmental theory and cognitive constructivism, and Martinez in intentional learning, as well as from problem-based learning reinforced with visual feedback.

Starting with cognitive constructivism, Piaget's genetic epistemology describes the natural development process of children. The progression in development is a natural part of all people, young and old, and

المنارة للاستشارات
www.manaraa.com

it is important for education to accommodate this innate learning process. Piaget breaks down the process as he observed it in children. A baby begins the exploration of his environment using simple sensorimotor skills called schemas. These schemas are the starting point for the creation of knowledge. For any individual, learning begins at some level that will be built upon.

1. The individual applies the schema to different objects: assimilation. For example, a baby applies the same action to different objects; e.g. after putting a rattle in his mouth, he may try putting a watch into his mouth. Similarly, when a person is confronted by a new object, that person draws from his old experiences to determine how to deal with the object. For example, if the object is protruding from the center of a door but does not look like a typical doorknob, the person will still attempt to rotate it and pull the door open.

2. When a schema does not work for a given object, the individual tries an adapted version to accommodate the object: accommodation. If the baby has a beach ball that cannot fit into his mouth, he may instead grab it and drool on it. Similarly, when an adult discovers that the door handle does not respond to turning and pulling, he attempts various different methods of unlatching, such as lifting, pressing, or tilting the handle, until he has appropriately accommodated the new door handle.

Using a combination of assimilation and accommodation, babies (and people) advance their knowledge and competence [58].

Piaget's genetic epistemology indicates that the learner is not a blank slate to be dictated upon, as once thought. Instead, even a developing toddler experiments and reasons out new information for himself. When confronted with unfamiliar objects, the individual constructs a mental model to assimilate or accommodate it. This development process is an active acquisition of knowledge, known as constructivism. "Constructivism is a theory of learning which claims that students construct knowledge rather than merely receive and store knowledge transmitted by the teacher" [6]. Piaget contributed much to creating the constructivist development model. The theory probably owes its origins to Immanuel Kant [37] and Jean-Jacques Rousseau [65]. Kant believed that our innate mental structures are used through the interpretation and organization of experiences. Rousseau suggested that human genes were the basis of intellectual development and that interaction with the environment was the basis for constructing understanding. As with all learners, computer science students experience and interpret the results of their work and build personal models of the behavior of objects of interest. Sometimes students have misconceptions that are later exposed through contradictory results. Gradually students correct their mental models, moving closer to an accurate model,

المنارة للاستشارات
www.manaraa.com

based on their experiences. An "accurate" model is a conceptual framework that facilitates interaction with the environment and prediction of outcomes.

A goal for the curriculum is to support this learning process through the exposure to real-world problems in order to facilitate the construction of knowledge. Such problems allow students to test and adjust mental models while developing new computational solutions through assimilation and accommodation. This constructivist learning process is well supported by problem-based learning (PBL). Since its inception in the 1960's as a way to improve training of physicians [5], problem-based learning (PBL) has been widely recognized as a successful method of improving retention and skills acquisition. In computer science, PBL is employed to teach subjects as varied as design patterns [11], assembly language [78], artificial intelligence [10], and CS1 [28]. Rather than having educational concepts drive the curriculum, PBL instead uses realistic problems to drive the educational experience. For example, people do not study various types of door latches, but instead solve the problem when they are confronted by a closed door with a new latch that they are motivated to open. In other words, problem-based learning "is a learner-centred approach in which learning episodes are motivated by an initial problem that bears some resemblance to 'real world' problems" [28]. With this approach, learning occurs when students encounter roadblocks on the path to the final goal (such as a large class project), and they are forced to make accommodations to circumvent the roadblocks. Through PBL, students experience opportunities for constructing new knowledge.

While students can and do learn computer science in ways completely counter to PBL, PBL is a better approach to motivate that learning. For example, people can learn Japanese by enrolling in a university course and memorizing vocabulary and grammar. However, these people would learn much more quickly and be much more serious about the process if they moved to Japan for a year. Suddenly their learning is prompted by a pressing need to communicate. After a year of studying in Japan, a person may not know as many grammar rules as his textbook-studying counterpart, but he will be much more fluent and capable of conversation. This is not to say that learning grammar and vocabulary is unnecessary. Quite the contrary, the person living in Japan will be more driven to self study than someone simply trying to pass a language class. In the same way, the immersion process invoked by PBL better prepares students for work in computer science and even encourages them to study on their own.

The *τέχνη* project uses constructivism as a theoretical basis for learning and problem-based learning to motivate the learning experience. Fundamental tenets of the approach are that a visual problem domain will most quickly capture the attention and interest of students who have grown up in a society that is increasingly visually oriented, that a connection between scientific and artistic components will stimulate both

المنارة للاستشارات
www.manaraa.com

deductive thought and creativity, and that toy problems would be of little value in effecting the principal desired accommodation, an ability to solve real problems.

We wish to move students closer to becoming intentional learners. "We use the term intentional learning to refer to cognitive processes that have learning as a goal rather than [as] an incidental outcome" [7]. Rather than have students work toward the goal of passing a class, students should be on a quest to become experts in computer science. Another way to put it is that an intentional learner is a person who learns when outside factors make it unnecessary to do so. Although these types of students seem rare, it is not wholly unrealistic to suggest that a good curriculum can cultivate intentional learners. Every child begins life enthusiastic, persevering, and ready to learn. A toddler works hard to improve his pronunciation, his ability to walk and run, and his understanding of the physical universe around him. Somewhere in life, students lose the desire to learn, possibly due to unmotivated, seemingly unnecessary busy work. Nevertheless, at the college level, most students choose for themselves what they wish to learn and can again become enthusiastic about the experience.

3.4.2 Approach Clarification

After establishing the basis of *τέχνη* and describing its components, we must point out what *τέχνη* is not. First, *τέχνη* is not a full implementation of constructivism. Constructivism taken to its logical end converts the teacher into a facilitator while asserting that there is no absolute truth, merely more or less useful models. While it is important for students to reason out problems on their own, college professors hold a wealth of information that should be passed onto the students. Obviously, teachers can lead students toward the solution, but sometimes in classroom environments, there is not enough time to have students rediscover for themselves everything that years of work in the field have uncovered. Also, especially in computer science, there is absolute truth in certain aspects of the discipline, e.g., how a computer stores data, what happens during execution, etc. While this approach is not a full adaptation of constructivism, the key concept that students build knowledge incrementally working toward an accurate model is foundational to the choice of incremental semester-long projects that begin at the bottom and build upward.

The *τέχνη* curriculum is also not a full implementation of PBL. Once again, *τέχνη* draws the key ideas from PBL without applying it in ways some predecessors have. This implementation is in line with the problem-centered learning variation [43], in which teacher is a resource instead of a coach, and the information is presented to the students in an organized way, but all the learning revolves around the problem.

Finally, the *τέχνη* curriculum pulls important key points from Margaret Martinez's System for Inten-

www.manaraa.com

tional Learning and Progress Assessment (SILPA), but it is not a full implementation of intentional learning. SILPA has the following six features: 1) domains of knowledge: enunciation of what an expert in the field knows and can do, 2) multidisciplinary lesson plans, 3) role differentiation: allow the students to be learner, teacher, and researcher, 4) practice and feedback 5) progress assessment, 6) multidimensional interaction: students manage the learning process to achieve their goals [47]. The presentation of the domains of knowledge exists in $\tau\acute{\epsilon}\chi\nu\eta$ by the statement of a real-life problem that is the goal of the given course. Role differentiation occurs in pair design, reports, and open problem discussion (See Chapter 5 for explanation of these supporting features). Practice and feedback are provided through the course and laboratory assignments, and nearly instantaneously with online coding practice. Progress assessment is always available to students. Nevertheless points 2 and 6 are not incorporated because we believe they would detract from the overall framework, the problem scaffolding that we supply.

This new approach fuses key points from Piaget, Kant, Rousseau, Martinez, PBL, and the visual problem domain into the $\tau\acute{\epsilon}\chi\nu\eta$ learning approach with exciting results, as it is used to create a focused, social learning environment that motivates, educates, and broadens students at all levels without undue burden to faculty.

3.4.3 Quest-Oriented Learning

3.4.3.1 Name

There is no single approach on which the $\tau\acute{\epsilon}\chi\nu\eta$ curriculum relies. Beginning with the union of art and technology, $\tau\acute{\epsilon}\chi\nu\eta$ has grown to become a synergistic blend of problem-based learning, intentional learning, constructivism, careful problem selection from the visual domain, and developmental theory. For a name encompassing all the components of this approach, but generalized for other learning environments, we suggest "Quest-Oriented Learning" (QOL).

Understanding the name choice requires understanding precisely what a quest is. A quest is defined as "A chivalrous enterprise in medieval romance usually involving an adventurous journey" [Webster's Seventh New Collegiate Dictionary, 1965], "a seeking or inquiring" [Webster's II New Riverside Dictionary, 1984]. The concept of a quest implies a journey with an unknown path taken to achieve or obtain something. When a quest is begun, the processes that will be required to complete are unknown. There will be skills to acquire, tools to obtain, roads to discover, challenges to face, and obstacles to overcome. There is no handbook completely covering all the information needed to achieve the goal. Instead, the traveler must

المنارة للاستشارات
www.manaraa.com

find direction from a multitude of sources and steadfastly struggling through each challenge to reach the final destination. A quest is an active process that cannot be replaced by a monologue from someone who has completed a quest of his own. Rather, the traveler is simply inspired by previous achievers to embark on the journey.

In terms of the $\tau\acute{\epsilon}\chi\nu\eta$ curriculum, how is becoming a computer scientist a quest? Becoming a good computer scientist is not something that can be handed to the students through lectures or textbooks. If success could be achieved through the memorization of lists of principles, textbooks would be certainly more than adequate, but there are skills as well as new thought processes that the students must develop. The educator inspires and equips the student to begin such a journey, but learning is an active process that the students perform. To obtain the knowledge, skills, and experience needed to excel in any field, students must learn to actively seek these components in the manner of a quest.

QOL incorporates PBL obviously. Additionally, the student going on a quest must be an intentional learner. He has chosen to undertake it through his own interest which spurs him to go beyond the requirements in order to learn more. As he works toward his goal, he learns what is necessary and builds tools incrementally, from the ground up, or in a constructivist manner. With the schema he has at the beginning, he gradually develops new skills necessary, in line with Piaget's Genetic Epistemology. Visual problem solving is incorporated, since the hero of the quest is operating in the real world with visible results to his actions. The quest is a real-life struggle. The more educational environments model the real world, the better the resulting education.

3.4.3.2 Components

Broadening the concept of QOL, $\tau\acute{\epsilon}\chi\nu\eta$ can be considered an application of QOL to undergraduate Computer Science Education. It is highly likely that other fields and levels of education could apply QOL to bring about improved student attitude, performance, and enthusiasm. Other applications of QOL should hold to the same principles present in $\tau\acute{\epsilon}\chi\nu\eta$.

The goal of Quest-Oriented Learning is to create a focused, social learning environment that motivates, educates, and broadens students at all levels without undue burden to faculty. QOL implementations should include this goal and the following components:

- A Fundamental Project that is:
  - Large enough to require multiple phases.

www.manaraa.com

  - The focus of the entire course of education.
  - A real-world problem that incorporate visual problem solving.
- Opportunities for excellence above covered material and requirements.
- Opportunities to research new information and teach others.
- Encouragement for immediate commencement of the quest.
- Opportunities to repeatedly hone newly-acquired skills.
- Open problem discussion amongst others on the quest.
- Use of external resources to solve the basic project.

المنارة للاستشارات
www.manaraa.com

Chapter 4

Related Work

$\tau\acute{\epsilon}\chi\nu\eta$ courses are structured by semester-long graphics-related problems to be solved by students without starter code or special libraries. While no introductory courses in computer science follows this model, there are those that rely on semester-long projects, and those that use graphics projects. Below is a literature review of these categories.

4.1 Semester-Long Projects in Introductory Courses

Most work with structuring courses around a semester-long project has been done with upper-level and graduate courses. A handful of papers discuss the viability and benefits of semester-long projects in CS1 and CS2. Bareiss's desire for CS1 students to have final projects that they could tackle entirely by themselves and yet were complex enough to be interesting, enjoyable, and challenging led her to assigning the sequential creation of the Othello game [4]. While no results are discussed and the display type of the game is unclear, the paper supports the belief that CS1 students can and should learn through large, realistic projects that demonstrate the benefits of good programming techniques and initiate student creation of a portfolio. Outside of $\tau\acute{\epsilon}\chi\nu\eta$, Bareiss's incorporation of a semester-long project into CS1 appears to be unique.

Semester-long projects have been used in some CS2 courses, including a maze generation and traversal program [73], a five-phase day planner application [70], and a tennis competition bracketing system [26]. The maze generation/traversal program incorporates matrices and linked lists, and provides visual feedback with command prompt ASCII displays of the maze and chosen path. The day planner application grew out of the instructor's concern that student assignments were unconnected to the real world and required no mainte-

للاستشارات المنارة
www.manaraa.com

nance, review, or real testing. Thus, students were assigned a realistic program that had to be maintained and updated throughout the semester. The tennis tournament program was a semester-long task, but four other toy programs were interlaced throughout the semester.

All four courses recognized the need for "real-world" problems presenting the challenges of code design and maintainability while grabbing the attention of the students.

4.2 Graphics in Introductory Courses

In most cases, graphics are infused into introductory courses to motivate students to learn from fun, challenging, "real" problems. Many of the projects involve image processing, which seems to be a tried-and-true method of introducing CS concepts. In one study [74], student enjoyment was charted as "very high" in a CS1 course taught using image processing. Using the Java Swing API, students created "MSPaint-like" effects, first with grayscale images and then with color images. The students were required to implement 15 effects and were given compiled image extraction and display libraries that provided them with 2D arrays of data to manipulate. Similarly, Burger [9] describes the classroom use of the Java Image I/O API and his own Image class to simplify the modification of grayscale images, including intensity changes, horizontal and vertical flip, 90 degree rotation, color inversion, conversion to black and white, histogram generation, and various filters (blur (3x3), gauss (3x3), sharpen (3x3), and Prewitt edge enhancement). It is unclear if his approach has been used in introductory courses.

An early proponent of graphical results, Robergè suggests data structures projects "intended to show that programming projects with visual impact can be constructed from the traditional elements of a wide variety of courses, in a wide variety of computational environments" [61]. His suggested projects (note card system, presentation manager, graphics editor, digital logic circuit laboratory kit, database system, routing utility, hypertext system, and menu system) were implemented for a text-based command-prompt display, but they were meant to address the already growing frustration and disappointment of students in basic programming assignments and instead emphasize real-world problems and visual engagement.

Another early and often-cited paper describes the use of downloaded gray-scale NASA Martian planetary images as realistic sources of data for students to manipulate [23]. Students were provided with functions to allow them to set single pixel values, read files, produce sounds, and manipulate bits. Given these provided libraries, students enhanced gray-scale images (by scaling the values to be across the full spectrum), equalized a histogram of the values, played music corresponding to the values of the pixels, and discovered

المنارة للاستشارات
www.manaraa.com

(planted) covert messages in the images. These activities were suggested for the laboratory setting and are an interesting way for students to discover useful applications of computer science.

In order to capture student attention in CS1 and CS2, Jimènez-Peris et al. explore real-world, challenging problems [35]. Although the majority of the projects suggested are graphical games (tetris, asteroids, card games, etc.), two suggested assignments involve image processing in CS1 and CS2: image edge detection and image compression using trees.

A study at Duke University found that "animations and interactive graphics generate student interest and enthusiasm, which usually translates into better comprehension and mastery of the material in our courses" [2]. Complexity of animation and visualization programs were hidden from students via provided object classes that drove the animation process. Students manipulated "balloon" (filled circle) movements, tracked frog (filled circle) movements, emulated a cardioverter/defibrillator monitor, built animated business histograms, and optionally but most notably, compressed portable bitmap images (PBM format).

Although it is outside the realm of the college introductory computer science course, the "Fun With Faces" [52] project educated ninth and tenth-grade students about image processing, and all students were excited by the interesting visual effects they were able to create with pictures of themselves. Work was done through the Scion Image program, which allows image capture and modification.

The importance and benefits of including graphics in required Computer Science courses is discussed in several papers. Hunt lists the fields that rely on image processing: "sports broadcasting, mail delivery, military target acquisition, satellite imaging, robotics, medical imaging and the traditional print industry" [34]. He advises the integration of image processing into existing courses and describes his Easy-BufferedImage class, built on the Java Image processing libraries as a way to incorporate image processing into CS1 and CS2. Cunningham and Shiflet emphasize the need for a graphics background in Computational Sciences and suggest the use of Matlab, Mathematica, Maple, Excel, STELLA, and OpenGL to incorporate graphics into upper-level, undergraduate courses [13].

Tashakkori utilizes graphics as a means to generate undergraduate interest in research and describes a post-CS2 class focused on the study of digital image processing [69]. The class encourages self motivation and identifies early on each student's interests as they pertain to image-processing research.

All of the projects described above are aimed at educating students in an exciting and visual way that gets their attention and challenges them to reach beyond expectations. The $\tau\acute{\epsilon}\chi\nu\eta$ approach reaches beyond these previous approaches by placing large projects at the center of learning and requiring students to write all code from the ground up without the use of starter code or special libraries.

المنارة للاستشارات
www.manaraa.com

Chapter 5

Implementation

The implementation of $\tau\acute{\epsilon}\chi\nu\eta$ involves the organization of the courses into phases, the laboratory environment, additional supporting features for the courses, and the decision of which languages to use. Section 5.1 discusses the phases of each of the first four courses of $\tau\acute{\epsilon}\chi\nu\eta$, Section 5.2 discusses the Pair Design laboratory structure and other supporting features, and Section 5.3 explains the choice of C for the introductory language.

5.1 Course Phases

Below are descriptions of the first four courses in the Computer Science curriculum and the description of an additional, single-semester course covering CS1 and CS2. Each course is structured around a large, semester-long, graphical project. The projects are broken down into as many as twenty phases (depending on the level of the course) to demonstrate a progression. Multiple phases should be grouped into single assignments. The number of actual assignments each semester is at the discretion of the instructor, as well as any modifications to improve the project in the class.

5.1.1 Computer Science I

CS1 is structured around the implementation of a color transfer program that will apply the color scheme from one image to another. The project will be done in phases, beginning with the simple creation of an image file.

المنارة للاستشارات
www.manaraa.com

5.1.1.1 Single-pixel image

The creation of a 1-pixel image in Portable Pixmap (PPM) format that is printed to standard out. Required Material: text editor, C compiler, library functions, IO functions, preprocessor directives, the main function, PPM format, binary data, ASCII data.

5.1.1.2 Large, solid image

Printing an 800 by 600 PPM format image to standard out. Required Material: variables, data types, variable declaration, variable assignment, conditional expressions, variable incrementation, counted loops.

5.1.1.3 File copy

Reading a file (image file) and printing it back out. This phase draws from previous knowledge and is working toward the goal of reading in images, manipulating them, and outputting them. Required Material: standard input, file streams, unsigned char, conditional loops, bytes, binary data.

5.1.1.4 Header parsing

Reading a PPM image file, and outputting the width, the height, and the total number of pixels in the image. Obtaining this information requires knowledge of the image header format, unlike the mere copying performed in the previous stage. Required knowledge: reading integers, addresses, character comparisons, nested loops, if statements, un-reading data, error conditions, functions, function return values, boolean expressions, multiplication, test for white space, stdlib.h, ctype.h.

5.1.1.5 PPM file copy

Reading a PPM image file, printing it back out, and outputting the width, the height, and the total number of pixels to standard error. Since this phase does not have a great deal of new information, it is a good time to re-factor the code to have better organization and a header file. Required knowledge: outputting to standard error, addresses, pointers, header files.

5.1.1.6 Single-pass modification

Reading a PPM image file and making a modification to the colors. This phase is leading to the final goal of modifying an image's color scheme to match another image. The phase provides practice modifying

المنارة للاستشارات
www.manaraa.com

image data and may be anything, including changing one channel's value to be a set value, lightening or darkening all values, converting the image to grayscale, inserting scan lines, fading the colors, monochrome, etc. This is a phase where students can be creative or the instructor may simply choose one modification to require. Additionally, more difficult modifications may receive extra credit. The instructor is not expected to explain how to do each option, but instead the students should inquire after the approach. Since more difficult modifications are worth more points, students should be motivated to learn how to do them. Required knowledge: information regarding the file modification that will be done.

5.1.1.7 Stored-data modification

Reading a PPM image file into an array and making a modification. The storage of the entire image at one time is necessary for the final goal of reading in two images entirely and manipulating one based on the other. Modifications to the image may now depend on knowledge of more than one pixel at a time, such as resizing, tiling, flip, rotation, blur, and sharpen. This is another possible place for creativity, the improvement of the previous phase, or combination with the next phase. Required knowledge: arrays, `fread()`, `fwrite()`.

5.1.1.8 Variable input image size

Reading a PPM image file of any size into an array and making a modification. Required knowledge: dynamic memory allocation (single dimensional or multi-dimensional).

5.1.1.9 Conversion to CIELAB

Reading a PPM image file, converting it to CIELAB colorspace, modifying the colors, and printing it back out. CIELAB color space is a three dimensional color space in which pairs of colors at equal distances from one another are perceived to be at equal distances from one another. It is sometimes called perceptually uniform color space. The transformation between RGB space and CIELAB space is arithmetic using matrix multiplication. The modification can be anything from modifying one component to produce an interesting effect to implementing a color balancing algorithm. The matrix multiplication for this phase may be covered in the laboratory setting or provided to the students. Required knowledge: matrix multiplication, knowledge of RGB to CIELAB format.

المنارة للاستشارات
www.manaraa.com

5.1.1.10 Command-line input

Reading a PPM image file specified on the command-line, converting it to CIELAB format to balance the colors, and printing it back out. Use of command-line arguments is necessary for the final phase when two images must be specified as input. Required knowledge: Command-line arguments, strings.

5.1.1.11 Color transfer

Reading in two PPM image files, converting them to CIELAB format, computing the means and standard deviations on each file, adjusting the values of the first image to the values of the second based on the paper [60], and printing out the resulting image. Required knowledge: computation of mean and standard deviation.

5.1.2 Computer Science II

CS2 is structured around writing a raytracer: a technique for rendering realistic images by modeling the path from a given starting point to geometrically-specified objects in Euclidean three-space. To create an image with the raytracer, start at a specified "eye" point, shoot one or more rays in the direction of each pixel, compute the color resulting from any intersections, and output that color.

5.1.2.1 Single-color image generation

The creation of a solid-colored, PPM formatted image file output to standard out. Although it is not yet raytracing, the program can still be structured in a way compatible with later raytracing. e.g. Create a raytracing function that merely fills the array with the appropriate number of pixels. Required knowledge: arrays, array access, data (bytes) functions, binary (raw) data and ASCII output, PPM image format, and I/O redirection.

5.1.2.2 Image of specified size

Creating an image of any specified size. Required knowledge: command-line arguments, dynamic memory allocation, and string (char*) to integer conversion.

المنارة للاستشارات
www.manaraa.com

5.1.2.3 Simplified sky

Creating an image of a sky of any specified size. Although it is a basic step, this phase introduces the fundamental structure of the raytracer. Required knowledge: header files, forward declarations, structures, enumerated types, function pointers, scaling, procedural texturing and projection (2D image location to 3D coordinate conversion).

5.1.2.4 Spheres

Creating an image of any specified size with a sky and any number of spheres. Required knowledge: unions, definition of a sphere, ray-sphere intersection, the quadratic equation, macros, the dot product, point subtraction, comparing floats/doubles to zero.

5.1.2.5 Plane

Creating an image of any specified size with a sky, any number of spheres, and a horizontal floor. Required knowledge: Ray-floor intersection.

5.1.2.6 Checkered plane

Creating an image of any specified size with a sky, any number of spheres, and a checkered floor. The checkered floor will have two alternating colors applied procedurally. Required knowledge: Mathematical function $floor()$ and modular arithmetic.

5.1.2.7 OO tracer

The use of an OO language (C++) to create an image of any specified size with a sky, any number of spheres, and a checkered floor. Converting to C++ can be postponed, but it is not suggested. At this point, students have experienced object-oriented programming, although it is implemented via structures, unions, and function pointers. Staying with procedural programming from this point on will not add any knowledge, and beginning with objects will introduce many new concepts with ample time for practicing programming in a new language. Additionally, an early conversion to C++ reduces the amount of code that will need to be rewritten in C++. If C++ is introduced late in the class, the data structures class should introduce objects or students should be advised to take an Object-Oriented class before the data structures class. Covered (but not required) knowledge: C++ classes, C++ inheritance, virtual methods, purely virtual methods, references,

المنارة للاستشارات
www.manaraa.com

static methods, destructors, operator overloading, anonymous structures, constructor initialization, default parameters, constant member functions, typedef, iostreams, and make files.

5.1.2.8 Shadows

An OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, and has shadows. The lights that cast the shadows will be child classes of the Sphere class. Required knowledge: protected attributes, static local variables, and diffuse lighting.

5.1.2.9 Weighted, diffuse contribution

An OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, has shadows, and modulates light contribution based on light source distance and angle. The cosine of the angle is used for computing the light contribution, and will be obtained using the dot product method. Required knowledge: distance formula, normalizing a vector, Sphere normal, and Lambert's cosine law.

5.1.2.10 Weighted, light contribution

An OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent lighting, and overall light intensity attenuation with distances. Although the attenuation of light is not necessary, the addition of a double variable tracking the distance light has traveled is necessary for specular lighting. (If light is not attenuated with distances, the ambient contribution could be reduced. Either approach is fine.) Required knowledge: quadratic attenuation of electromagnetic radiation.

5.1.2.11 Reflectivity

An OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent lighting contribution, overall light attenuation, and specular reflectivity. Reflectivity is produced in a raytracer by bouncing rays off reflective objects and recursively tracing their paths. Required knowledge: bouncing a ray with the law of reflection, recursion.

المنارة للاستشارات
www.manaraa.com

5.1.2.12 Anti-aliasing

An OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent lighting contribution, overall light attenuation, specular reflectivity, and anti-aliasing. (Anti-aliasing is the technique of minimizing the distortion artifacts known as aliasing when representing a high-resolution at a lower resolution.) The edges of the spheres, and the reflections in them may have sharp, stair-stepped edges that do not adequately represent the appropriate curved shapes. Anti-aliasing techniques will smooth the edges in the images. The method of anti-aliasing used here is to capture multiple samples for each pixel by using pseudo-randomly jittered sample points as ray directions through each pixel. These jittered traces are averaged to determine the final pixel value. The result of using the average of multiple, jittered traces is a blended final pixel value that smoothes transitions among colors in the image. The method for generating the random sample points per pixel may be provided to the students to reduce complexity. Covered knowledge: random number generation, a method of generating random 3D points that are restricted to a small, 2D plane segment (pixel).

5.1.2.13 Box generation

An OO program that creates an image of any specified size with a sky, any number of spheres, any number of boxes, a checkered floor, shadows, distance/angle dependent lighting contribution, overall light attenuation, specular reflectivity, and anti-aliasing. Boxes in this raytracer are defined as 3D cubes whose sides are aligned with the x, y, and z axes. Thus, a box is defined by two x,y,z coordinates: a minimum xyz value designating the left, lower, near corner, and the maximum xyz value designating the right, higher, far corner. Required knowledge: a fast ray-box intersection algorithm, comparison of doubles.

5.1.2.14 Linked-list object storage

Changes the storage of the objects in the scene to a linked list. The purpose for this data structure alteration is support for the next phase, in which the objects in the scene are read from a file. Required knowledge: linked lists, iteration.

5.1.2.15 Input file scene specification

A description of the scene to raytrace is read in from a file. Optionally, students may use the input operator `operator>>` to read in information. Required knowledge: file I/O, and optionally friend functions

المنارة للاستشارات
www.manaraa.com

and the input operator.

5.1.3 Data Structures

The course titled Algorithms and Data Structures is structured around the implementation of photon mapping: an augmentation to a raytracer that supports global diffuse illumination with diffuse color bleeding, caustics, and participating media. The first part of photon mapping is emitting photons from the lights into the scene and storing them when they hit non-reflective objects. The second part is rendering the scene using statistical techniques to extract lighting information from the maps. Photon mapping specifies sampling of photons mapped in 3D space, which specifies use of kd-trees (balanced or unbalanced), maxheaps, and use of sorting algorithms. Students will implement photon mapping to produce diffuse inter-reflections and caustics.

5.1.3.1 Random array-based map

Lighting with a photon map composed of photons (in an array) randomly-placed on a surface. The benefit of randomly-placed photons is the ability to get visual results from the beginning before the entire algorithm is implemented. Required knowledge: random numbers, beginnings of runtime complexity, nearest neighbor function, illumination algorithm.

5.1.3.2 Array-based unreflected photon map

Lighting with a photon map composed of unreflected photons from a single point light stored in an array. The runtime will still be remarkably slow. Required knowledge: photon emission algorithm.

5.1.3.3 Array-based, unreflected photon map with heap utilization

Lighting with a photon map composed of unreflected photons in an array traced by nearest neighbor with a maxheap. Required knowledge: heaps and binary heap structure, complexity

5.1.3.4 Unreflected photon map with kd-tree and heap utilization

Lighting with a photon map composed of unreflected photons stored in an unbalanced kd-tree. Required knowledge: kd-tree, complexity

المنارة للاستشارات
www.manaraa.com

5.1.3.5 Basic photon mapping

Lighting with a photon map composed of photons that have been reflected, transmitted, or absorbed using Russian roulette to statistically determine the fate of each photon. If 50% of photons are absorbed after the first intersection, another 50% of those reflected or transmitted should be absorbed after the second intersection. Required knowledge: use of reflection and refraction, color bleed.

5.1.3.6 Photon mapping with projection map

Lighting with a photon map composed of photons that we traced based on a projection map. The projection map limits photons emitted to directions that will lead to an intersection with an object. Required knowledge: matrices

5.1.3.7 Balanced kd-tree photon mapping

Lighting with a photon map that is a balanced kd-tree. Required knowledge: balancing algorithm.

5.1.3.8 Photon mapping with caustics

Lighting with a photon map and a caustic photon map. Caustics are effects caused by light passing through a refractive object and focusing to a strong intensity that causes highlights on another object. A large number of photons should be emitted toward refractive surfaces to generate good caustics. A separate caustic photon map should hold the resulting photons.

5.1.3.9 Photon mapping with ellipsoid nearest neighbor algorithm

More accurate lighting effects are achieved by using an ellipsoid obtained by compressing the lighting sphere in the direction of the surface normal. This modification means that photons incorrectly used at edges and in corners will be minimized. Required knowledge: compression of sphere in the direction of the normal.

5.1.3.10 Photon mapping with filtering

More accurate lighting effects are achieved through use of a 2D Gaussian filtering. Filtering reduces blurriness and leaked photons by increasing the weight of photons that are close to the point of interest. Required knowledge: Gaussian filters

المنارة للاستشارات
www.manaraa.com

5.1.3.11 Photon mapping with multiple lighting types

Lighting with multiple lights and varying light types. Required knowledge: methods of emitting photons from different light shapes.

5.1.3.12 Photon mapping with participating media

Inclusion of participating media, such as fog. This topic will likely need to be an optional challenge for more advanced students and involves the creation of a volume map and use of ray marching and a volume radiance estimate.

5.1.4 Object-Oriented Design Course

The course titled Tools and Techniques for Software Development, or CS4, is structured around the creation of a GUI-based, networked chess game, using Java.

- ASCII checkerboard Phase “zero” can be completed as the first laboratory assignment. It is the ASCII printing of a set of checkers pieces with “x” for black pieces and “o” for white. This phase is more or less the Java “Hello, World!” program, but it already gets students thinking about board layout. Required Material: Introductory Java, main method.

5.1.4.1 GUI-based checkerboard

The creation of an empty, GUI-based checkerboard. Required Material: Java (Swing) graphics, inheritance, overriding methods, invoking parent methods.

5.1.4.2 Flat, colored pieces

The creation of a GUI-based checkerboard set correctly with colored, filled circles. Required material: Graphics2D drawing tools.

5.1.4.3 3D pieces

The creation of a GUI-based checkerboard set correctly with smooth, 3D-looking pieces. Required knowledge: Anti-aliasing command, gradient paint tool.

المنارة للاستشارات
www.manaraa.com

5.1.4.4 Moving pieces

The creation of a GUI-based checkerboard that allows pieces to be dragged to any square. Required knowledge: Mouse events.

5.1.4.5 Legally moving pieces

The creation of a GUI-based checkers game that allows pieces to be moved to valid move locations (diagonally forward). Turns do not matter yet. No new knowledge is needed.

5.1.4.6 Legally playable pieces

The creation of a GUI-based checkers game that allows pieces to be moved or singly-jumped legally. Turns do not matter yet. No new knowledge is needed.

5.1.4.7 King creation

The creation of a GUI-based checkers game that allows pieces to be moved and singly-jumped, and crowns pieces that reach the last rows. Turns do not matter yet. Required knowledge: A way to draw stars on pieces.

5.1.4.8 King plays

The creation of a GUI-based checkers game that allows pieces to be moved, singly-jumped, and crowned, and allows king plays. Turns do not matter yet. No new knowledge is needed.

5.1.4.9 Forced multiple jumps

The creation of a GUI-based checkers game that allows pieces and kings to be moved, jumped, crowned, and requires multiple jumps to be completed. Turns do not matter yet. Required knowledge: Mouse moved event.

5.1.4.10 Turns

The creation of a GUI-based checkers game that allows pieces to be moved, jumped, and crowned, and requires sides to take turns. Required knowledge: Optionally threads.

www.manaraa.com

5.1.4.11 Turn display

The creation of a GUI-based checkers game that allows pieces to be moved, jumped, and crowned, requires turns, and displays current turns. Required knowledge: Layout managers, labels.

5.1.4.12 Required jumps

The creation of a GUI-based checkers game that allows pieces to be moved, jumped, and crowned, requires turns, displays current turns, and requires jumps whenever they are available. No new knowledge is needed.

5.1.4.13 Game completion notification

A fully-functional, GUI-based checkers game allowing all valid plays, requiring turns, and displaying the winner then the game is completed. Required knowledge: Game lost algorithm.

5.1.4.14 Double buffering

A fully-functional, GUI-based checkers game with double-buffered graphics for smooth screen redraws. Required knowledge: Double buffering.

5.1.4.15 Seizable board

A fully-functional, double-buffered, GUI-based checkers game that allows resizing. Required knowledge: Component listeners.

5.1.4.16 Networked game

A fully-functional, double-buffered, resize-able, networked, GUI-based checkers game. Required knowledge: exceptions, sockets, Java I/O.

5.1.4.17 Multi-threading

A fully-functional, double-buffered, resize-able, networked, GUI-based checkers game that uses threading to prevent freezing during network communication. Required knowledge: threads

المنارة للاستشارات
www.manaraa.com

5.1.4.18 Basic Chess

A fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game with all typical moves. Detecting checkmate and stalemate are not required yet. Required knowledge: loading images, chess moves.

5.1.4.19 Complete Chess

A fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game with special moves (en-passant, castling, and pawn promotion). Detecting checkmate and stalemate are not required yet.

5.1.4.20 Game end notification

A fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game that notifies on checkmate and stalemate. Required knowledge: Checkmate and stalemate rules.

5.2 Supporting Features

The core of the $\tau\acute{\epsilon}\chi\nu\eta$ approach is using large-scale, graphical projects to structure the learning environment. However, the Quest-Oriented Learning (QOL) model is more than just large, graphical projects, and include elements of encouraged research into outside materials for better results, opportunities for repeated practice and feedback, and peer learning. While some of these supporting elements may not be as important later on in the students' academic careers when they have already learned how to seek outside information, how to learn from and teach others, and how to practice and hone skills on their own time, it is still important for the environments in each classroom to provide encouragement for these healthy learning behaviors. In beginning classes, students need more direct instruction as they learn to take control of their individual quests.

5.2.1 Pair Design for Laboratory Work

This component encourages the student to take the role of teacher if his partner does not understand the problem. Unfortunately, some students actively avoid collaboration [45], possibly in an effort to be better in control of their results or out of fear of social environments. It is important to motivate these reluctant collaborators to begin to build their abilities to work with other computer scientists.

www.manaraa.com

5.2.1.1 Background

Pairs programming is widely recognized as a successful method of improving students' competence in computer science. Pairs programming has been shown to improve course retention [53], improve programming ability [29], improve test scores [54], accelerate the programming process [75], and increase student enjoyment [53] while lowering students' dependence on teaching staff [76].

Pair design [49] is a modification on pairs programming which additionally provides a place for students to practice design skills. Teaching students design concepts in first-year courses has been a goal of many educators, leading to classes solely in design [3], required design submissions [44], and having students code from a design [25]. However, while all these approaches expose students to design, they do not actively demonstrate to students how practicing design can actually improve their program accuracy and efficiency.

Consequently, by modifying pairs programming to depend on good program design, we have created a new method of collaboration called pair design. Pair design provides similar collaboration benefits as those afforded by pairs programming with 4 additional benefits: 1) students practice design before coding; 2) students see the benefits of doing a thorough job of design; 3) students must be able to understand the program enough to work alone; and 4) students learn to teach one another.

5.2.1.2 Support of QOL

The pairs design approach is used in the laboratory environment in order to support the Quest-Oriented learning model by allowing students to learn from and teach others, encouraging students to develop the social relations needed to achieve open problem discussion, and preparing students for the planning needed in large projects. As students work through the phases of semester-long projects, they become aware of the benefits of designing solutions in advance. Otherwise, the code becomes difficult to manage and upgrade. The $\tau\acute{\epsilon}\chi\nu\eta$ curriculum brings with it an emphasis on planning solutions before coding. Even if students are required to hand in designs for their solutions, it is difficult to teach them how the design process should truly work.

To answer this need, we use pair design to train students in the lab how to design before coding. In that the semester-long projects of $\tau\acute{\epsilon}\chi\nu\eta$ require good designs, the $\tau\acute{\epsilon}\chi\nu\eta$ approach is driving both lab approach and the class lecture approach. Pair Design provides the support needed for the class projects while also emphasizing the social aspects of learning that are recognized in the problem-based learning [21], constructivism [6], and intentional learning [47] approaches incorporated in the $\tau\acute{\epsilon}\chi\nu\eta$ model. The labs for the

المنارة للاستشارات
www.manaraa.com

$\tau\acute{\epsilon}\chi\nu\eta$ project utilize teamwork to teach students programming design and social skills, as well as reinforce their programming skills.

5.2.1.3 Description

The concept behind pair design is to adapt pairs programming to small tasks in order to improve the lab experience and reinforce other important computer science skills. There are 4 key components to the approach: 1) students are to pair up with someone of a similar level; 2) students work designing a solution to the project on paper before coding; 3) students code separately; 4) students are rewarded for the performance of their partner and the speed of their completion.

The first component of pair design is based on previous research with pairs programming. Katire et al. found that students perform better when paired with students they perceive to be at a similar level [38]. Students who consider themselves more advanced than their teammates are unlikely to accept input. Similarly, students who consider their teammates to be more advanced than themselves may be unwilling to contribute. Therefore, the best team performance is from students who perceive one another as equals.

The second component is the most important part of the approach. Each pair of students is asked to design a solution before beginning any coding. Students must work on paper to outline an approach that is best for solving the problem. This step is key in training students to think through problems and perform design testing before beginning code. As students work out the solution together, they learn to locate information from their notes and textbooks themselves, rather than relying only on the lecturer. The key to this phase for the instructor is choosing problems that can be designed in an hour, written in an hour, and yet still require planning.

The third component gives students a chance to work by themselves. While this is a departure from the pairs programming approach, this step provides a few benefits. One is that both students have the opportunity to code what they designed together. Therefore, both students have the opportunity to look at the code themselves and understand it. Moreover, if one student is taking the lead on the project, he must confirm that the other student understands the design well enough to work on his own. On a small scale, students are taking on the role of a teacher and practicing use of programming terminology.

The final component is to provide motivation to students to work with their partners and to do a good job of designing the solution. A few students may be prone to ignore their partners and work separately, and others may do a slip-shod job of designing and testing the design. Thus, the last component is to provide positive reinforcement for working together by making part of each student's score depend on the performance of

المنارة للاستشارات
www.manaraa.com

his partner as well as emphasize the importance of coding from a correct design. With this approach, students who refuse to work with their partners will not be able to achieve as high a score as students who do, and students who design poorly will take much longer to complete the assignment than those who have a good design to work from.

5.2.1.4 Implementation

We have utilized pair design in the laboratory environments for all four beginning computer science courses. The components of pair design were implemented in the following manner:

Pairing At the beginnings of the semesters, students were allowed to pair with whomever they chose. In classes in which the students did not know each other, they were assigned partners. Each week, students were permitted to choose different partners. Once students began to differentiate themselves by performance, students were assigned to pairs based on similar programming ability. With minor exceptions though, students had already paired themselves with those of a similar level. In the case of CS2, most students naturally chose the partners they had had from the previous semester.

Design The first half of the lab period (approximately 45 minutes), student pairs were asked to write down on paper a design for the problem at hand. Students were allowed to use textbooks and notes, but not the computer. This final restriction was to prevent students from coding and testing the program early. Students were encouraged to design in pseudocode, but some tended to write out the program on paper instead.

Coding At the half-way point of the lab, students were asked to separate from their partners and code the solutions on their own, using the paper design they had worked out with their partners. At this point, students could no longer consult with their partners, but instead had to rely on the accuracy of their design.

Reward for Performance At the end of the lab, the first pair to submit the program correctly was awarded an extra 5 points. This reward was to help motivate the pairs to plan well. Programs that are planned well do not require as much debugging. If the design is perfect, students can simply translate pseudo-code to code and work out any syntax errors. Therefore, this award seeks to motivate students to do well designing the solution in the first 45 minutes. By rewarding pairs, students who are simply fast at programming cannot win if they have not designed well enough to keep their partners on track.

المنارة للاستشارات
www.manaraa.com

A second incentive for designing well with partners was that the student's partner must correctly complete the assignment in order for the student to receive more than an A-. This point system instills in students the necessity of ensuring their partners' understanding.

5.2.1.5 Results

The second-year programming class was split into two labs: one with the previous approach of working individually with no required design and one with pair design. Both sections were given surveys about teamwork, design, and other programming behaviors. Statistically, the results showed no difference between the two sections. While it is disappointing that the students involved in pair design did not outperform the students in the original approach, they did not under-perform the students in the original approach either. We can conclude that without any detriment to performance, students were able to use half of the lab time to work through solutions with a teammate and half the time to code, while the other lab used the entire time to code the same lab. We conjecture that more difficult problems may be necessary to differentiate performance.

Benefits The intangible benefits of using pair design in CS1 and CS2 seem to be that the new computer science students quickly made friends and learned from each other at a point in which many first-year students feel isolated and lost. The environment of the CS2 class seemed more open and relaxed than usual, since the students were more accustomed to their peers in the class. Students were ready to discuss solutions without fear of being wrong. Gradually, students were learned what design processes work best for them and led them to complete the assignments more quickly and correctly. Pair design are provided them the opportunity to engage in trial-and-error learning of the design process.

Needed Improvements A couple of problems surfaced with using pair design with the first-year classes. While designing the solution was not too difficult for the second-year students, first-year students struggled to work through logic without simply coding the entire assignment on paper, thus eating into their design time. Similarly, first-year students were more likely to design a flawed solution and not identify the problems until coding time. Since the goal of pairs design is to provide practice and motivation for designing, this is not necessarily a flaw in the approach. However, in light of these difficulties, future first-year classes will include more training focused on how to design and test a solution separate from compiling and running the program. This will better equip the students to apply these skills in the pair design environment.

المنارة للاستشارات
www.manaraa.com

Conclusions Pairs design provides students with the opportunity to learn the benefits of design, design testing, and teamwork. The current implementation of the approach seems to indicate that the students learn equally well in second-year courses while first-year students learn the value of discussing and designing a solution before coding. While beginners struggle with design, they are learning the benefits of thinking problems through completely before typing code. Since many beginning programmers are accustomed to a technique of "changing things until it works," pair design alternatively exposes students to a more efficient and reliable programming method.

5.2.2 Additional Components

Semester-long graphics projects provide the core structure in the $\tau\acute{\epsilon}\chi\nu\eta$ curriculum. Additional components in line with the goals and foundation are also employed, especially for students in lower-level courses. Five additional reinforcing components used are ice breakers, online coding practice, encouragement of extra credit, reports, and open problem discussion.

Ice breakers are a method of introducing students to one another and establishing a comfortable learning environment. Examples include creative self introductions or group games. These activities encourage socialization, decreasing isolation. Also, students who do not make friends with others in the class may be too shy to ask questions in class. While ice breakers may not be as necessary for upper-level courses, they are strongly encouraged for any group of students in which the majority of students do not know one another, such as the first course in a sequence.

Online coding practice is provided to first-semester $\tau\acute{\epsilon}\chi\nu\eta$ students through the automatically-graded, online coding practice suite called CodeLab (*http://www.turingscraft.com/*). While repetitive coding practice may be achieved other ways, CodeLab has proven effective in many settings. The National Science Foundation funded the development of CodeLab out of the successful WebToTeach program [1]. CodeLab gives beginning students the repetitive practice drills needed at an early stage to learn language constructs at their own pace without any burden on the instructor to create or individually grade these drills. CodeLab has over 200 questions for students to answer, beginning with the declaration of variable types to the creation of recursive functions. Students must pay to subscribe to CodeLab, but CodeLab is free to the instructor.

www.manaraa.com

Extra credit encouragement may be provided by capping maximum grades for perfect assignments below 100%, requiring some added feature for additional points. The benefit of encouraging extra credit is the opportunities for students to express their creativity by devising useful additions to the program not covered in class. Also, students must take the initiative to discover how to add these features. To encourage the creation of attractive images, a teacher may offer extra credit for artistic results. Anecdotally, the most creative work done by students in $\tau\acute{\epsilon}\chi\nu\eta$ has been from open-ended extra credit.

Class reports let the student presenters discover where to find needed information and practice explaining these concepts to the rest of the class. Each students gets experience in minor research and teaching, while the class learns about new tools. For example, in the first-year courses using the unix OS, every student may give a 5-minute presentation on a different unix command. Students learn how to use *man* (or the internet) to find information on the command and write a presentation about it. In later courses, students may give short presentations on programming techniques, library functions, and related work at the instructor's choosing.

Open problem discussion means that students are encouraged to discuss the problems with each other as long as they never talk about code. Additionally, students who have completed their assignments are encouraged to help each other with minor syntactical compiler errors, in which a "minor" error is defined as an error caused by mistyping, such as misspelled function or variable names, missing or extra symbols, or unclosed comments. Errors that are caused by a misunderstanding of the solution, such as adding instead of multiplying, are not considered minor. Concerns about cheating have led to restrictive interaction rules, inhibiting healthy academic growth through contact with others. Restrictive rules have not prevented cheating; instead, first-year students in highly restrictive environments tend to feel isolated and frustrated, leading some to panic and flagrantly copy another student's file. Since some students will cheat regardless of the rules, it is not useful to penalize honest students. Instead, it is recommended that a clear policy of appropriate interaction be established and agreed upon. For example, in one first-semester course, students were required to have anyone who helped them debug minor syntactical errors sign a paper stating what help was given and when. At the bottom of this so-called "honesty sheet," the student signed that no help occurred outside the help listed and that he did not participate in discussions involving code or pseudocode, outside of help with solving minor compilation problems. The benefit of such sheets is that students are clear on what signifies cheating and must clearly indicate whether they were a part of it. Honor codes have been shown to reduce

المنارة للاستشارات
www.manaraa.com

cheating, allowing students the freedom of open discussion. Psychologist Lev Vygotsky asserts that learning happens not by itself but through social interaction [71]. Although interaction with the professor is definitely social, more interaction prolongs the learning process.

All of these reinforcements are in line with the goals of Quest-Oriented Learning and bolster the foundation of the curriculum. Ice breakers, open problem discussion, and pair design encourage socialization. Reports, open problem discussion, encouraged extra credit, and pair design put students in the roles of researchers and teachers, in line with intentional learning. Online coding practice gives students repetitive practice without burdening the instructor, and open problem discussion also lessens the load on the teacher to help with minor debugging problems.

5.3 Introductory Language Selection

It is increasingly common for Computer Science educators to introduce students to programming using object-oriented languages, especially the popular Java programming language. Because of this current trend, the technique of beginning students with an imperative language (C) is not typical and should be explained.

First, the author is certainly not unfavorably biased against Java or object-oriented programming per se. Not only was Java her first programming language with object concepts postponed ("Objects Late"), but the author's Master's thesis was a method of extracting UML class diagrams from C++ source code [48]. The decision to begin with an imperative language was instead based upon which approach was more appropriate for the education of good computer scientists. Under this curriculum, objects are introduced to students in the second semester, and Java is currently the language of choice for the design class in the fourth semester.

5.3.1 Java Versus C

The debate over the introductory programming language may be broken down into two categories: debate over specific language choice and debate over language type (imperative versus object-orientation). The specific language choice that has been debated since the inception of this curriculum is C versus Java. Some criticism is aimed almost exclusively at perceived problems with starting with C, while other criticism emphasizes a perceived superiority of Java over C.

المنارة للاستشارات
www.manaraa.com

5.3.1.1 Addressing C "Traps"

At the first suggestion of beginning with C instead of Java, one colleague recommended Andrew Koenig's technical report on C pitfalls [40]. His report and following book by the same name bring to light a great number of tricky points with the C programming language, and it is a great resource for C users. The paper does not however make the case for Java over C, as sixteen of the addressed pitfalls also occur in Java (due to its similarities to C), including confusion about bitwise and logic operators, string and character notation, operator precedence, incorrect semicolon placement, switch statement structure, dangling else clauses, expression evaluation order, zero-based arrays, shallow vs. deep copies, integer overflow, shift operator issues, division truncation, function invocation parenthesis requirements, and naming restrictions. An additional shared issue, confusion about "=" and "==", is addressed in Java by the existence of a boolean type, but can be addressed in C by requiring students to put the constants first in conditional comparison statements. e.g. `if(5 == x) {}`. Two pitfalls relate to macro use (which should be avoided, especially in the first semester), three relate to misconceptions about library functions that can be cleared up by reading library documentation, three have been addressed in newer compiler versions, two would be caught by the compiler, and three are tremendously rare and would be issues only for experts in the C language who exploit advanced features and notations in C. Three traps relate to portability, which is a problem in any language except Java, and Java of course possesses sixteen other pitfalls.

The three remaining pitfalls are the existence of both signed and unsigned character types, the differences between pointers and arrays, and assumptions by `scanf` about its parameters' types. Java's lack of an unsigned character type complicates image processing, so the availability of a choice in C is beneficial. Pointers and arrays are definitely tricky aspects of the C language, and while they can be postponed to later in the first semester, students will eventually need to learn C, and placing its introduction into one second-year course does not afford students enough time to become comfortable with the language [63]. Similarly, use of `scanf` can be postponed through use of other I/O functions until students are ready to learn how `scanf` truly handles parameters. An old proverb says, "knowledge is easy to him who understands"(Prov. 14:6b), and once students understand how `scanf` works, they can easily avoid mistakes.

5.3.1.2 Comparing Java to C

After addressing general concerns about C, the question remains whether Java or C is better suited to introductory courses. Java is described in *The Java Language Environment* by James Gosling and Henry

المنارة للاستشارات
www.manaraa.com

McGilton (May 1996) as a simple, interpreted, portable, robust, high-performance, multithreading, adaptable, secure programming language platform. On the other hand, C is described in *The C Programming Language* [39] as a "general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators." As might be guessed by their descriptions, the C programming language is much smaller than the Java programming language, and the provided C library is much smaller than the Java API. Thus, the first argument for C over Java is that C is a smaller language, and it is actually simpler to fully understand. Students beginning in Java may be overwhelmed by the extensive (and still evolving) Java API, whereas the C programming language's API is much more limited and more stationary. Java, with its many advanced features, is overkill for beginning students who will not benefit from a large portion of them. Once students are comfortable with programming and using provided libraries, the Java API becomes a wonderful, extensive resource.

Additionally, since Java draws from C++ in its design (which in turn draws from C), program control (counted loops, conditional loops, and branches) in C and Java are virtually identical. Most concepts needed in C are also needed in Java, and therefore C can be used as a stepping stone to Java once the shared concepts are mastered. One cannot learn to program in an object-oriented manner without also learning logical program control, and therefore C provides the foundations needed for later OO development.

Extra functionality provided in C that is not in Java includes memory allocation control (i.e. choice of dynamic or non-dynamic memory), pointers, and more primitive data types (with the exception of a separate boolean type). All three features lead to discussions of the computer memory model and promote student understanding of what occurs inside the machine when they write code. With its strong ties to assembly language, "C's design follows an underlying logic" [63], providing a clearer view of the machine than higher languages do. The direct correlation of C structure to the computer memory model allows students to observe any flaws in their naive mental models of computing systems, accommodate the observed behaviors, and construct more accurate models.

For the purposes of this curriculum, Java's lack of an unsigned character data type makes it less suitable than C for modifying binary image data. This problem is exacerbated by the lack of flexibility in the provided Java I/O libraries. The simple portable pixmap (PPM) format image files used in the first year of classes have ASCII headers followed by binary image data. Java's I/O library makes reading a file with two different parts difficult. (Note that no image reading libraries are used in C or Java in the introductory courses.) Java provides a binary data reader, a highly versatile ASCII scanner, and a reader that allows data to be "pushed back" into the stream, but each is a separate reader that cannot be used in tandem with any other.

المنارة للاستشارات
www.manaraa.com

PPM image files can be read with some difficulty using Java's data reader, but the resulting data are signed bytes. Any modifications made to the data must be done after converting each byte into an unsigned format (using addition and modular arithmetic) to place it to the range [0, 255].

While C has some useful features Java does not have, Java has many useful features C does not have, such as garbage collection, built-in string handling, and simpler compilation/interpretation. These features are very helpful for experienced programmers, but obscure the mechanisms from beginners. Just as mathematics students learn to do arithmetic before using calculators, CS students should learn about memory handling, string representation, and the compiling process before using tools that perform the work for them. Similar to the benefits of C's extra features (pointers, memory allocation control, etc.), C's *lack* of garbage collection, built-in string handling, and one-step compilation lead to better understanding of underlying mechanisms. By needing to free allocated memory and being forced to understand the structure of a string, students observe the impact of their programs on the machines. Similarly, seeing the phases of compilation through use of preprocessor directives and header files, and the separation of compiling and linking, students understand what a compiler is doing, preparing them for later classes in compilers and optimization.

The understand-before-tools approach that drives teaching mental arithmetic before calculator use applies directly to debugging skills. While Java's clear compiler and runtime error messages aid rapid error location and correction for skilled programmers, these supporting features can be detrimental to beginners. Students often become dependent on an environment's ability to almost always pinpoint the offending line and fall into a "change something in that line" routine, inhibiting the development of debugging skills, and frustrating students when the line indicated is not actually the problem. Students need to learn from the beginning when the programs are small how to find errors through code reading and analysis so that they have the debugging skills necessary when program sizes increase. Since computers are not yet better able than a good programmer, students need to develop the skills to find errors themselves in order to benefit later from the hints provided by error messages.

One final argument draws from the belief that study of older and simpler tools naturally segues into study of newer and more complex tools. The progression from C to Java is better explained in historical sequence, revealing the logical steps leading from one to the other. For example, by starting without garbage collection, phasing into garbage collection is straightforward. Also, since classes and objects were created to improve design in imperative programming, starting without them and building the need for them more naturally follows the historical evolution. Starting a path with C and leading to Java reveals to beginning students the legacy and profound impact of the C programming language on Computer Science.

المنارة للاستشارات
www.manaraa.com

5.3.2 Imperative versus Object-Oriented

Stepping away from the C and Java discussion, imperative languages are more appropriate introductory languages than object-oriented languages for a number of reasons. First, imperative languages begin students with a concrete, logical foundation before exploring abstract ideas. As pointed out in previous work [51], first-year college students are better suited for concrete knowledge [59], than for the abstract principles involved in object-oriented programming. Abstraction is "forgetting" details of implementation and is a wonderful tool for students, once they understand the underlying details, but ". . . how is it possible to forget detail that you never knew or even imagined?" [6] Instead, students learn (constructively) what is shared by both language types: a ". . . programming foundation (loops, selections, algorithms, and procedures) which is best learned in a structured programming environment, where good algorithms can be addressed through concrete, yet complex in nature, actions or processes" [32].

Second, depending on the educational environment, some students taking introductory programming are there merely to learn the basics of programming to allow them to write single-use, utility programs, such as scientific data manipulation and analysis. Object-oriented programming adds a great deal of overhead to the first course, resulting in a ". . . change in the content and structure of first-year programming courses. The requirement to teach object-oriented design principles tends to relegate the essential concepts such as selection and iteration into a secondary position in the initial teaching framework" [64]. This added complexity has sometimes led to a three-course introductory sequence [67] [57], instead of the traditional, two-course sequence. Of course, a larger number of introductory courses is not a concern, but the courses are then called by faculty and publishers as CS1, CS1.5, and CS2 [57], implying that students are not gaining more knowledge but instead taking two semesters of content in three. Objects-early slows learning of fundamentals and opens students to a variety of misconceptions, including confusing objects with single variable wrappers, confusing objects with aggregate types, assuming limited method abilities, believing an object is the same as a class, and confusing references with the objects [30].

The argument that one must start with objects to understand objects does not stand to reason, in that object orientation did not always exist, and the first group of people to understand objects and explain them to future generations obviously did not learn programming in an objects-early approach. OO is a wonderful paradigm for code organization and functionality that does not stand separate from imperative programming, but instead is dependent upon it, and actually grew out of it. Furthermore, objective searches ". . . for empirical work relating to the difficulty of making the shift from imperative to object-oriented programming"

المنارة للاستشارات
www.manaraa.com

reveal only anecdotal evidence from two papers [41] [19], ". . . but neither paper reported the results of a systematic study" [46].

Despite the claimed benefits of teaching objects early, debates about using the methodology have continued to rage. Debates on a SIGSCE mailing list were published in 2004 to shed light on some of the arguments [8]. Contributer Stuart Reges complained

> I think that the reason we're having this heated debate is that it hasn't turned out that a broad range of teachers have been able to easily teach [objects early] effectively. We hear things like: Professor A has succeeded because he uses a custom IDE designed specifically for teaching objects; Professor B has succeeded by developing a framework of graphics classes; Professor C has succeeded by developing tons of supporting code for each assignment. These are bad signs, not good signs. Where is the list of professors who have succeeded because the material is straightforward to teach? And if the material isn't straightforward for a lifelong computer scientist to teach, then can it really be all that fundamental? [8]

Bruce summarizes the debate with the following observation: "The one thing that there was near universal agreement on during the discussion is that it is a challenge to teach objects early." She continues by saying that those who have been successful use "pedagogical IDEs, special libraries providing useful classes, or microworlds" [8]. In contrast, imperative programming can be taught with no special libraries or interfaces, equipping students in one semester to write programs without props.

As a final point, students should be introduced to programming in a non-OO manner because students never genuinely begin programming in an OO fashion, as evidenced by the community's shift to the term "objects early" over "objects first." The instructor may describe OO and perhaps even design classes and objects on paper, but the first programs in objects-early textbooks are never truly OO. Either the coding is done in static methods or the classes demonstrate none of the attributes associated with objects, such as maintained state (instance variables), encapsulation, reusability, etc. The only true difference between beginning programming with objects or without objects is how soon the imperative starter code is replaced with OO code. If it is impossible (or tremendously difficult) to introduce programming in an OO fashion, then objects are clearly a complex topic best left for later classes after students have learned programming basics.

المنارة للاستشارات
www.manaraa.com

Chapter 6

Adaptations to Other Environments

6.1 Adaptation to Small Colleges

During the 2006-2007 year, Professor John Hunt adapted τέχνη to fit the needs of Covenant College, a small, faith-based, liberal arts school where he is employed. His experiences [33] demonstrate that existing τέχνη courses need to be and can be re-tooled to meet the needs of different environments. As large institutions discover ways to improve curricula, small colleges with fewer resources can benefit from adapting proven approaches to fit their own needs.

Large universities with strong science and engineering programs, such as the type at which τέχνη was created, structure their computer science courses with the natural assumption that most students will be taking multiple courses in computer science and likely will be entering a field requiring a great deal of programming knowledge. In contrast, at Covenant College the vast majority of first semester students intend to take only one computer science course to fulfill other field requirements. In fact, the 2006-2007 year saw only one computer science major in a class of the thirty two students. The goals of the other students, outside of fulfilling major requirements, include the abilities to create small, single-use programs that solve specific problems, such as data from a physics experiment. While these students may not be looking for careers in computer science, they still benefit from the challenge and motivation provided by Quest-Oriented Learning.

Certain aspects of the τέχνη implementation at Clemson may be overkill for students intending to enroll in only one semester of programming. For example, use of the Unix environment from the beginning for computer science students equips them for many career opportunities as well as providing them with utilities not afforded by other commercial operating systems.

المنارة للاستشارات
www.manaraa.com

Although controversial, Professor Hunt suggests that while the C language is a great way to start for computer science students, it affords more deep understanding than needed by non-majors seeking only basic programming understanding. While we can all agree that everyone benefits from a deeper knowledge of any field, Hunt's concern is that the overhead of the deeper knowledge to students who have no future in computer science may be unnecessary.

In the case of Covenant College, while Professor Hunt had at his disposal the Unix environment with its C compiler, the decision was made to select a language more suited to the students' expectations that could be written and compiled on an operating system already familiar to them. (Although Windows and Mac OS/X compilers are available, C is not standardized across these compilers.) Outside of requiring a language with the ability to read and write binary files, support for arrays, and some ability to manipulate bytes, the first-year $\tau\acute{\epsilon}\chi\nu\eta$ courses do not mandate any particular programming language.

The programming language chosen for environments such as Covenant College needs to be one that is useful and freely available outside of the educational environment, allowing students to apply their newly-acquired skills outside of the classroom. Thus, teaching languages (Karel the Robot, Alice, etc.) are not appropriate, and neither are languages that require licenses (e.g. C# and Visual Basic). Finally, if faculty members of other departments assume students have knowledge of languages with conventional algorithmic outlook [20] (as is the case at Covenant College), languages like Scheme and Haskell are not appropriate. The desire to use a freely-available, well-known, and highly standardized language at Covenant College led to the Java programming language. Java is well known for its clear error messages both at compile time and particularly at runtime. It is also suitable for additional computer science courses for students who do enroll in other CS courses.

The choice to teach the course in Java brings its own problems, not the least of which is Java's object orientation. Non-majors learning programming certainly do not need the added complexity of the object-oriented Paradigm. Additionally, while Java meets all the requirements for the $\tau\acute{\epsilon}\chi\nu\eta$ courses, including the ability to read and write binary files, Java's I/O is complicated and requires exception handling.

To address these complexity issues, one may use the objects-late approach as Hunt did, requiring all methods to be static. In the second course (the raytracer), students may learn OO, as is appropriate for the raytracer. Discussion of exception handling may be postponed by use of the "throws" clause any time a method that throws a checked exception is invoked.

Choosing the programming environment to use for Java requires some deliberation. Using learning environments, such as BlueJ and DrJava, for a single semester tends to lead students to think the interface

المنارة للاستشارات
www.manaraa.com

and the language are one and the same, and students may not always have access to these environments. At Covenant College, Hunt instead instructed students to use basic text-editing programs, such as Notepad and Wordpad. Compilation may be done with the Sun™Java compiler from the command prompt. With such a minimal tool set, students may use any operating system with which they are comfortable.

Thus, the *τέχνη* approach may be applied to other operating systems and other programming languages that allow generation of PPM format image files. Of course, the PPM image format is not as common as some other more complicated image formats, and while Unix has built-in image viewing applications that support PPM format, other operating systems do not. Fortunately, the internet affords many free, GUI image viewing tools that do support the PPM format. Hunt's class settled on XnView, an image viewer available for Windows, MacOS X, Linux x86, Linux ppc, FreeBSD x86, OpenBSD x86, NetBSD x86, Solaris sparc, Solaris x86, Irix mips, HP-UX, and AIX. Other viewers are freely available, and there will likely continue to be freeware programs supporting this format and conversion between it and more commonly-used formats.

Computer science students, both majors and non-majors, may benefit from the *τέχνη* curriculum. The curriculum affords a motivating and challenging learning experience that draws away from the typical basic programs that tend to bore this generation of students who, regardless of institution, are more visual and desire creative expression. With their lifetimes of exposure to computers, basic computer functioning (or even typical GUI's) cease to impress students. Whether they seek a lifetime of computing or a passing grade in a single course, students need a curriculum that sparks their interests.

6.2 Adaptation to an Upper-Level Course

Computer graphics provides a natural platform for teaching a variety of general computer science courses due to the rich variety and complexity of the problems encountered in computer graphics and the educational advantages of visual learning through the generation and evaluation of graphical images. Additionally, the real-world problems in the computer graphics field tend to capture the attention of students living in an ever increasingly visual culture.

In a standard course on programming, assigning large, graphics-based projects, as opposed to traditional "toy" projects, can be accomplished in a fairly straightforward manner, while simultaneously engaging the students at a much higher level. For a network programming course as discussed here, we propose that students will find projects such as distributed rendering (as used in computer-animated feature film production) and interactive networked-based graphical games more engaging than traditional projects, such as network

المنارة للاستشارات
www.manaraa.com

monitoring for performance evaluation [14].

6.2.1 The Course: CPSC 360

Students enrolled in CPSC 360–Networked and Distributed Computing, a third-year networking course at Clemson University–are required to know C, but may not have any background in networking; therefore, the class combines material on networking concepts with experience in network programming. Topics covered in the course include network types and characteristics, service paradigms, the OSI network model, DNS, sockets, network formats, and various protocols (IP, UDP, and TCP). Programming topics include sockets, Unix processes, and object-oriented network programming with Java and C++.

Programming assignments in the course traditionally build toward the final project, a performance management client-server application. Students use this application to assess network performance with two types of tests: an echo test and a sustained throughput test. For the echo test, the client computes the round trip time (RTT) of a message traveling between client and server to measure network performance. The sustained throughput test between client and server tracks the speed of data arrival and computes overall average throughput. Both tests are performed for UDP and TCP.

6.2.2 Supporting Resources

Many graphics applications, such as raytracing, are ideally suited for parallelization. One way to perform parallel rendering is to employ a network of workstations acting as a single machine. This approach, termed "distributed" or "cluster" computing, is conceptually similar to multiprocessing, but here each processing element consists of an independent machine connected to a LAN, usually much slower than a multiprocessor interconnect (backplane) network. While this network can be of any type (e.g., Ethernet, ATM) or any topology, the computers connected to it must support some type of distributed programming environment to help the machines work together.

With such a system, we can introduce graphics projects, such as distributed raytracing or real-time rendering of complicated surfaces, as a means of learning network programming. The next section outlines a proposal for the using these resources in the course.

المنارة للاستشارات
www.manaraa.com

6.2.3 New Approach

The re-designed course under $\tau\acute{\epsilon}\chi\nu\eta$ will cover the same content and require the same type of programming, but with a Quest-Oriented approach. In this way, we provide students the opportunity to learn about networking through problems relevant to professional practice and provide the motivation to excel in the educational environment.

The project used to structure the learning for this networking course could involve animation or manipulation of a large-scale, complicated surfaces (such as the rendered image of the Hunley submarine in Figure 6.1), implementation of an interactive game played across a network, or the development of a new method of subdividing production rendering across a large cluster. The instructor could vary the approach in different semesters. Depending on the amount of work required, students could work in pairs to implement the assignment. Once a project has been chosen for the course, the instructor must break it down into phases.

Figure 6.1: Fully rendered image of the Civil War-era Hunley Submarine

Choosing the development of a new method of subdividing production raytracing across a large cluster as the example, we could identify the first phase of the project as rendering a single image on multiple, networked machines, given the code to render on a single machine. This network adaptation would expose students to the advantages of parallelization, as well as the overhead incurred in using the network. Classroom instruction at this stage should cover networking basics, sockets, programming models, and sample code for distributing rendering tasks and compositing the results into a final image.

Building upon the first phase, the second phase could test differing network configurations, protocols, and service paradigms to determine what combinations work best for various problems. As students

المنارة للاستشارات

www.manaraa.com

work to write and debug these programs, they will rely on visual feedback provided by the resulting composited image to determine where problems lie. Methods for load balancing the rendering tasks across multiple machines can also be addressed at this stage. A rich set of load-balancing possibilities exist, including non-uniform tiling in image space, decomposition in object space, as well as variable task assignment between CPUs and GPUs. If students have previously taken a raytracing course, they can research and apply a wide variety of distributed computing techniques to their previous projects. Additional information on network protocols and programming techniques could be explored at this point.

Finally, after having experimented with different protocols and configurations to improve run-time, students would be ready to develop their own models of subdivision and configuration. This phase does not need to be completely open, but instead the instructor could recommend papers on new approaches or recommend ones not yet explored. Having some amount of freedom in the implementation may lead to a bit of competition between teams to have their images render faster, pushing students to seek solutions outside of classroom material.

The experiences at Covenant College and the example networking course demonstrate that $\tau\acute{\epsilon}\chi\nu\eta$ is not restricted to introductory courses at large universities, but instead can motivate, support, and broaden students in many, if not all, computer science courses, regardless of level or learning environment.

المنارة للاستشارات
www.manaraa.com

Chapter 7

Results and Evaluation

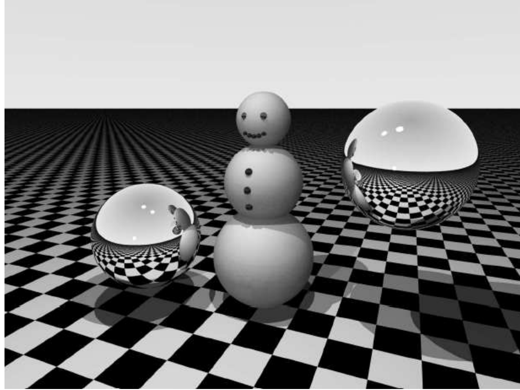
7.1 The Original, Second-Year Raytracing Course (215)

This second-year course, which explored programming methodology through the introduction of the C language via the raytracing project, generated encouraging results [15]. Based on the work performed and student evaluations, we concluded that the projects were more engaging to the students than previous approaches.

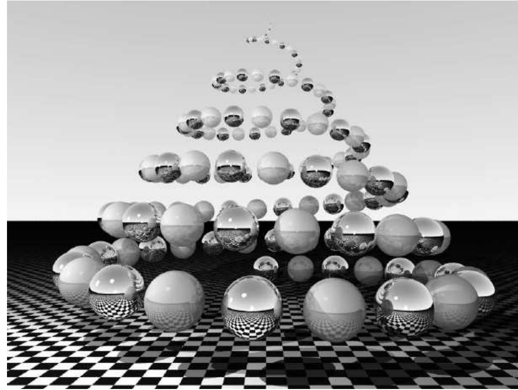
7.1.1 Student Images

Throughout all phases of the raytracing project, students were encouraged to be creative in scene design. In spite of the limited tools available to them, the students showed an impressive creativity. The images in 7.1 (a) and (b) represent work from the first phase of the ray tracer. Considering that the only geometries known to the students at this stage were the sphere and the infinite plane, we find these images show an impressive capability, which can probably be attributed to the students' heightened level of interest. The remaining images show mastery of additional features, such as reflection and anti-aliasing, as well as optional features, such as quadrics and textures. Figure 7.2 represents work from the same course taught two years later by a different instructor. The image 7.2 (a) is an Anaglyph, an image that appears 3D with two-color (red-blue) glasses. The sky in image 7.2 (f) was generated using a noise algorithm the students downloaded. The creativity and artistic components present in both sets of images, which come from classes with different instructors, demonstrate that the creativity expressed by the students was not restricted to one

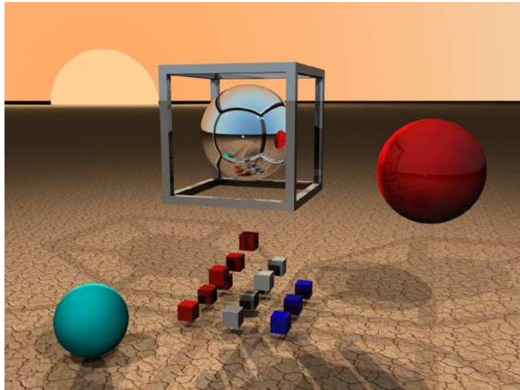
المنارة للاستشارات
www.manaraa.com



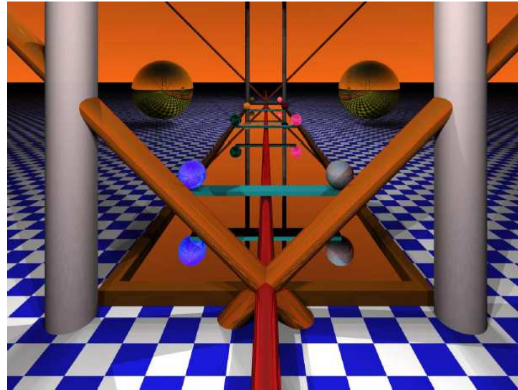
(a) By student S. Duckworth



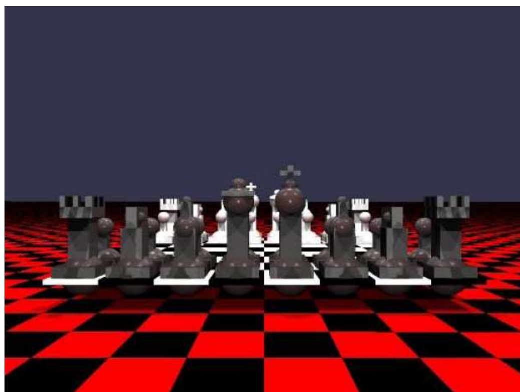
(b) By student T. Nguyen



(c) By student S. Duckworth



(d) By student T. Nguyen



(e) By student J. Holcombe

(f) By student S. Haroz

Figure 7.1: 2002 CPSC 215 Example Student Renderings.

المنارة للاستشارات
www.manaraa.com

particular instructor or particular group of students.

7.1.2 Student Evaluations

As evidenced by anonymous semester-end evaluations, students responded positively to learning C/C++ through graphics. Many students felt that the semester-long project was educational and interesting to implement. They especially seemed to appreciate the visual feedback from their projects, both for aesthetic and problem determination purposes. Corroborating evidence is supplied by the near absence of student decisions to drop the course, which was unusual for these classes. Many students brought their laptops (Clemson requirement) to class to discuss (or show off) the previous night's rendering successes and failures. The following are sample excerpts from anonymous student evaluations of the course [15]:

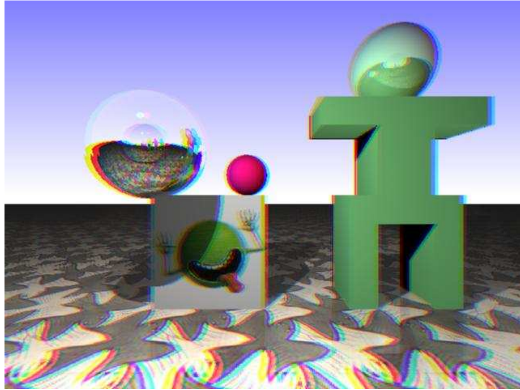
- The raytracer project was good because it gave visual feedback of your accomplishments and impressive results. I liked that we continued with several versions of the project leading to a large and useful program in the end.
- The raytracing project was great. It provided practical usage to learning C rather than just making a useless program that 'implements a linked list or binary tree.'
- The raytracer also gave me a much stronger knowledge of C than [other courses] did with Java.
- It is the first class where I wrote a program that I will not throw away at the end of the semester.
- The class wasn't just like some ordinary class. We got to do something fun and different.
- Making a raytracer is so much cooler than making a card game.

Other areas of comment, but on the negative side, involved the amount of work required toward the end of the semester, which was addressed by moving the project from a three-hour course to a four-hour course and by structuring the previous class to cover C and image processing in order to better prepare students for raytracing.

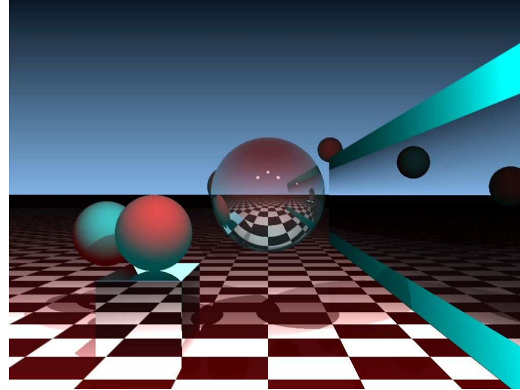
7.1.3 Survey Results

During the 2003-2004 academic year, 73 students from three different sections (taught by three different instructors) of the second-year raytracing course were surveyed to collect their impressions of the course. One section of the course had beginning Digital Production Arts (graduate, MFA program) students,

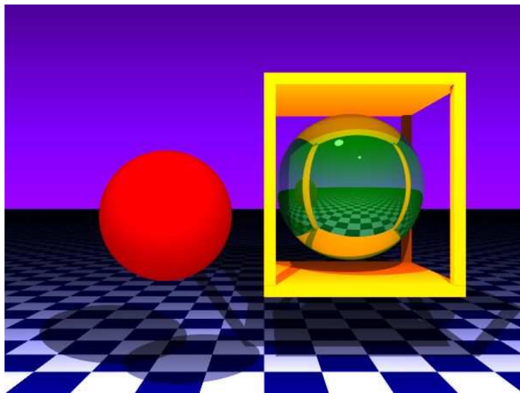
المنارة للاستشارات
www.manaraa.com



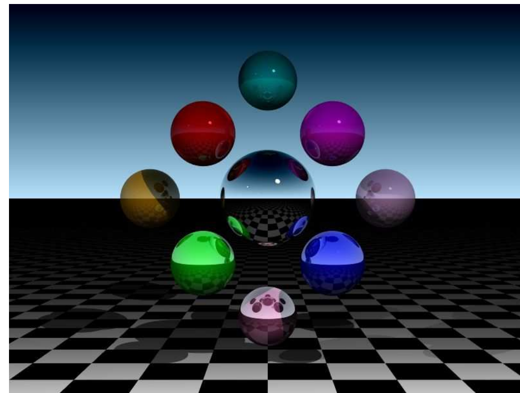
(a) 3D image by C. Guirl



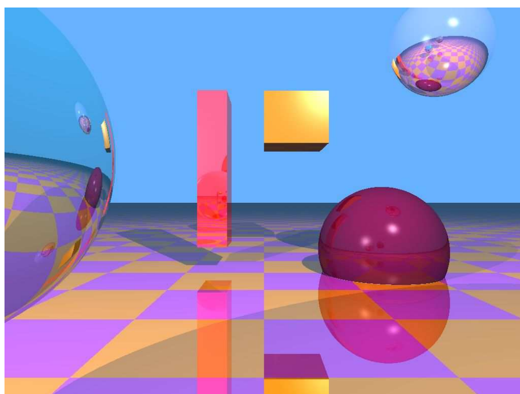
(b) By student D. Duvall



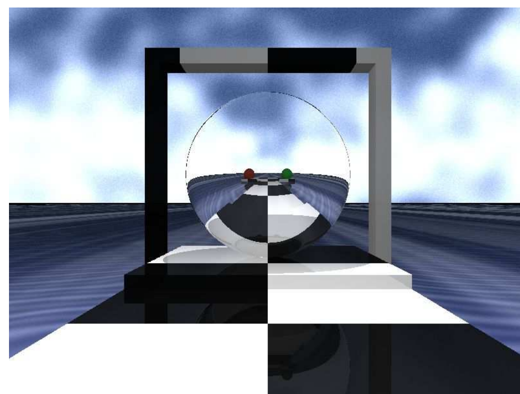
(c) By student J. Donboch



(d) By student J. Fisher



(e) By student K. Johnson



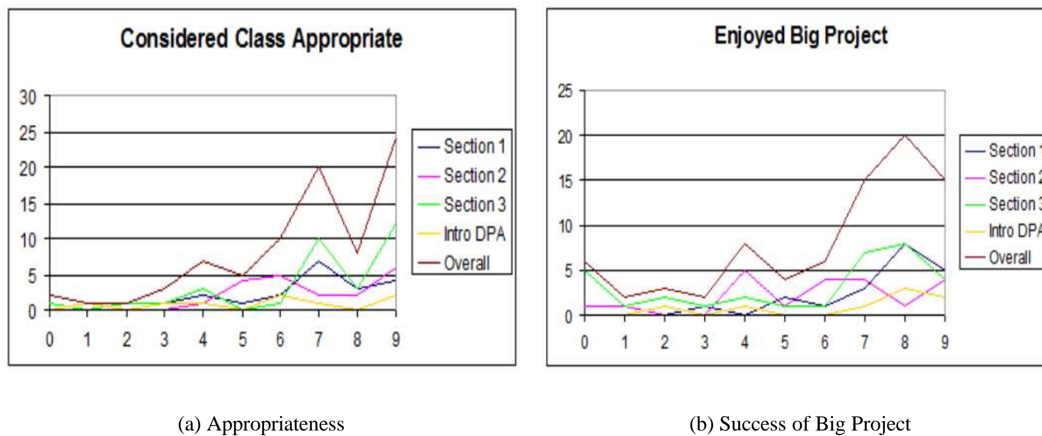
(f) By student R. Coleman

Figure 7.2: 2004 CPSC 215 Example Student Renderings.

المنارة للاستشارات
www.manaraa.com

and 8 of those students took the surveys as well. The survey questioned students on the relevance of the course, its impact on their interest in graphics, and the improvement of their C and Unix skills.

7.1.3.1 Course Relevance



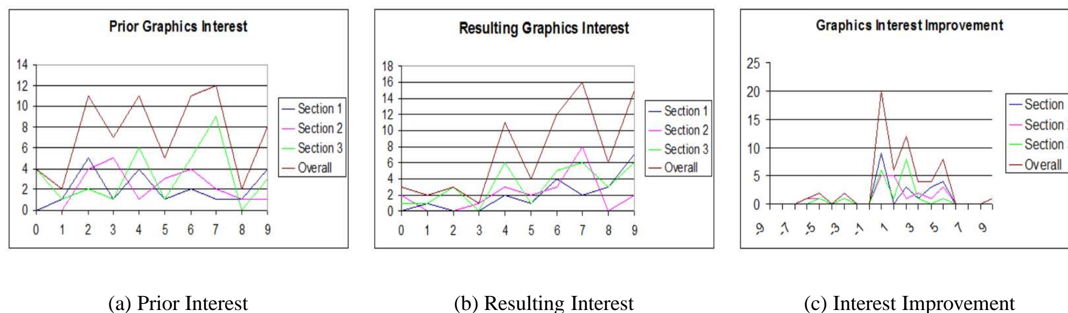
(a) Appropriateness

(b) Success of Big Project

Figure 7.3: 2003-2004 Pilot 215 Course Relevance

The charts (7.3) indicate that while not all students loved the approach, the majority of students believed the course appropriate and relevant. On a scale of 0-9, the students gave the appropriateness of the course a 6.73, with a median of 7, and a standard deviation of 2.23. On a scale of 0-9, the student enjoyment of doing a big project was 6.20, with a median of 7, and a standard deviation of 2.73.

7.1.3.2 Graphics Interest



(a) Prior Interest

(b) Resulting Interest

(c) Interest Improvement

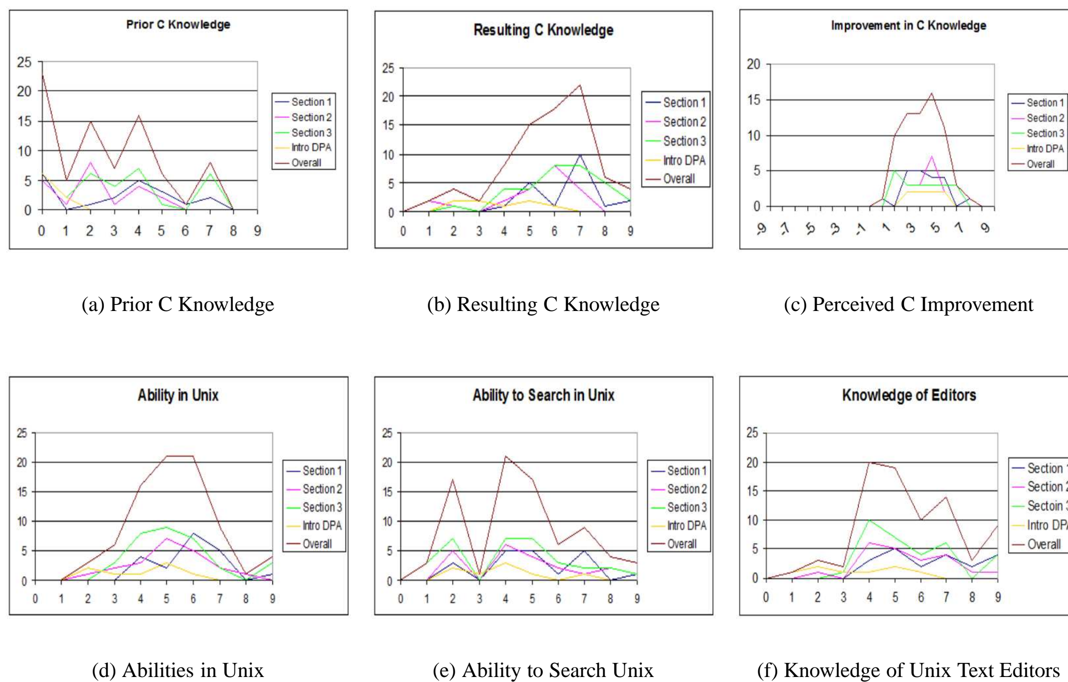
Figure 7.4: 2003-2004 Pilot 215 Graphics Interest

A concern that non-DPA students might dislike the classes if they are not interested in graphics was

www.manaraa.com

addressed by a question about prior and post graphics interest. The graphics interest beforehand was widely spread (Figure 7.4(a)) with an average value of 4.82 on scale of 0-9, with a median of 5 and a standard deviation of 2.56. Afterward, students placed interest in graphics at 6.05 with a median of 7 and standard deviation of 2.47 (Figure 7.4(b)). The graphics interest improvement in students averaged a statistically significant 1.16 points ($P < .0025$) with a median of 1 and a standard deviation of 2.66 (Figure 7.4(c)). In conclusion, students were not turned off by a class heavy in graphical content but developed new interest in the field.

7.1.3.3 C and Unix Skills



(a) Prior C Knowledge (b) Resulting C Knowledge (c) Perceived C Improvement

(d) Abilities in Unix (e) Ability to Search Unix (f) Knowledge of Unix Text Editors

Figure 7.5: 2003-2004 Pilot 215 Perceived Skill Development

Since the course targeted C and Unix, students were surveyed about their perceived skill developments therein. On the scale of 0-9, students placed their prior knowledge of the C language on average at 2.62 with a median of 2 and a standard deviation of 2.26 (Figure 7.5 (a)). After the course, students placed their C knowledge at a mean of 5.79 with a median of 6 and a standard deviation of 1.79 (Figure 7.5 (b)). The average perceived gain in C knowledge was a statistically significant 2.7 points ($P < .0005$) with a median of 3 and a standard deviation of 1.55 (Figure 7.5 (c)).

المنارة للاستشارات
www.manaraa.com

On average, students placed their overall Unix abilities at 5.26 points with a median of 5 and a standard deviation of 1.55 (Figure 7.5 (d)). Students considered their abilities to search for files in Unix to be on average 4.53 points (median 4, standard deviation 2.04, Figure 7.5 (a)), and their average abilities with Unix text editors at 5.57 (median 5, standard deviation 1.88, Figure 7.5 (f)). Students felt that they knew C better than Unix by an average of 61.4% C knowledge to 48.6% Unix knowledge (median of 60%, standard deviation of 1.66).

Overall, the surveys indicate that the class was successful in improving students' knowledge of C and Unix, as well as their interest in graphics, and the majority of students enjoyed the big, semester-long project. Having thus maintained the educational level of the course while adding the benefits of student and instructor enthusiasm, the pilot course was considered a success, leading to the installation of the $\tau\acute{\epsilon}\chi\nu\eta$ approach in the first four courses of the Computer Science major.

7.2 Computer Science I

In Computer Science I, the students incrementally work to create a color transfer program [50]. Using an algorithm due to Reinhard, Ashikhmin, Gooch, and Shirley [60], students read in two images, apply the color scheme from one image to the other, and output the resulting image. No libraries other than the built-in C library were used. The project requires knowledge of array indexing, pointers, dynamic memory allocation, and structures, making it well suited to first-semester students [51].

7.2.1 Phase 1 at Clemson University

In the fall of 2005, less than three weeks into the introductory computer science course, students were asked to turn in a C program that generated a PPM image. Although the images could be solid-color images (7.6 (c)), students were encouraged to create an interesting pattern for extra credit. From a class of 36 students, 50% turned in extra credit images with patterns or shapes for the first assignment. Surprisingly, none of the students asked for help with the extra credit versions. See Figures 7.6 and 7.7. Some images have intricate patterns that have been enlarged in a box in the upper left-hand corner of the images.

7.2.2 Phase 1 at Covenant College

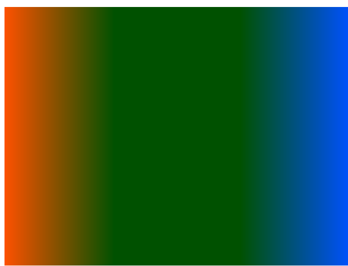
In the Fall of 2006, Covenant College used the $\tau\acute{\epsilon}\chi\nu\eta$ approach and saw similar results. (More information about Covenant College's experiences may be found in Chapter 6). Students came up similar but

المنارة للاستشارات
www.manaraa.com

(a) By student J. Leyh

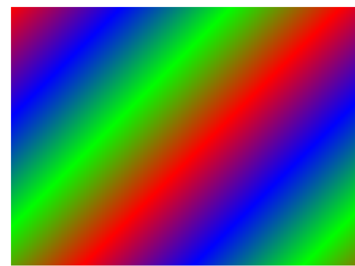
(b) By student M. King

(c) By student B. Holder

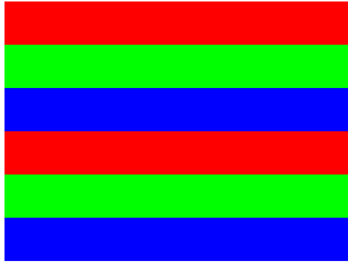


(d) By student M. Rardon

(e) By student V. Tan



(f) By student C. Daugherty



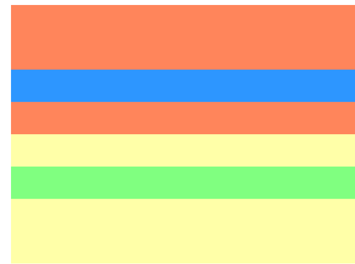
(g) By student B. White

(h) By student B. Schneider

(i) By student J. Canter

(j) By student J. Griswold

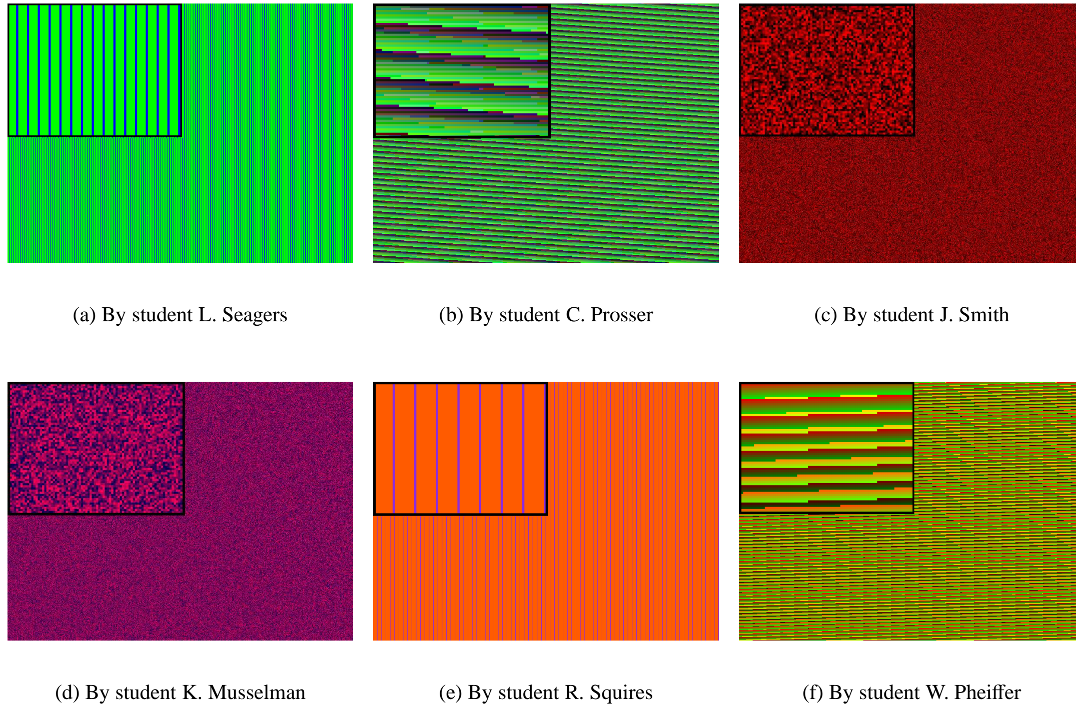
(k) By student K. Abbot



(l) By student T. Williams

Figure 7.6: 2005 CPSC 101 Phase 1

المنارة للاستشارات
www.manaraa.com



(a) By student L. Seagers (b) By student C. Prosser (c) By student J. Smith

(d) By student K. Musselman (e) By student R. Squires (f) By student W. Pheiffer

Figure 7.7: 2005 CPSC 101 Phase 1: Images with Enlarged Detail

different creative patterns, as shown in Figure 7.8. Again, images with intricate detail have enlarged views in the boxes in the upper left-hand corners.

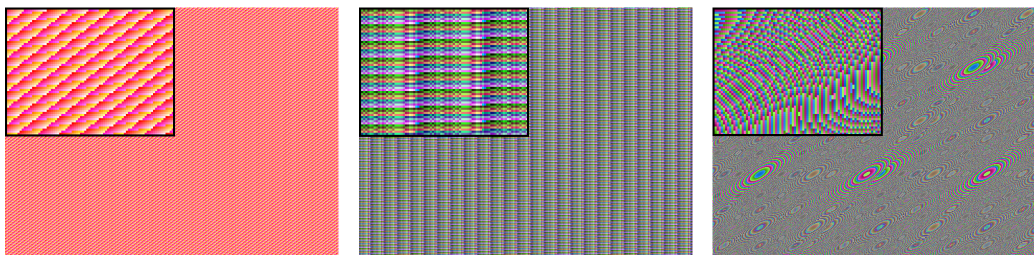
7.2.3 Phase 2

Phase 2 was the conversion of an image to grayscale (or another image alteration). At Clemson University, 50% turned in a correct solution for the second phase, with a few doing creative alterations. See Figure 7.9.

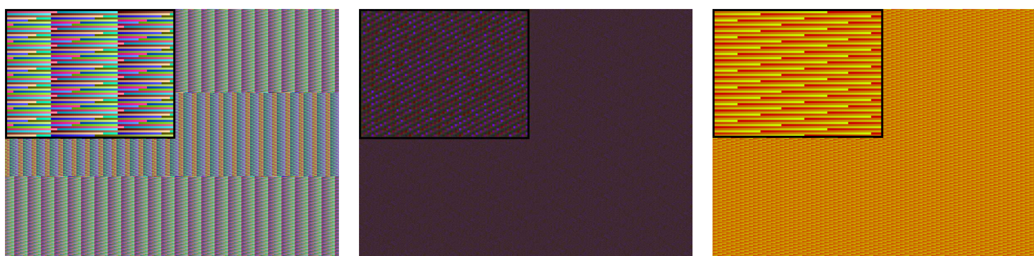
7.2.4 Phase 3 at Clemson University

Phase 3 was a more complicated modification that required buffering the entire image. At Clemson University, of the remaining 27 students (fewer students due to standard attrition), nearly 50% completed a working image effect (Figure 7.10), with five students programming the more complex convolution filter (blurring, sharpening, or edge detection). See Figure 7.11.

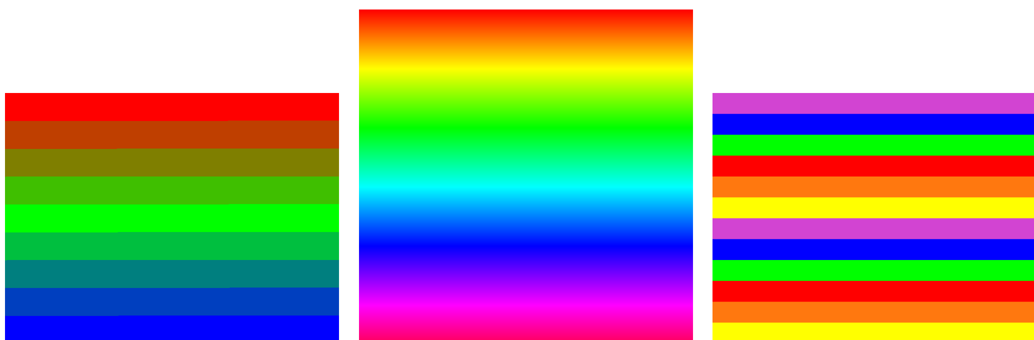
المنارة للاستشارات
www.manaraa.com



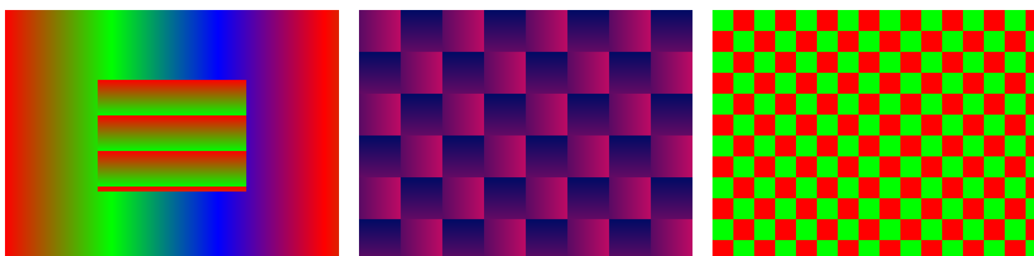
(a) By student A. McKerihan (b) By student H. Scott (c) By student J. Fleming



(d) By student J. Lewis (e) By student J. Swanson (f) By student T. Wigboldy



(g) By student A. Alms (h) By student J. Menard (i) By student J. Lawing



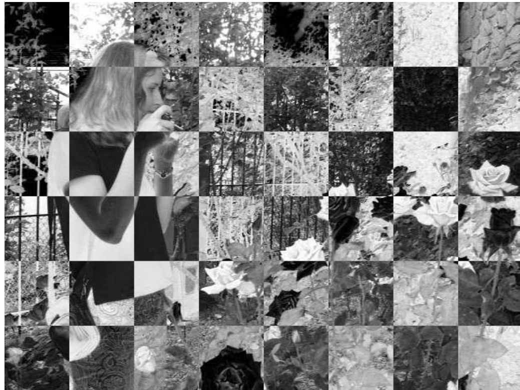
(j) By student J. Davis (k) By student N. Jenkins (l) By student C. Stow

Figure 7.8: 2006 Covenant CPSC 101 Phase 1

المنارة للاستشارات
www.manaraa.com

(a) By student Y. Feaster

(b) By student M. Rardon



(c) By student C. Daugherty

(d) By student V. Tan

Figure 7.9: 2005 CPSC 101 Phase 2

www.manaraa.com

(a) Rotate 90 by C. Gulotta

(b) Vertical Flip by J. Leyh

(c) Rotate 180

(d) Horizontal Flip by T. Steel

Figure 7.10: 2005 CPSC 101 Phase 3

المنارة للاستشارات

www.manaraa.com

(a) Blur by M. Rardon

(b) Sharpen by Y. Feaster

Figure 7.11: 2005 CPSC 101 Phase 3: Convolution Filters

7.2.5 Phase 3 at Covenant College

Covenant College students demonstrated a great deal of creativity in exploring image manipulations. Given a source image of Carter Hall on the college campus, students applied color modifications (Figure 7.12), filters (Figure 7.13 (a-c)), image rotation, and in one case, an imaginative image twist algorithm (Figure 7.13 (d-f)).

7.2.6 Phase 4

Working in pairs on the final project of writing the color transfer program, two thirds of Clemson students were able to provide a correct solution for the color transfer algorithm, generating the effects in Figure 7.14. Similarly, two thirds of Covenant College students were able to complete a correct solution.

7.2.7 Survey Results

In the Fall of 2005 at Clemson, students were randomly assigned to two different curriculum tracks: the $\tau\acute{\epsilon}\chi\nu\eta$ track, and the standard track (called the control group). The control group learned programming under the "Object Early" approach using Java and a GUI programming interface (BlueJ™). At the end of the first semester, 20 students from the $\tau\acute{\epsilon}\chi\nu\eta$ group and 16 from the control group completed surveys. Charts

المنارة للاستشارات
www.manaraa.com

(a) Original (b) By A. McNaughton (c) By J. Menard

(d) By J. Lawing (e) By J. Davis (f) By J. Larkins

Figure 7.12: 2006 Covenant CPSC 101 Phase 3

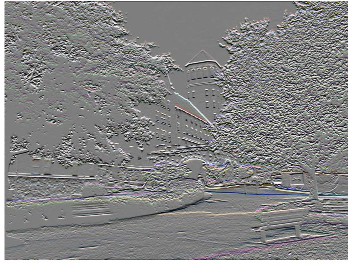
on the answers are coupled with answers on the second-semester survey. The questions covered students' feelings about the courses and tested their skills.

7.2.7.1 View of the Course and Its Success

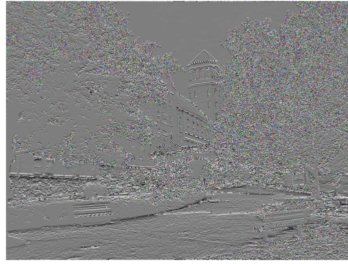
Questions were answered on a scale of 1-5, with 5 being the best and 1 being the worst. Table 7.1 below provides the means, medians, and standard deviations for both groups. $\tau\acute{\epsilon}\chi\nu\eta$ students felt they knew the language more ($P < .0025$), considered the resources more useful ($P < .005$), would be more likely to recommend CS ($P < .01$), more strongly considered CS a good decision (not significant), and felt they expressed their creativity more than the control group ($P < .0005$). The control group slightly preferred working with others, and reported nearly identical liking for CS and problem-based learning (all statistically insignificant).

$\tau\acute{\epsilon}\chi\nu\eta$ students outperformed the control group on all skills tests, but not all mean difference were statistically significant. The population sizes were not large enough to justify t-tests for differences in means, so we used a Monte Carlo version of Fisher's permutation test [24]. $\tau\acute{\epsilon}\chi\nu\eta$ students' feelings about creative expression, language knowledge, and recommendations were significantly (0.05 or better) better than the

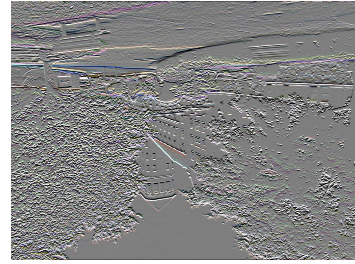
www.manaraa.com



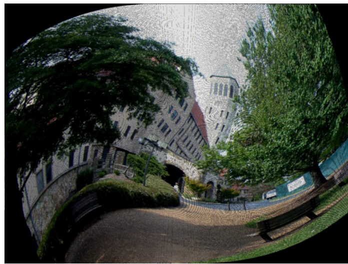
(a) By C. Stow



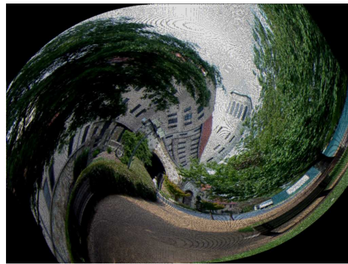
(b) By J. Lewis



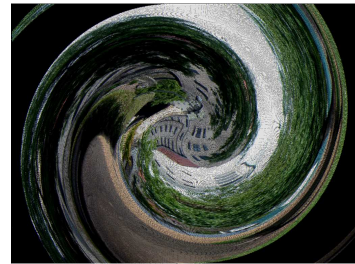
(c) By T. Wigboldy



(d) By A. Musser



(e) By A. Musser



(f) By A. Musser

Figure 7.13: 2006 Covenant CPSC 101 Phase 3

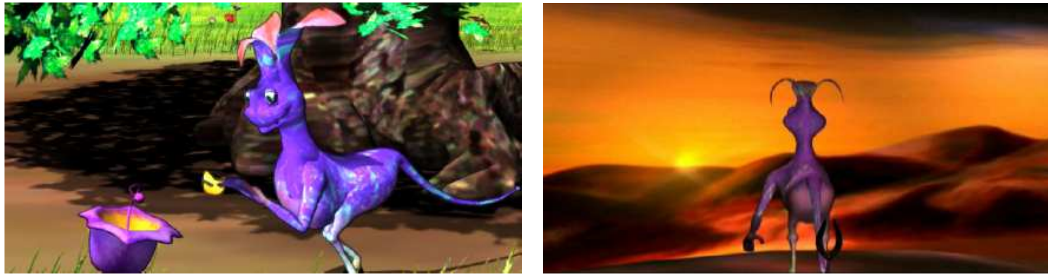
(a) Original

(b) Bright colors

(c) Sunset

Figure 7.14: 2005 CPSC 101 Phase 4 Color Transfer

المنارة للاستشارات
www.manaraa.com



(a) Bright color source

(b) Sunset source

Figure 7.15: CPSC 101 Phase 4 Color Transfer Sources

	$\tau\acute{\epsilon}\chi\nu\eta$			Control		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Know language	3.2	3	.93	2.3	2	.70
Like CS	2.0	2	1.21	2.0	2	.63
Like working with others	2.2	2	.88	2.2	2	.91
Useful resources	4.2	4	.59	3.5	4	.82
Recommend CS	3.9	4	1.02	2.9	3	1.29
CS right decision	4.2	4.5	1.09	3.6	4	1.09
Expressed creativity	3.4	3	1.14	2.1	2	.77
Prefer PBL	2.9	3	1.37	2.9	3	1.59

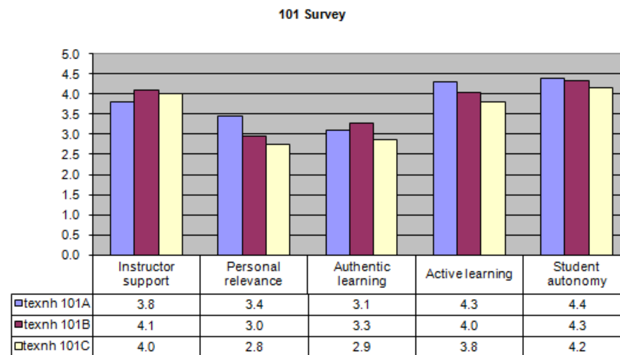
Table 7.1: 2005 101 Comparison

www.manaraa.com

control group, as were their performances on the code reading test.

7.2.7.2 Classroom Environment Survey

Walker and Fraser [72] report a strong correlation between conventional measures of student performance (grades, test scores, etc.) and perceptions of the classroom environment. Based on their observations of numerous studies, we used 28 items of their 36-item survey to gauge students in five areas: instructor support, personal relevance, authentic learning, active learning, and student autonomy. The sixth category (8 questions) covers student interaction and collaboration, and while Quest-Oriented Learning does encourage open problem discussion (but individual work), many instructors still restrict student interaction, rendering the results in that category less meaningful. During the 2006-2007 school year (one year after the previously



	Instructor support	Personal relevance	Authentic learning	Active learning	Student autonomy
texnh 101A	3.8	3.4	3.1	4.3	4.4
texnh 101B	4.1	3.0	3.3	4.0	4.3
texnh 101C	4.0	2.8	2.9	3.8	4.2

Table 7.2: 2006-2007 101 Walker-Fraser Surveys

mentioned surveys), three sections of *τέχνη* 101 were given the Walker-Fraser classroom environment perception survey. The surveys were taken by seven students from one section, eleven from the second, and thirteen from the third. Since all three courses were to follow the *τέχνη* model, the only thing that can be shown by the survey results is the consistency of the courses between semesters and instructors, See 7.2. Active learning and student autonomy seem to be the strongest points of this and many of the *τέχνη* courses.

7.3 Computer Science II

In CS2, students now create raytracers as was originally done in the experimental 215 course. The students work toward the final raytracer in phases, resulting in attractive, realistic images. These raytracers depend on the use of structures, unions, function pointers, trigonometric functions, and eventually justify the use of object-oriented programming.

المنارة للاستشارات
www.manaraa.com

7.3.1 Images at Clemson University

Despite the few geometries covered in class, students in the Spring 2006 course produced creative images (Figure 7.16, published in [17]). Students seemed enthusiastic about the project, evidenced by course evaluations and the quality of work. Students routinely went beyond the requirements of the projects to produce more advanced effects that they could show off [17].

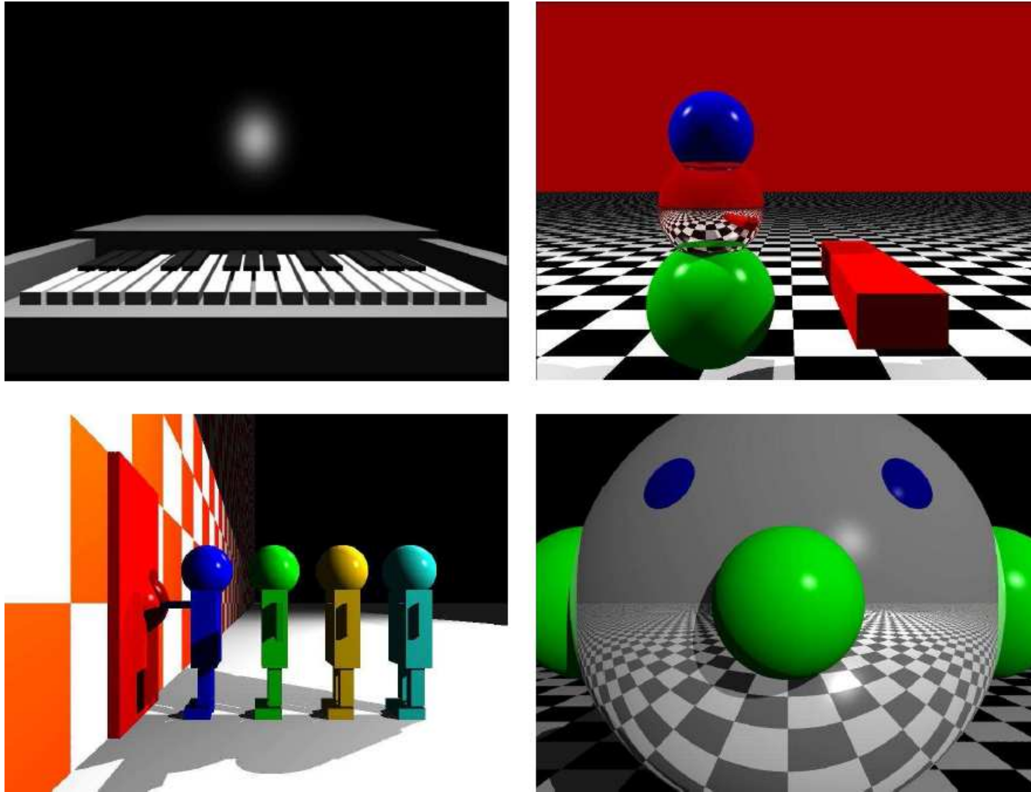
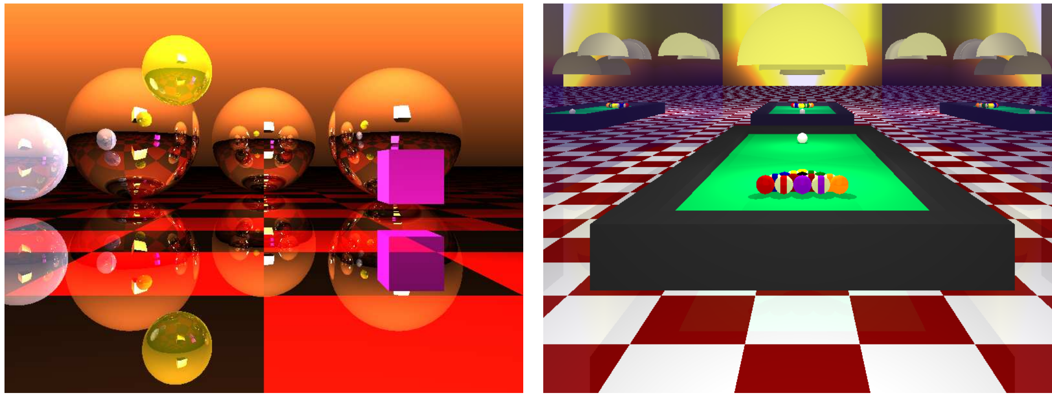


Figure 7.16: 2006 CPSC 102 Student Images

7.3.2 Final Phase at Covenant College

The Spring 2007 class at Covenant College was composed of two computer science majors, seven pre-engineering majors (on track to finish at Georgia Institute of Technology), and three mathematics majors. Despite a minority of computer science majors, the students demonstrated immense creativity with their raytracers, as seen in Figure 7.17.

المنارة للاستشارات
www.manaraa.com



(a) Final Student Image

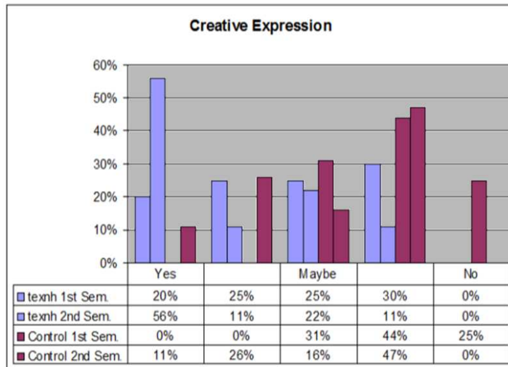
(b) Final Student Image

Figure 7.17: 2007 Covenant College CS2 Images

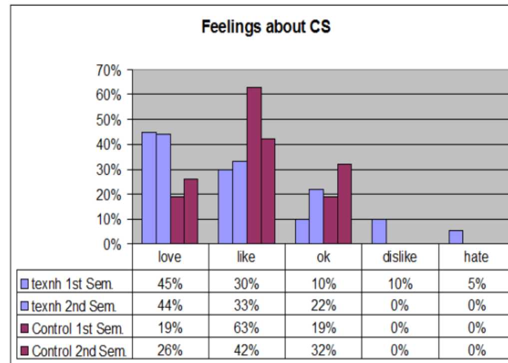
7.3.3 Survey Results

Continuing with the two groups begun in the fall of 2005 for CPSC 101, students were in two groups for CPSC 102 with the control group using the standard approach (Java without a GUI interface in the Unix environment). Due to a technical problem with the web surveys, the surveys were not given to students until the beginning of the third semester (data structures) course. Nine $\tau\acute{\epsilon}\chi\nu\eta$ students and nineteen control group students participated in the surveys, and the results of the surveys are in Tables 7.3 and 7.4. Overall, $\tau\acute{\epsilon}\chi\nu\eta$ students reported better feelings toward CS and teamwork (insignificant), were more likely to think CS was the right decision (not significant), were better able to express their creativity ($P < .02$), and were more likely to recommend CS (insignificant). The control group felt they knew their language at a slightly higher rate, preferred PBL, and considered the resources as helpful at the $\tau\acute{\epsilon}\chi\nu\eta$ students did (all statistically insignificant). The means, medians, and standard deviations of the questions are in Table 7.5. Once again, $\tau\acute{\epsilon}\chi\nu\eta$ outperformed the control group in all skills assessments. The skills assessment section was composed of eight questions covering data representation, a basic linked-list function, writing a recursive function, the recognition of a sorting algorithm (quicksort), and prediction of the number of times the algorithm will loop. See Table 7.4 for 101-102 skills comparison and the overall skills assessment for 102. The average results were significantly better for $\tau\acute{\epsilon}\chi\nu\eta$ students($P < .01$).

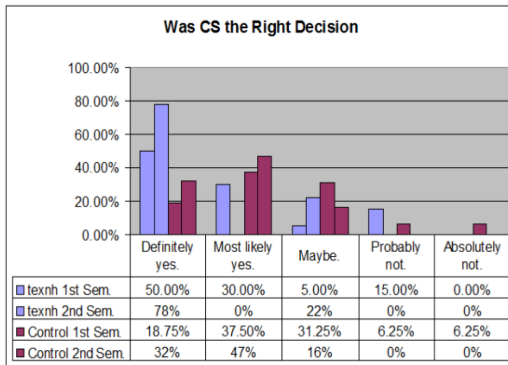
www.manaraa.com



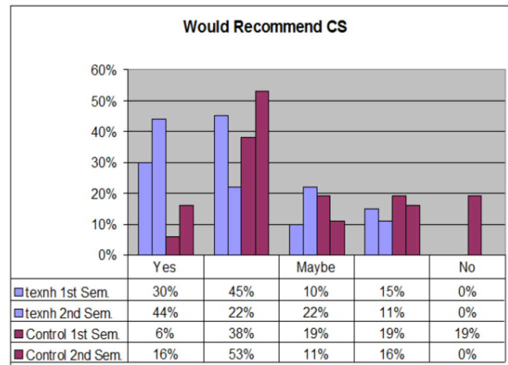
(a) Able to Express Creativity



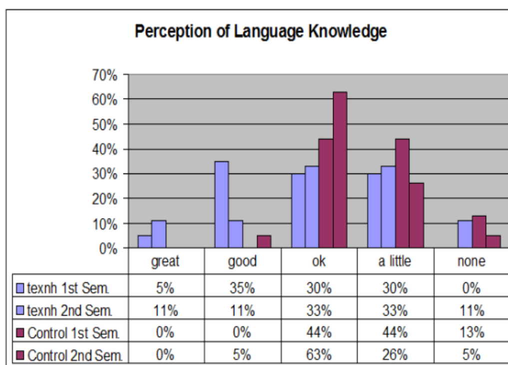
(b) Feelings Toward CS



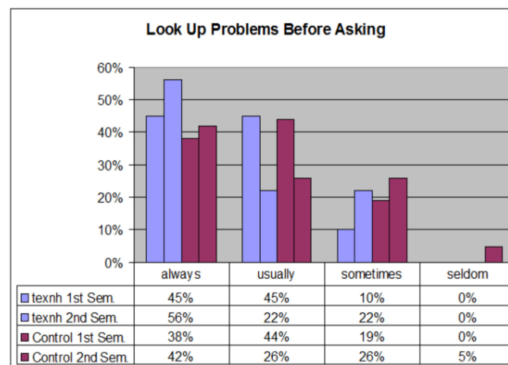
(c) Believe CS to be the Right Decision



(d) Likeliness to Recommend



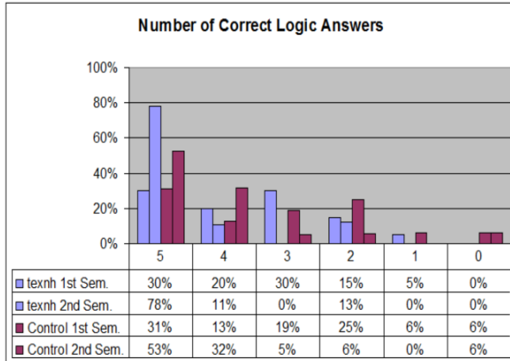
(e) Knowledge of the Language



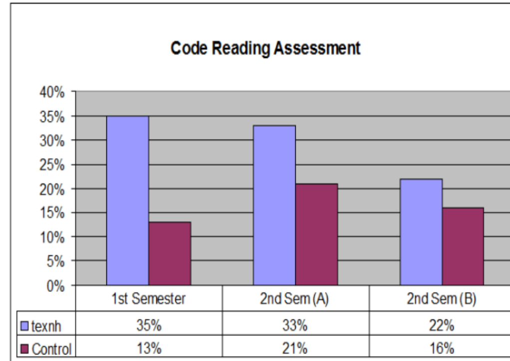
(f) Willingness to Look Up Problems

Table 7.3: 2006 Student Perceptions of 102

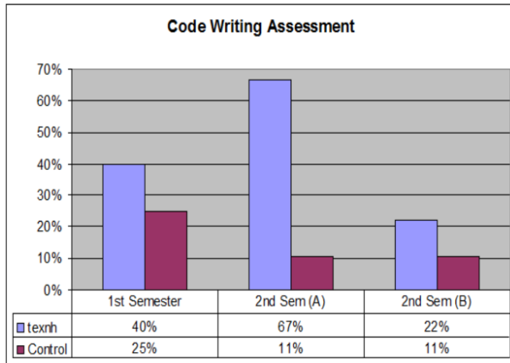
المنارة للاستشارات
www.manaraa.com



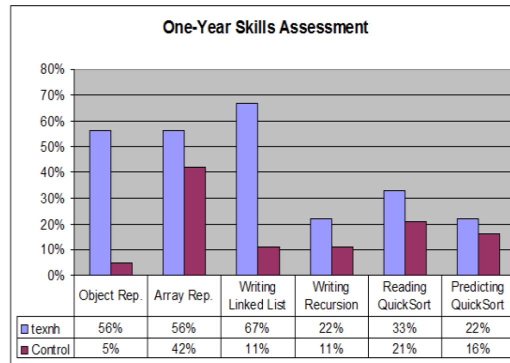
(a) Logic Questions Answer Right



(b) Code Reading Abilities



(c) Code Writing Abilities



(d) 102 Overall Skills

Table 7.4: 2006 102 Skills Comparison

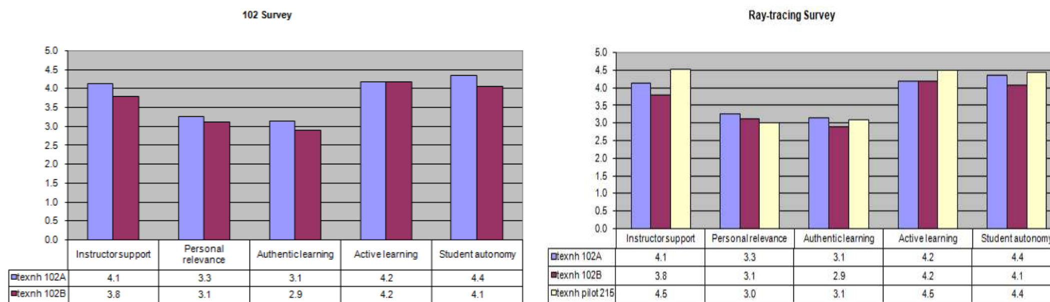
	$\tau\acute{\epsilon}\chi\nu\eta$			Control		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Know language	3.2	3	1.2	3.3	3	.7
Like CS	4.2	4	.8	3.9	4	.8
Like working with others	4.0	4	.7	3.8	4	.9
Useful resources	4.1	4	.6	4.1	4	.8
Recommend CS	4.0	4	1.1	3.5	4	1.3
CS right decision	4.6	5	.9	3.9	4	1.2
Expressed creativity	4.1	5	1.2	3.0	3	1.1
Prefer PBL	3.3	3	1.2	3.7	4	1.2

Table 7.5: 2006 102 Comparison

المنارة للاستشارات
www.manaraa.com

7.3.3.1 Classroom Environment Survey

In the spring of 2007, the two sections of 102 took the Walker-Fraser survey, seventeen from one section and nine from the other. Both courses were taught through the implementation of a raytracer and show similar results. See 7.6 (a). Additionally, that semester saw the last of the pivot $\tau\acute{\epsilon}\chi\nu\eta$ 215 taught with the raytracer, and six of those students took the survey. For lack of a better comparison point, the pilot 215 course is compared with the 102 course results 7.6 (b). Results are very similar and demonstrate strong active learning and student autonomy.



(a) $\tau\acute{\epsilon}\chi\nu\eta$ 102

(b) raytracing courses

Table 7.6: 2007 Walker-Fraser Surveys of raytracing Courses

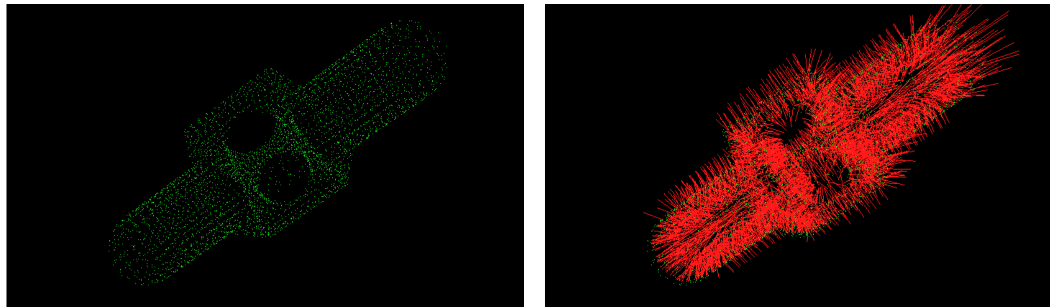
7.4 Second-Year Data Structures Course (212)

In this second-year course on algorithms and data structures, students used an algorithm by Hoppe, DeRose, Duchamp, McDonald, Stuetzle [31] to reconstruct surfaces from unorganized points. This project required the use of advanced tree structures, sorting, graphs, and other typical topics in data structures. The description of this application of $\tau\acute{\epsilon}\chi\nu\eta$ to Algorithms and Data Structures has been accepted for presentation at Eurographics [22] and contains these images showing the phases of surface reconstruction. Figure 7.18 (a) shows the 4102 points representing the mechpart used by Hoppe et al. and used in this course.

7.4.1 Phase I: Tangent Plane Estimation

Tangent plane estimation uses a kd-tree to organize the points, allowing efficient location of the nearest neighboring points. Normals are computed using principal components analysis. See Figure 7.18(b).

المنارة للاستشارات
www.manaraa.com



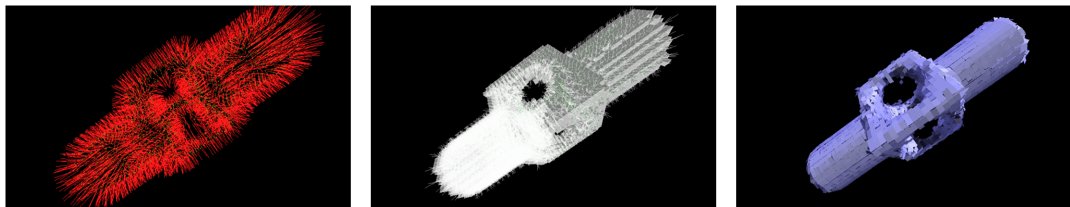
(a) Mechpart points

(b) Tangent plane normals

Figure 7.18: 2006 CPSC 212 Phase 1: Tangent plan estimation

7.4.2 Phase II: Consistent Tangent Plane Orientation

This phase utilizes a minimum spanning tree and a priority queue to generate consistent tangent plane orientation. See Figure 7.19(a).



(a) Consistent plane orientation

(b) Signed distance function

(c) Triangulated surface approximation

Figure 7.19: 2006 CPSC 212 Phases 2-4

7.4.3 Phase III: Signed Distance Function

This phase re-uses the kd-tree to construct a signed distance function on 3D space, where distance is measured to the nearest surface tangent plane. See Figure 7.19(b).

7.4.4 Phase IV: Contour Tracing

This phase uses the marching cubes algorithm to extract the final surface approximation from the signed distance function. See Figure 7.19(c).

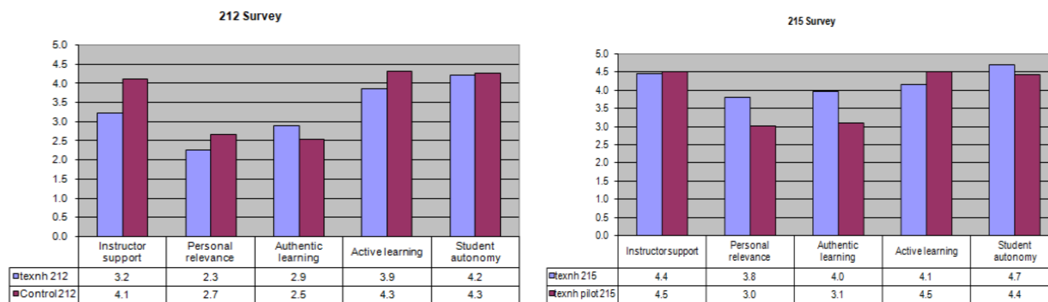
المنارة للاستشارات
www.manaraa.com

The problem for this course was quite difficult and students seemed at times to be overwhelmed. The paper [22] explores improvements that may be made, such as beginning immediately with 3D representations instead of an introductory 2D representation and narrowing of the breadth of potential algorithms and data structures compared (e.g. Kruskal's vs. Prim's algorithm, Euclidean MST vs. Riemannian graph, Fibonacci heap vs. binary heap) in order to keep the focus and momentum of the course.

7.4.5 Survey Results

In the 2006-2007 school year, students were given the Walker-Fraser survey to assess classroom environment, and in the fall of 2006, ten $\tau\acute{\epsilon}\chi\nu\eta$ students turned in the surveys. Unfortunately, not a single student from the control group participated in the end-of-semester survey. Therefore, comparison of the 212 $\tau\acute{\epsilon}\chi\nu\eta$ group to the control group is impossible. However, eleven students in the following semester 212 course, again taught with the standard approach, did take the survey and can be used for comparison.

Table 7.20 (a) demonstrates that $\tau\acute{\epsilon}\chi\nu\eta$ students perceived the class environment to have more authentic learning, but reported below the control group in all other areas. This dip implies that, although students perceived that they were studying were real-world problems and solutions, other aspects of the course suffered, perhaps due to previously discussed issues. Future implementations of the course will seek to better support the classroom environment in all areas.



(a) 2006-2007 CPSC 212 Comparison

(b) 2007 215 Walker-Fraser Results

Figure 7.20: 2006-2007 CPSC 212-215 Comparisons

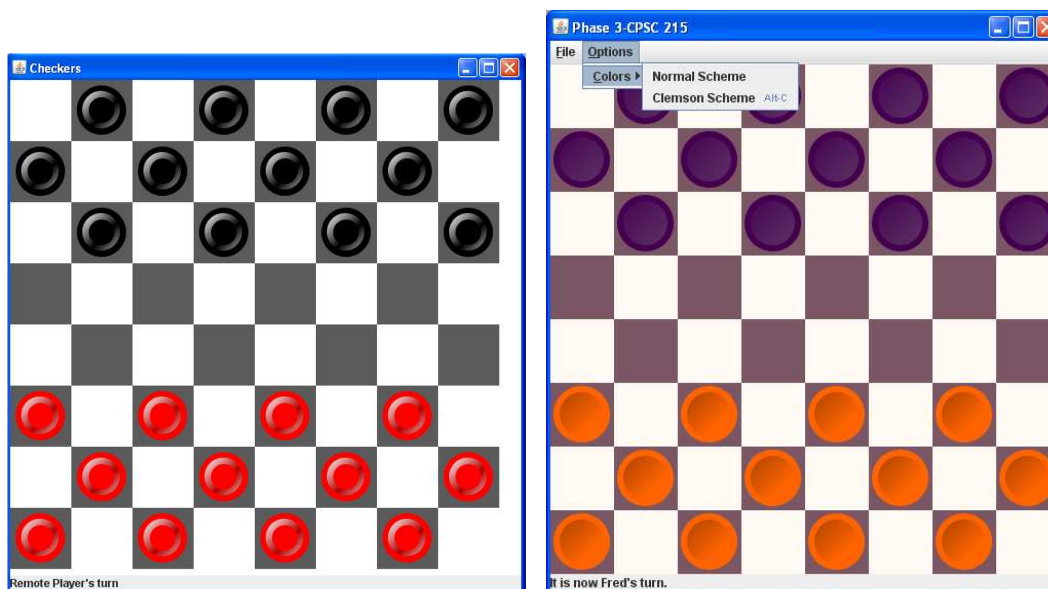
المنارة للاستشارات
www.manaraa.com

7.5 Second-Year, Tools and Techniques for Software Development (215)

In this three-hour course, students learned Java and OO design in order to write a GUI-based, networked chess game. Since Java was new to most of them, the students worked in pairs or groups of three.

7.5.1 Student Programs

The project was done in four phases, the first three of which were checkers games (Figure 7.23. Once the students had all the components of legal piece movements and networking in place, the games were converted to chess (phase four). Students were encouraged to add extra credit features to their games, and all did. Features included selectable piece shapes (7.22 (b)) and colors (7.23 (b)), menus (7.23 (b)), antialiased graphics (7.23 (a)), chat windows 7.23 (b), choice of screen name (7.23 (a)), choice of networking or local game (7.23 (a)), and many other options.

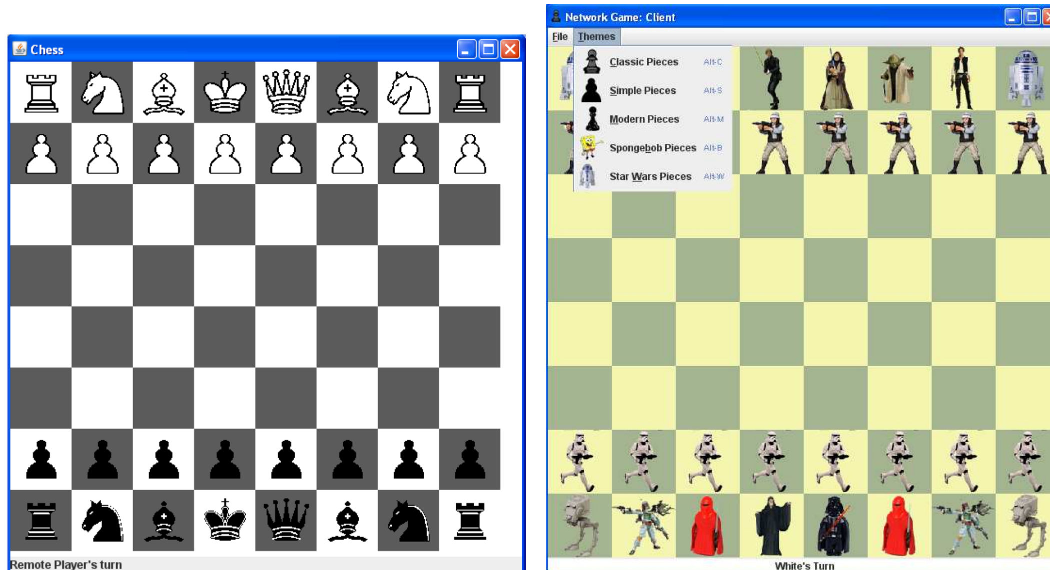


(a) Game by M. Rardon and B. White

(b) Game by C. Daugherty, J. Canter, and Y. Feaster

Figure 7.21: 2007 CPSC 215 Student GUI Checkers Games.

للاستشارات المنارة
www.manaraa.com



(a) Game by M. Rardon and B. White

(b) Game by A. Webber, B. Sterrett, and J. Leyh

Figure 7.22: 2007 CPSC 215 Student GUI Chess Games.

7.5.2 Survey Results

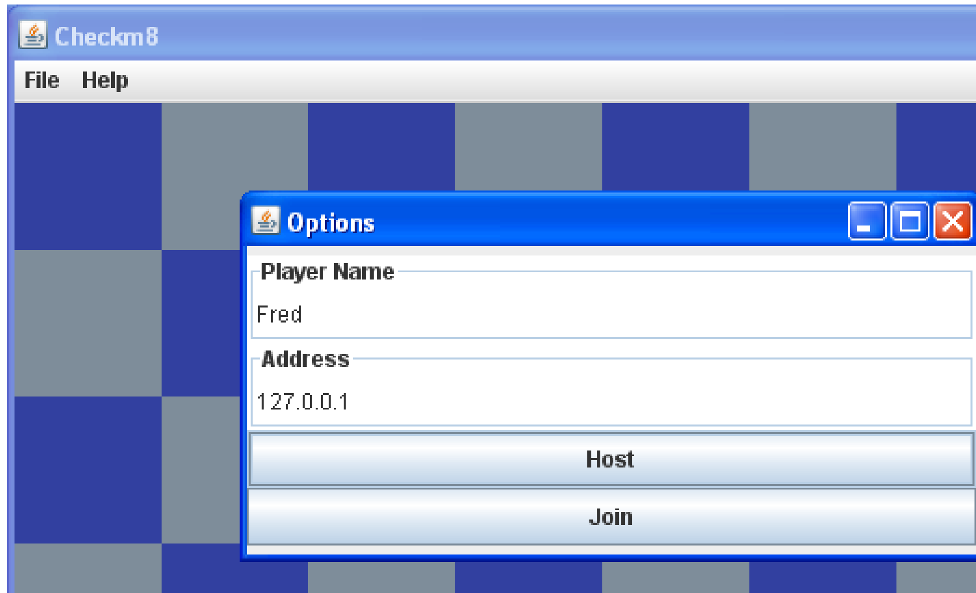
7.5.2.1 Walker-Fraser

In Spring 2007, six students in the pilot 215 (raytracer) course and nine from the new 215 course took the Walker-Fraser survey. While the results of the raytracer course have already been shown with 102 results, it may be of interest to see the courses compared in the Table 7.20 (b).

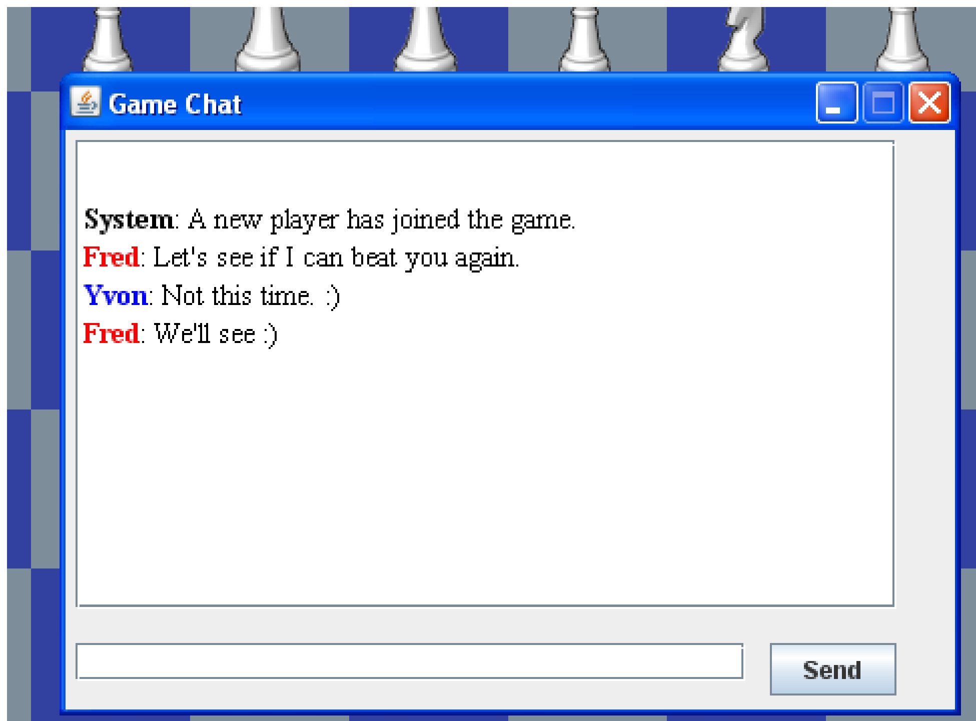
7.5.3 Quantitative Results

In the spring of 2007, 13 $\tau\acute{\epsilon}\chi\nu\eta$ 101 students, 26 $\tau\acute{\epsilon}\chi\nu\eta$ 102 students from two sections (seventeen and nine, respectively), eleven non-$\tau\acute{\epsilon}\chi\nu\eta$ 212 students, six pilot 215 students, and nine $\tau\acute{\epsilon}\chi\nu\eta$ 215 students answered two questions testing understanding of the computer memory model, one on OO design, and one that involved writing an algorithm to solve a simplified programming contest question. The OO design question presented the idea of a Singleton design pattern and provided multiple choice answers for what type of functions and attributes would be necessary to create a Singleton class. The question seems to have been above the levels of the students, and 102 students who had little OO exposure were the most likely to get the answer correct. The simplified programming contest question was not attempted by most people, and though

المنارة للاستشارات
www.manaraa.com



(a) User Name and Host Input



(b) Chat Window

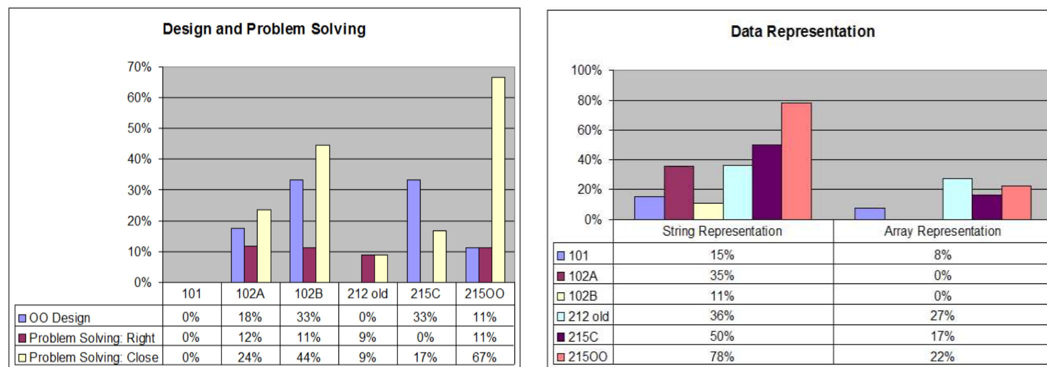
Figure 7.23: 2007 Features in Chess Game by Seagers, Musselman, and Squires

المنارة للاستشارات
www.manaraa.com

many of those who tried achieved answers that were nearly correct, very few were able to correctly answer it. See Table 7.7.

More encouraging were the results of the memory data representation questions. First, the array representation question was correctly answered by a little more than a quarter of the non-$\tau\acute{\epsilon}\chi\nu\eta$ 212 course, with $\tau\acute{\epsilon}\chi\nu\eta$ 215 a close second. The question tested students' knowledge of the storage of 2D arrays as flat arrays in memory.

The other question asked students what the expression, "`while (*t++ = *s++);`" achieved. The majority of the second year $\tau\acute{\epsilon}\chi\nu\eta$ students recognized the operation as a string copy. Second-year students who began with C were substantially more likely to understand the expression than those who had not. Interestingly enough, Joel Spolsky refers directly to this fast string copy, saying "I don't care how much you know about continuations and closures and exception handling: if you can't explain why '`while(*s++=*t++);`' copies a string, or if that isn't the most natural thing in the world to you, well, you're programming based on superstition, as far as I'm concerned: a medical doctor who doesn't know basic anatomy, ..." (*Advice for computer science college students*, January 2005, http://www.joelonsoftware.com/).



	101	102A	102B	212 old	215C	215OO
OO Design	0%	18%	33%	0%	33%	11%
Problem Solving: Right	0%	12%	11%	9%	0%	11%
Problem Solving: Close	0%	24%	44%	9%	17%	67%

(a) OO and Problem Solving

	String Representation	Array Representation
101	15%	8%
102A	35%	0%
102B	11%	0%
212 old	36%	27%
215C	50%	17%
215OO	78%	22%

(b) Data Representation

Table 7.7: 2007 215 Walker-Fraser Results

7.6 Retention

In the fall of 2005, 31 students started in $\tau\acute{\epsilon}\chi\nu\eta$ 101, and 30 started in the control group 101. In the spring of 2007, ten of the $\tau\acute{\epsilon}\chi\nu\eta$ students completed $\tau\acute{\epsilon}\chi\nu\eta$ 215 (nearly one third), and six of the control group completed the pilot 215 (one fifth). See Table 7.8. One student from each course was off schedule due

المنارة للاستشارات
www.manaraa.com

to involvement in coop experiences, one from each was a Computer Engineering major, one from the control group was a Chemical Engineering major and one was a General Engineering major, and one from $\tau \acute{\epsilon} \chi \nu \eta$ and three from the control group were behind due to failing a course. Thus, both groups retained 13 students in some form. The others either left Clemson University or transfered to unrelated fields. While the retentions may appear similar, students in the $\tau \acute{\epsilon} \chi \nu \eta$ approach were more likely to stay computer science majors and to pass the classes. One may speculate that the $\tau \acute{\epsilon} \chi \nu \eta$ courses provide the motivation to keep students on track, but more information is needed to come to that conclusion.

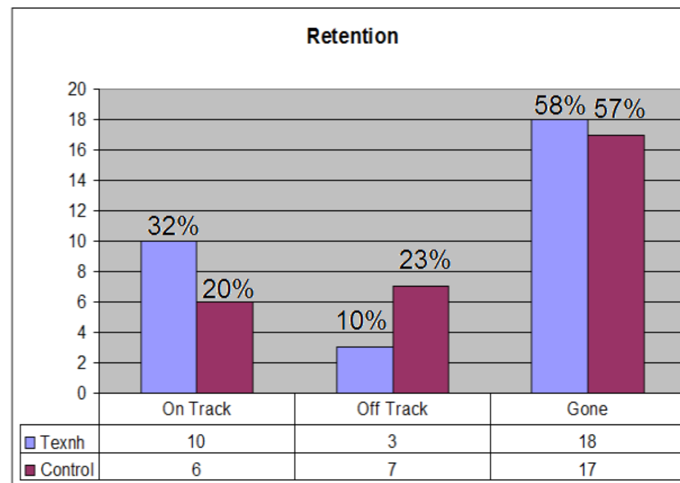


Table 7.8: Retention

7.7 Observations

In addition to an insignificantly better retention rate, better performance on one-year skills assessment, and improved enjoyment of the CS major, some positive observations have been made about the students who have come through the first four-semester sequence of the $\tau \acute{\epsilon} \chi \nu \eta$ curriculum. First, students of this group consistently started assignments early enough to complete them on time. In $\tau \acute{\epsilon} \chi \nu \eta$ 215, only one team on one assignment was unable to complete the program on time. In contrast, a large minority of students in the control group turned in assignments late or incomplete. The behavior of the experimental group is counter to the culture of procrastination typical with beginning computer science students, and may be due to the size of the assignments, the interest generated by the assignments, and the expectation of others in the class that starting early was the right approach, by virtue of the previous two reasons.

Second, the students in the experimental group seemed to get along uncommonly well. The cama-

المنارة للاستشارات
www.manaraa.com

raderie was evidenced by the nearly universal initiative on the part of the students to help each other with non-academic problems, such as when one student was hospitalized. Additionally, in the fourth semester, when teams were required for every class assignment, team selection for the students was quickly (and apparently easily) done, even with groups who had not previously worked together. Not one team from the experimental group ever complained of team or teammate problems, and all their assignments were good quality.

Finally, students who learned under $\tau\acute{\epsilon}\chi\nu\eta$ seem to exhibit less than the typical hubris associated with computer science students. While students were confident in their abilities to solve problems, no student seemed to think himself/herself better than any other student, despite acknowledged differences in skills. Students were comfortable enough with each other to discuss comparative strengths and weaknesses while breaking up projects. This importance of these students' abilities to work together well may be seen by comments made by the founder and chairman of the world's largest software company. On May 30, 2007, Microsoft Chairman Bill Gates and Apple CEO Steve Jobs were interviewed by Kara Swisher and Walt Mossberg at the D5 conference. During the following question and answer session, Gates stated that one of the greatest challenges in building Microsoft, and the area in which he likely made the most mistakes, was getting people with broad skills sets to work well together. Thus, being an asset to a company is more than being a good researcher and programmer but also requires the social skills to put those talents to use.

It appears that $\tau\acute{\epsilon}\chi\nu\eta$ students, with their significantly better computational problem solving skills at the one-year assessment (compared to the control group), their apparent abilities to work with and learn from others, and their expressed creativity, are on track to becoming what we define as "good computer scientists." They are more likely to remain computer science majors and are likely better suited for upper-level classes and industry careers with their disdain for procrastination, comfort with teamwork, and open-mindedness to other programmers' ideas. We eagerly anticipate their success in future classes and in following careers.

المنارة للاستشارات
www.manaraa.com

Appendices

المنارة للاستشارات
www.manaraa.com

Appendix A CS1 Guide

A.1 Credits

4 (3 hours lecture and 2 hours lab)

A.2 Prerequisites

MTHSC 105, or satisfactory score (520) on the Mathematics Level II Achievement Test, or consent of the instructor. Students are not expected to have programming experience. Nevertheless, this is not a general introduction to computing. It is intended primarily for computing majors and minors.

A.3 Course Goals

This course teaches the following computer science skills and techniques:

- Understanding the basic process of problem-solving using a computer.
- Understanding the basic machine memory model.
- Developing competence in the Unix environment.
- Developing the ability to implement simple computer programs.

A.4 Course Description

This course, like all $\tau\acute{\epsilon}\chi\nu\eta$ courses, is based on a large, semester-long project from the visual problem domain. CS1 is focused on the implementation of a color transfer program that will apply the color scheme from one image to another. The project is done in phases, beginning with the creation of a simple image file.

A.5 Resources

For image processing, a good reference is *Digital Image Processing*, 2nd edition by R. C. Gonzalez and R. E. Woods. (2001. Addison-Wesley Longman Publishing Co., Inc.).

For color transfer, the source paper is a great reference: *Color Transfer between Images.* by E. Reinhard, M. Ashikhmin, B. Gooch, and P. Shirley (2001. IEEE Comput. Graph. Appl. 21, 5 (Sep. 2001), 34-41.)

المنارة للاستشارات
www.manaraa.com

A.6 Lesson Guide

A.6.1 Suggested Course Policies

- Recommended textbooks: *The C Programming Language, Second Edition* by Brian W. Kernighan and Dennis M. Ritchie and *Bulletproof Unix* by Tim Gottleber
- Required service: *http://www.turingscraft.com/* Codelab is an online service the students must pay for that provides an opportunity for practicing programming with immediate feedback. Sections (or department/school) can be assigned with deadlines at the instructor's discretion. One approach that has been found to be effective is to require each student to complete 100 exercises before receiving a copy of the midterm examination and 200 exercises (total) before receiving a copy of the final examination. The benefits of this approach include allowing students to work at their own pace and reduced pressure, since the exercises are not included in their grades.
- Presentations. Each student should do one 5-minute presentation on a topic helpful to the class that will not be incorporated into the lecture or lab class. These presentations can be done each day during the opening minutes of class. The purpose of these presentations is to educate students on how to find information about a topic and how to explain it to others. Suggested topics for this class are Unix commands. Once the students have learned about "man," they can easily look up and present information about any other typical Unix command. For a list of possible topics, see the terms used in the example ice breaker below. Students should name the command, explain the reason for its name, explain its typical useful, and demonstrate how to use it.
- Assignments
  - On-time assignments have a maximum value below 100% to encourage the addition of extra credit features.
  - Style contributes to assignment grades. Suggested style guidelines:
    - Use a fixed number of spaces for each indentation. Be consistent. Pick a number and stay with it.
    - Use the space bar to indent to avoid differing interpretations of tab size.
    - Every C statement must begin on a separate line.

المنارة للاستشارات
www.manaraa.com

  * If the condition for execution, such as in a while or if statement can be written in the form "(constant==var)" it should not be written in the form "(var==constant)".
  * Comments must be included when they will aid the reader in understanding the program.
  * Use meaningful variable names. Long names are better than short, unclear names.
  * Avoid the single letter 'o' and the single letter 'l' for variable names.
  * Use white space (hard returns) to set off logical ideas.
  * Keep functions short.

- Academic Dishonesty: Some forms of collaboration are beneficial to all and enhance learning; others are not. A clearly stated policy is important. Suggested: cheating will be taken very seriously, resulting in harsh penalties. Since the skills required in this class are also required in the next class, cheating in this class will seriously hamper your ability to be successful in the next class. Appropriate Collaboration:

  1. Sharing class notes with another student.
  2. Discussing anything that was covered in class.
  3. Helping a fellow student locate a bug in his program, provided the following are true:
     - The helper has already completed his program.
     - The helper never types or dictates code for the other student.
     - The helper helps with minor details of small sections, not solving the programming problem for him.
     - The helper signs the other student's honesty sheet. An honesty sheet is a paper listing any help received on an assignment, including the date, the name of the person, the type of help, and the signature of the person. At the bottom of the page, the student must sign that he did the assignment completely by himself with the exception of the listed help, which was all in line with the honesty rules.

  Inappropriate Collaboration

  1. A student showing another student his code.
  2. A student copying code from another student.

المنارة للاستشارات
www.manaraa.com

3. A student stepping another students logically through the program. (Giving his the key to solving the problem.)
4. A student helping other students during a test or quiz.
5. A student doing another student's work (including online assignments).

A.6.2 Ice Breaker

In order to prevent the students from feeling isolated in a new environment with all new information, students should be given the opportunity to get to know each other. Any method of socialization can be used, and below is one possible game in which the instructor first prepares a large collection of pairs of cards containing matching terms. The goal for the student it to find the matching term:

1. Pick a card.
2. Get a pen and paper (or a PDA and a stylus) ready.
3. When directed, find the person with the matching card. (1-2 minutes)
4. Tell each other your names, where you are from, your majors, and why you are taking this class. (If this class is required, why you are in the major that requires this class.) (2-3 minutes)
5. Give the person at least one good way to contact you (email, phone number, IM account). Write down his name and contact information.
6. When directed to, introduce yourself to 3 more people and get contact information from them. (2 minutes)

Cards: On each card, put 1) the term or definition for the card and 2) the matching term/definition the student is looking for. Examples:

1. Kernighan – Ritchie
2. cat – display file contents
3. chmod – set permissions
4. cd – change directory
5. cp – copy

المنارة للاستشارات
www.manaraa.com

6. dos2unix – convert windows files to unix files
7. find – locate files
8. gzip – compress files
9. gunzip – decompress files
10. head – print beginning of a file
11. ls – list files
12. man – manual
13. mkdir – make a directory
14. more – displays one screenful at a time
15. mv – move
16. passwd – change password
17. ps – process status
18. pwd – print working directory
19. rm – remove
20. rmdir – remove directory
21. sftp – secure file transfer protocol
22. ssh – secure remote shell
23. tail – Print end of a file
24. tar – tape archive
25. vi – text editor
26. vim – vi improved

المنارة للاستشارات

www.manaraa.com

A.6.3 Background Information

1. History of Computing

   - 1623: Mechanical calculator. Wilhelm Schickard invented the first known mechanical calculator, which was capable of simple arithmetic. A similar mechanical adding machine was built in the 1640s by Blaise Pascal. It is still on display in Paris.
   - 1673: More advanced mechanical calculator. It was created in 1673 by German mathematician Gottfried Leibniz, and it was capable of multiplication and division. It was purely mechanical with no other source of power.
   - 1823: Charles Babbage began work on the Difference Engine. He designed it, but it was completed by a Swedish inventor in 1854.
   - 1833: Charles Babbage began work on the Analytical Engine. It was never completed, but it introduced an important concept: a general-purpose machine capable of performing different functions based on programming.
   - 1834: Ada Byron (Lady Lovelace, daughter of poet Lord Byran) was impressed with the concept of the Analytical Engine at a dinner party. She created plans for how the machine could calculate Bernoulli numbers. This is regarded as the first "computer program," and she as the first "programmer." The Department of Defense named a language "Ada" in her honor in 1979.
   - 1890: punched cards were used by Herman Hollerith to automate the Census. The Concept of programming the machine to perform different tasks originated from Babbage. Punched cards were based on Joseph Marie Jacquards device to automate weaving looms. Hollerith founded a company that became International Business Machines (IBM) to market the technology.
   - 1939: prototype of the first electronic computer. It was assembled by John Atansoff and Clifford Barry. John Atansoff proposed the concept of using binary numbers. Completed in 1942 using 300 vacuum tubes, it could solve small systems of linear equations.
   - 1946: ENIAC (Electronic Numerical Integrator and Computer) was completed by Presper Eckert and John Mauchly. It used 18,000 vacuum tubes and occupied a 30 by 50 foot room. It could be programmed by plugging wires into a patch panel. Because this style of programming required intimate knowledge of the computer, it was very difficult to do.

المنارة للاستشارات
www.manaraa.com

- 1946: John von Neumann architecture stored-programming concept. He (and others) suggested that programs and data could be represented in a similar way and stored in the same internal memory. All modern computers store programs in internal memory.
- Four generations of computers
  (a) Vacuum tube (1939)
  (b) Transistor (invented in 1947, used in IBM 7090 in 1958)
  (c) Integrated circuit or chip (invented in 1959, used in IBM 360 in 1964), which is a small wafer of silicon that has been photographically imprinted to contain a large number of transistors.
  (d) Large-scale integration: microprocessor (1975). The entire processing unit is stored on a single chip of silicon.
- In the early 1970s, Robert Noyce, one of the inventors of the integrated circuit and founder of Intel speaking of a computer chip compared to the Eniac: "It is 20 times faster, has a larger memory, is thousands of times more reliable, consumes the power of a light bulb rather than that of a locomotive, occupies 1/30,000 the volume and costs 1/10,000 as much" (Source: Roberts, Eric S. *The Art and Science of C*. Addison-Wesley Publishing Company, 1995).

2. Computer Hardware

- Central Processing Unit (CPU)
  - Brain of the computer
  - Performs the actual computation and controls the activity of the entire computer.
  - In current computers, a CPU is an integrated circuit
    * Consists of a tiny chip of silicon with millions of transistors imprinted onto it.
    * Capable of carrying out simple arithmetic and logical operations
- Primary Storage: Main Memory
  - Type of data storage device: a piece of hardware capable of storing and retrieving information
  - Storage device used while the program is actively running
  - Current computers use RAM (random access memory) chips.
    * Random access means that any location can be accessed at any time. (Like a DVD as opposed to a cassette tape.)

المنارة للاستشارات
www.manaraa.com

    * Built from a special integrated-circuit chip
    * Very fast, but relatively small
    * Data is lost when the computer is turned off.
- Secondary Storage
  - Hardware device that stores permanent data.
  - Slower than RAM.
  - Current computers use magnetic disks for data storage, circular spinning platters coated with magnetic material, as well as optical storage, and solid state disks.
  - Examples:
    * Hard drive
    * Floppy disk
    * CD or DVD (optical)
    * Flash drives (solid state)
- I/O (Input/Output) Devices
  - These devices are the tools by which the user communicates with the computer
  - Examples of input devices:
    * Keyboard
    * Mouse
    * Touch screen
    * Scanner
  - Examples of output devices:
    * Graphics card
    * Screen
    * Printer
    * Speaker
- Bus. The components (CPU, GPU, primary storage, secondary storage, I/O devices)are all connected by a communications bus.

المنارة للاستشارات
www.manaraa.com

- Graphical Processing Unit(GPU): high end graphics cards have processors, called GPUs, that handle a great deal of image processing for display and rival CPUs in overall processing capability.

3. Computer Science

   - Related Terms

     – Programming: providing a computer with a set of instructions

     – Software: the name for programs created for a computer. A sequence of steps that can be interpreted by the hardware of the computer.

   - CS Definition: The creation of the steps necessary for a computer to solve a problem.

   - CS Definition: "the science of problem solving in which the solutions happen to involve a computer" (*The Art and Science of C* by Eric S. Roberts. Addison-Wesley, 1995).

4. Exercise in giving instructions. This suggested exercise exposes students to the concepts involved in giving precise instructions to a machine and thus how a CPU works. Together, the students work out the commands needed by a robot to perform addition.

   - Robot's Abilities

     (a) Speaks and understands English.

     (b) Can read and write any word or number.

     (c) Can search for a number or word, given a starting and ending point.

     (d) Obeys any written command it understands.

   - Limitations

     (a) Does not have short term memory. (It cannot even remember a number if it does not write it down.)

     (b) After it performs an instruction, it immediately forgets what it did.

     (c) Cannot do arithmetic at all.

     (d) Cannot count.

     (e) Cannot make decisions. It simply obeys commands.

     (f) Can obey only 2 spoken commands:

www.manaraa.com

– "Write the following instructions at location xxx: . . ."

– "Begin sequentially following instructions starting at location xxx."

- Resources

(a) Has a list of numbers 1 through 10 at the location on the board starting with 100.

(b) Has a list of blanks for writing numbers or words starting at location 200.

(c) Has a list of blanks for writing instructions to follow starting at location 300.

- Tasks

(a) Give instructions to a robot to add 6 and 2 and give the answer in the format "Six plus two equals xxx." The class will work through this. It will take a few tries to understand the problem and solution, so do not be afraid to guess.

(b) Break up into groups of five (5) and make instructions for a robot to add any two numbers from 1 to 10 whose sum is no larger than 10. (e.g. 4+5, 3+7, etc.) Same format for answer: "xxx plus xxx equals xxx."

(c) In your same group, make instructions for a robot to add any two numbers from 1 to 100 whose sum is no larger than 100. (e.g. 53+37, 95+2, etc.)

(d) Participate in instructor-led discussion of how this example relates to real computers.

- Example solution

(a) Write the following instructions beginning at location 300.

i. Write the number "6" at location 200.

ii. Write the number "2" at location 201.

iii. Write "100" at location 202 to represent the starting location for the search for the number stored at location 200.

iv. Check if the memory location stored at 202 holds the same number as is stored at location 200. As long as it does not, update location 202 to hold the next memory address. (After this loop is complete, location 202 holds the address of the number held at location 200 (6).)

v. Write "100" at location 203 to represent the starting location for the search for the number stored at location 201.

www.manaraa.com

vi. Check if the memory location stored at 203 holds the same number as is stored at location 201. As long as it does not,

A. Update location 202 to hold the next memory address.

B. Update location 203 to hold the next memory address.

vii. Write on the board the value at location 200, then " plus ", then the value at location 201, then " equals ", then the number at the location stored at location 202.

(b) Execute instructions from line 300.

5. Introduction to levels of languages

   (a) Lowest level: absolute machine code (binary code)

   (b) Assembly language

   - Symbolic representation of machine code
   - Translated into binary code with an assembly

   (c) High-level language/Compiled Language

   - One high-level command translates to many assembly commands
   - A compiler converts the high-level source code into machine language for the appropriate computer

   (d) High-level language/Interpreted Language: similar to a high-level language but the translation from the high-level command to machine code it performed at run-time.

6. Introduction to compiler concepts. A compiler is a program that converts a higher-level language to a lower-level language.

7. Introduction to the types (levels) of software

   - Application Programs, e.g. editors
   - Utility Programs, e.g. compilers
   - Operating Systems

8. Introduction to the Unix operating system: history of Unix.

   - 1964: AT&T Bell Labs, General Electric, and MIT wanted to create a multi-tasking, multi-user operating system.

المنارة للاستشارات
www.manaraa.com

  - Previous computers were single threading: one program at a time.
  - The new operating system named Multics (MULTiplexed Information and Computing Service)
- Four years later, Bell Labs withdrew from the project, since it was able to support only 3-4 concurrent users.
  - The Multics team at Bell Labs missed the collaborative environment of all working in the same room.
  - Ken Thompson lost his game machine. (Space-travel simulation he developed on the Multics system.)
  - Thompson found an old Digital Equipment Corporation (DEC) PDP-7 computer and moved his code over to that.
- The old Bell Labs Multics group starting developing code for the PDP computer.
  - This was the beginning of the Unix Operating System.
  - Brian Kernighan named the system UNICS (UNiplexed Information and Computing Service).
  - In one month, Ken Thompson built the core of Unix.
  - Bell Labs' management asked about the new toy.
  - Thompson said they were building a text-processing system.
- Management gave them a bigger computer.
  - Unix was written in PDP-7 assembly.
  - The group needed to transition the code from PDP-7 to the newer machine.
  - The group decided to rewrite Unix in the high-level language B (condensed BCPL), but B was too slow (interpreted language).
- Dennis Ritchie created a new, high-level language called C with which they could rewrite Unix.
  - Because Unix was written in a high-level language, it can be ported (moved) to other hardware platforms (computers with different CPUs, memory, and/or peripheral devices).
  - In the 1970s, Bell Labs was not allowed to sell software.
- The team gave away copies of the source code for free.

المنارة للاستشارات
www.manaraa.com

- Unix source code is still shared today in the form of Linux.
- Unix and C became popular, because it offered tremendous computational efficiency, and everyone could get it for free.
- Giving away code necessitated writing documentation about the code.

- The team created documentation to go with the code in the form of man pages.

  Source: Bulletproof Unix by Timothy T. Gottleber. Prentice Hall: 2003.

9. Introduction to C

   - C was developed by Dennis Ritchie in the early 1970s.
   - C is a structural, general-purpose, high-level language.
   - The C language strongly reflects its tie to assembly language and helps students understand how the computer works.
   - Statements in C are terminated by semicolons and spacing does not change the meaning of the program.

A.6.4 Phase 1

Phase one of the semester-long, target problem is the creation of a 1-pixel image in Portable Pixmap (PPM) format that is printed to standard out. Required Material: text editor, C compiler, library functions, I/O functions, preprocessor directives, the main function, PPM format, binary data, ASCII data.

1. Create a program file. With a Unix environment, students can use GUI text editors or console-based text editors. The benefit of console-based text editors is their availability through simple remote connections such as ssh (secure-shell). While editors such as `vi` may be quite powerful, students are likely to initially benefit more from simplicity of operation, and a simple editor such as pico might be a better choice. Whatever they use, A C program file should be given a lowercase name and end with the `.c` extension.

   ```
   pico image.c
   ```

   Once the file is opened, students can modify it as needed and save it.

2. Include the library functions needed to output information to standard out. A function is a group of commands to be executed sequentially, and library functions are available to perform common tasks.

المنارة للاستشارات
www.manaraa.com

In the case of this program, the only functions needed are the standard input and output functions specified in `<stdio.h>` file. These functions allow programmers to read data from the logical file called "standard in" (typically the keyboard) and to write data to the logical file called "standard out" (typically the screen).

To gain access to the needed library functions, programming must "include" the file declaring them. In C, the `#include` directive has the C preprocessor copy the contents of the file specified into the current file.

The C preprocessor is a tool for preparing a C source file to be compiled. Some of the things the C preprocessor handles include removing comments (any text between an opening comment marker `/*` and a closing comment marker `*/`), including specified files, and replacing constant names with their values (such constants are defined with the `#define` directive).

3. Create the main function. The "main" function is the starting point of all C programs. To create the main function, type the name "main" followed by open and closed parentheses (non-empty arguments inside the parentheses will be used later) followed by an opening brace ({). After the code in the main, the function is ended by a closing brace (}). Depending on the version of the compiler, the main may be required to specify a return type of `int`. It is probably best to delay an explanation of its meaning until later.

4. Output to standard out a PPM image file header. Portable Pixmap (PPM) image format is a very simple, uncompressed file format. The format consists of an ASCII header followed by binary image data. Here is the format specification (in order as it occurs in the image file):

   (a) A "magic number" for identifying the file type. A full-color, binary PPM image's magic number is the pair of characters "P6".

   (b) Whitespace (spaces, tabs, carriage returns, line feeds). NOTE: Characters from a "#" to the next end-of-line character are comments and are ignored. Comments can occur anywhere in the PPM header.

   (c) The width, formatted as ASCII characters in decimal.

   (d) Whitespace.

   (e) The height in ASCII decimal.

   (f) Whitespace.

المنارة للاستشارات
www.manaraa.com

(g) The maximum color value in ASCII decimal. It must be less than 65536 and more than zero. The most common value is 255, which indicates one byte per color component (R, G, B) per pixel.

(h) A newline character or other single whitespace character. Note that in Windows, new lines are expressed as two characters. Be sure to restrict the whitespace after the maximum value to a single character.

(i) The line of image data. This is a raster of "height" rows, in order from top to bottom. Each row consists of "width" pixels, in order from left to right. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in pure binary by either 1 or 2 bytes. If the maximum color value is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first. There is no whitespace between bytes. In particular, there are no newline characters in the image data.

The header of the PPM file (all but the raster of data) is in ASCII file format. ASCII stands for American Standard Code for Information Interchange. It is a character encoding used to represent text with values in the range [0,127]. Since all information on a computer is represented numerically, characters must also represented as numbers. Thus, in an ASCII (or text) file, each character is represented by a number. For example, A is represented by 65, B by 66, C by 67, . . . and Z by 90. Lowercase letters are in the range 97-122. C provides functions for outputting ASCII characters to standard out. One useful function is `printf`. `printf` is a function for formatting and printing (outputting) data. e.g.

```
printf ("Hello, World!\n");
```

Notice that "Hello, World!\n" is in quotation marks. These marks specify that
"Hello, World!\n" is text. Additionally, "Hello, World!\n" is in parentheses. These parentheses specify that "Hello, World!\n" is an argument being passed to the `printf` function. The above line of code will output "Hello, World!" and a newline character to standard out. Output devices (displays, printers) interpret the newline character as a signal to perform a line feed and a carriage return. A backslash is considered an escape sequence and the '`\n`' character is a special sequence for representing a new line. This character can be generated from the keyboard by typing "¡ctrl¿ j". (Notice that the command ends with a semicolon. C statements are terminated by semicolons.) Thus `printf` can be used to output the PPM header.

5. Output the binary image data to standard out. The image data is not ASCII but is instead in binary format. The colors of the pixels in a PPM file (with a maximum value of 255) are represented by three

المنارة للاستشارات
www.manaraa.com

```
#include <stdio.h>
main () {
   printf ("P6\n1 1\n255\n");
   printf ("%c%c%c\n", 255, 0, 0);
}
```

Algorithm .1: Output of a single-pixel image file

bytes. The first byte is the value of the red channel in the pixel, the second is the value of the green channel, and the third is the value of the blue channel. A red value of of 0 is no red contribution, and a value of 255 is full red contribution. Thus, if the bytes are 255, 0, 0, the color is red. The bytes 255, 255, 255 represent white and 0, 0, 0 represent black. In the example image, only one pixel value needs to be represented, and it will be represented by three bytes of data for the three color channels. The data must not be converted to ASCII format. Note that `printf("255, 0, 0");` would produce nine bytes of output, not three. The printf function uses what are called "format codes" to specify how data should be output. One such format code is `%c`. This format code forces the data to be output directly without conversion to ASCII. Using this format code, a red pixel can be specified in the file with the following command:

`printf(''%c%c%c'', 255, 0, 0);` This command creates the three bytes needed to represent a pixel and gives them values specifying full red contribution with no green or blue contribution.

6. End the main function with a closing brace. The code to produce an image that has only one pixel with a value of red is shown in Algorithm .1.

   Thus, the whole program is the following code and will output a single red pixel: Algorithm .1.

7. Compile the program. If the file is in a Unix environment, use the `gcc` compiler: the GNU project C and C++ compiler. The `gcc` compiler will convert a correct C/C++ program file into executable binary code. The default location of the code is `a.out`. The location of the generated executable file (among other things) can be specified by the arguments to the `gcc` command.

   ```
   gcc image.c
   ```

8. Run the program and redirect the output to a different file. When the image program is executed, by default its output will be printed to the screen. To send the output to a file, use "I/O redirection." I/O redirection allows the user to specify where standard output is written and from where standard input

المنارة للاستشارات
www.manaraa.com

is read. Output is redirected using the "greater than" symbol:

```
./a.out > out.ppm
```

The file name can be anything, but it should end with the PPM extension for identification purposes.

9. View the output file. PPM is not a common image file, but it can be viewed by Gimp, the GNU Image Manipulation Program, which is freely available for many operating systems. The "display" command, which is part of the ImageMagick package, is also available on most Linux systems. Additionally, Unix provides a tool for converting PPM to the more common PNG format:

   ```
   pnmtopng < out.ppm > out.png
   ```

   or, with ImageMagick,

   ```
   convert out.ppm out.png
   ```

A.6.5 Phase 2

Phase two is the creation of an 800 by 600 PPM format image. Required Material: variables, data types, variable declaration, variable assignment, conditional expressions, variable incrementation, counted loops.

1. Create a variable to keep track of the number of pixels that have been printed. An image with a width of 800 pixels and a height of 600 pixels has 480,000 pixels each made up of three bytes. Besides being unnecessary, writing 480,000 printf statements is tremendously difficult. Instead, the computer program can be instructed to repeat a command a specified number of times.

   An instruction specifying that a block of code is to be repeated is called a "loop." If the loop executes a specified number of times, it is called a "counted loop." A counted loop fits the problem of creating 480,000 pixels.

   To use a counted loop, there must be a way to count how many times the loop has been executed. The way to track information such as a counter is a "variable." A variable is a named space in memory that can store information. In this case, the information needed is an integer (a whole number). To create an integer variable in C, specify that the variable is an integer and give it a name. The name can be anything that starts with a letter and is made of only letters, numbers, or underscores. In C, variables

www.manaraa.com

```
#include <stdio.h>
main () {
   int i;
   printf ("P6\n800 600\n255\n");
   for (i=0; i < 480000; ++i) {
      printf ("%c%c%c", 255, 0, 0);
   }
}
```

Algorithm .2: 800 by 600 image creation

must be declared at the top of the block of code. Thus, to create an integer named “i” to track how many times the loop has executed, use the code shown in Algorithm .2.

2. Update the PPM header print statement to specify a width of 800 and a height of 600.

3. Created a counted loop to output the 480,000 pixels. The counted loop in C is the “for” loop statement. There are several parts to the for loop: the keyword “for”, an initialization section for assigning the starting value(s) to use in the loop, a test condition specifying how many times the loop repeats, a step section for updating the variable(s) controlling the loop, and the block of statements to be repeated. In the case of the image, the variable “i” should be given the beginning value 0. The condition for looping is as long as “i” is less then 480,000. (Once it has reached 480,000, all the pixel values have been printed.) The step section should increment “i” to the next numerical value. To assign i to a value of zero, use the expression `i=0;`. To test whether i is less then 480,000, use `i < 480000;`. Note that comments are not used in numbers in C. To increment i to the next value, use the operator ++. The ++ operator changes the value of the variable to be the next whole number. The block of statements to be looped through is contained between opening and closing braces.

4. Compile and execute the program, redirecting the output to a .ppm file. The resulting image should be a solid red image of size 800 by 600. For extra credit, students may add stripes or patterns.

A.6.6 Phase 3

Phase three is reading a file (image file) and printing it back out. This phase draws from previous knowledge and is working toward the goal of reading in images, manipulating them, and outputting them. Required Material: standard input, file streams, unsigned char, conditional loops, bytes, binary data.

1. Create a new file that #includes `stdio.h` and has a main.

المنارة للاستشارات
www.manaraa.com

2. Create a variable to hold each datum as it is read read from the file. Since the file to be read is a PPM image file, the variable must be able to hold ASCII characters and bytes of image data. A "byte" is 8 bits or 8 binary digits. All data on a computer is stored as binary digits (0 and 1). One binary digit (bit) can have two possible values: 0 or 1. Two binary digits can holds four possible values: 00, 01, 10, and 11. Eight digits can hold 256 different values.

   The data type `unsigned char` holds a byte of data with values in the range [0,255] (256 different values). Unsigned chars can thus hold ASCII values which are in the range [0,127] and the bytes of image data, which are in the range [0,255].

3. Read in the first character in the input file "stream." (In a PPM file, the first character is 'P'.) A file stream is an access point for a file within a program. Opening a named file within a program will provide the associated access point or stream name. Standard input is always open, and its stream name is `stdin`. Each time a character is read from a stream, the location from which to read the next character is moved forward. Thus, sequential reads result in sequential values from the stream. A library function in C that can be used to read characters from a stream is `fgetc`. (`scanf`, `fgetc`, and `getchar` are others.) `fgetc` starts with an "f" to indicate it is a file stream-related function, and "getc" indicates that is gets (reads) a character. `fgetc` returns a character read from the specified input file. The input file used for this program is standard input with stream `stdin` in C. To store the value returned from the function, assign to "ch" the result of the function `fgetc()`.

4. If the value assigned to `ch` is valid data and not a marker indicating the end of the file, print it out, and continue to read and print out bytes of data as long as the values are valid. The ends of files in computer file systems are marked with what is called an "end of file" (EOF) character. The actual value of EOF is system dependent, but C provides a function (`feof`) that indicates if the EOF character has been read. The EOF character must have already been read for `feof` to confirm that it has been read. `feof` requires one parameter: the file stream which is being read from: `stdin`. Therefore to test if the EOF character has already been encountered, use `feof(stdin);`. This function returns a value of zero if the end of file has not been encountered yet and a non-zero value if it has. As long as the program has not encountered the end of the file, it should print out the character just read and read a new one. This process may be repeated many times. Unlike the previous example of printing an image however, the actual number of times the commands must be repeated is unknown.

   A conditional loop allows a group of commands to be repeated for an unknown number of times based

المنارة للاستشارات
www.manaraa.com

```
#include <stdio.h>

main () {
   unsigned char ch;
   ch = fgetc (stdin);
   while (feof(stdin) != 0) {
      printf ("%c", ch);
      ch = fgetc (stdin);
   }
}
```

Algorithm .3: Complete program to read in and print back out a file

on a given test condition. In C, conditional loops are called "while" loops. These loops repeat while a given condition has a non-zero value. (In C, zero is false and non-zero is true.) To repeatedly read characters, confirm they are not at or past the end of the file, and print them out, use a 'while' loop. The while loop has 3 parts: the keyword "while", the condition in parentheses, and the block of statements to repeat inside opening and closing braces. The condition for looping for this program is to continue as long as `feof` returns a false value. Since while loops continue for true values (not false values), the returned value of `feof` must be negated, using the "!" operator. A "!" before a true expression results in a false expression, and a "!" before a false expression results in a true. Algorithm .3 pulls all this together: testing whether standard in has been read past the end of file, printing out the character read, and reading the next character.

5. Compile and run the program, redirecting input using the < `filename` notation and redirecting output using the > `filename` notation:

```
./a.out < image_in.ppm > image_out.ppm
```

A.6.7 Phase 4

Phase four is reading a PPM image file and outputting the width, the height, and the total number of pixels in the image. Obtaining this information requires knowledge of the image format, unlike the mere copying performed in the previous stage. Required knowledge: reading integers, addresses, character comparisons, nested loops, if statements, unreading data, error conditions, function return values, functions, boolean expressions, multiplication, checking for whitespace, stdlib.h, ctype.h.

1. As well as `stdio.h`, #include the `stdlib.h` and `ctype.h` files. The program will be reading and

المنارة للاستشارات
www.manaraa.com

altering a PPM file, and it must be confirmed immediately that the user indeed provided an image in PPM format as expected. If the user did not, an error message should be printed, and the main function should return a value indicating its failure. `stdlib.h` provides constants for indicating the success and failure of the main function. `ctype.h` declares functions for determining the type of data read in and will be useful in removing the commands and spaces in the file.

2. Declare a function for skipping over the whitespace and comments in the image file. The function will not return a value and should thus be marked as "void." The name may be anything.

3. Begin the function by reading in a character and determining whether it is a pound sign (#) or a whitespace character (e.g. a space, tab, carriage return, new line, etc.). If it is whitespace, it should be skipped. If it is a #, the remainder of the line is a comment, and all characters from # through the next newline character ('\n') should be skipped. If it is neither a comment nor a whitespace character, no more reading should take place, because the function is now reading useful information from the header, such as the width, the height, or the maximum value. Thus the function loops as long as the next character read is whitespace or part of a comment.

   Determining if a character is a whitespace character is simple. `ctype.h` declares a function for determining if a character is whitespace called "isspace()." It returns a non-zero value if the passed-in character is whitespace. "isspace()" expects the argument passed in to be of type `int` instead of a character. An integer usually is four bytes and can hold values in the range [-2,147,483,648 – 2,147,483,647], which more than covers the character range of [0,255]. Thus, characters in this function will be declared of type "int" merely for compatibility with the "isspace()" function.

   Determining whether a character is a "#" requires use of the comparison operator "==." Unlike the assignment operator ("="), the comparison operator returns a true value if the two things being compared are equal and false otherwise. A mistake programmers often make is to accidentally use "=" sign instead of "==". The compiler does not catch this error because the assignment, say `x=4`, returns the value assigned, in the case `4`, so `while (x=4)` is legal, if not terribly useful. The result is that instead of comparing the two values, the first value is assigned to the second. To prevent this common bug, if either value is constant, it should occur first. e.g. `3 = x` is not valid, because 3 cannot be assigned a new value. Thus, the compiler will catch the mistake and print an error message to correct the line to `3 == x`.

   To specify a single character in C, place it in single quotes. e.g. `'#'`, `'a'`, `'1'`, etc.) Thus, use the

المنارة للاستشارات
www.manaraa.com

following notation to determine if a character is a pound sign: `ch == '#'`

The skip function must continue as long as either condition is true. That is, if the character is whitespace *or* a pound sign, the loop must continue. To specify an "or" condition in C, use the double pipes: "||." The pipe is typically located above the backslash on the keyboard and appears to be a broken line. Two of them are the C boolean operator "or" e.g. to determine if x is a 3 or a 5, use the following statement: `3==x || 5==x`.

To begin the whitespace skipping function, first declare a variable of type `int` to hold the character currently being read, and read in the first character.

Next, create a while loop that continues as long as the character is either whitespace or a pound sign.

4. Create another loop inside the first loop to skip comments. A loop inside another loop is called a "nested loop." Each time the outer loop executes, the inner loop is executed and repeats until its condition fails. In this case, the inner loop will specify that as long as there are lines beginning with a pound sign, to skip them.

   If the character is a pound sign, the rest of the line must be skipped, using another nested while loop. This third loop will read characters until the character is the newline character '`\n`'. Thus, the loop continues as long as the character is *not* the newline character. The operator for determining if two items are not equal is the "!=" operator.

5. Create another loop inside the first loop of the function to continue reading characters as long as they are whitespace characters.

   This is the end of the outer loop as well, so it should be closed with a brace.

6. After the loops have completed, put back the last character read. The loop completes only after a non-comment, non-whitespace character has been read. This character is important data and should be put back into the file stream to allow it to be properly read later. The C function for putting a character back into the stream is "ungetc", and it accepts two arguments: the character to put back (whatever is stored in "ch") and the file stream in which to put it in ("stdin"). After putting back the character, close the function.

   Thus, the entire function is shown in Algorithm .4.

7. Update the main to have a "return value" of type "int." All functions in C may return a value. That is, when the function is completed, it returns a value to the place from which is was called. Main

www.manaraa.com

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void skip_comments_and_ws () {
   int ch = fgetc (stdin);
   while ('#' == ch || isspace(ch)) {
      /* remove command line, up to but not including the
          newline character */
      if ('#' == ch) {
         while ('\n' != ch) {
            ch = fgetc (stdin);
         }
      }
      /* remove whitespace, including the newline, at the end
         of a comment. */
      while (isspace(ch)) {
         ch = fgetc (stdin);
      }
   }
   ungetc (ch, stdin);
}
```

Algorithm .4: Complete function for skipping over comments and white space

functions traditionally return an integer value specifying whether the function was successful. A zero value indicates success and all other values indicate failure. To specify that the main function will return an integer, place the keyword "int" before the main function: See Algorithm .5.

8. Declare variables to hold the width and height and the magic number. The magic number is two characters: a 'P' and a '6'.

9. Read in the first two characters in the file and confirm that they are 'P' and '6'. If they are not, end the function early by returning an error code. The "return statement" in a function immediately ends a function and (if given a value) returns the specified value. For the error code, use the constant defined in `stdlib.h`: `EXIT_FAILURE`. (A constant is similar to a variable except that its value cannot be changed after its initial assignment. Constants are traditionally assigned names that are all capital letters.) Of course, it is possible to simply return an integer value, but the constant improves readability and guarantees a valid error code.

   To check whether the characters are correct, use an "if" statement. An if statement is similar in structure and behavior to a while loop. The difference is that an if statement is executed only once when the

المنارة للاستشارات
www.manaraa.com

```
int main () {
   int width, height;
   unsigned char letter, number;
   letter = fgetc (stdin);
   number = fgetc (stdin);

   if ('P' != letter || '6' != number) {
      printf ("Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   skip_comments_and_ws ();
   scanf ("%d", &width);
   skip_comments_and_ws ();
   scanf ("%d", &height);

   printf ("width: %d\n", width);
   printf ("height: %d\n", height);
   printf ("total pixels: %d\n", width*height);
   return EXIT_SUCCESS;
}
```

Algorithm .5: Main function that returns an integer

condition is true. There are three parts to an if statement: the keyword `if`, the condition, and the statement block to execute if the condition is true. The condition in this case is the first character is not 'P' OR the second character is not '6'. If the condition is true, an error message should be printed and the function should return an error code.

10. Call the function for skipping comments and whitespace, since there may be any number of comments and whitespace characters between the magic number and the width.

11. Read in the width as an integer. Until now, all data that has been read has been read as characters. An integer is represented in an ASCII file as multiple characters, one for each digit. The multi-digit integer should be read in and converted to one complete number stored in 4-byte, integer format. In C, the `scanf` function will read in data in multiple formats, depending on the format code provided. The format codes used by `scanf` are basically the same as the codes for `printf`. The format code for reading an integer is "`%d`" (decimal). The arguments to the function are text specifying the data type to read in and then the address of the variable to store the read data in. The data type of the variable must match the format code specified, or the results could be incorrect. The address of the variable may be obtained using the & operator. Similar to printf, scanf can read multiple items in each call, but only one

www.manaraa.com

integer needs to be read at this time.

12. Skip the comments and whitespace after the width and read the height.

13. Print out the width and the height using printf. To make the output more readable, put some text before the format code specifying what data is being displayed.

14. Print out the total number of pixels. The total number of pixels is the width multiplied by the height. To multiple two numbers in C, use the * operator.

15. At the end of the main function, return the constant indicating that the program was a success.

A.6.8 Phase 5

Phase five is reading a PPM image file, printing it to standard out, and outputting the width, the height, and the total number of pixels to standard error. Since this phase does not have a great deal of new information, it is a good time to refactor the code to create a better organization and a header file. Required knowledge: outputting to standard error, addresses, pointers, header files.

1. Create a header file to hold all the preprocessor directives (include statements) and a forward declaration for the `skip_comments_and_ws` function. (The header file should be named after the name of the C file with the extension `.h`.) Forward declarations are similar to the first line of a function, except that the parameters do not need to be named and, instead of a block of statements, ends with a semicolon. Forward declarations provide information to the compiler about the function before the actual code is compiled, meaning that the function may exist in a different file or after that point at which it is called. If a function has no parameters, it is best to specify that the parameters are "void" in order to prevent the compiler from assuming that the forward declaration merely chose to not specify parameters. Forward declarations may be delayed to give students practice defining functions before use.

2. Add an include statement at the top of the C file to include the header file.

3. Create a "read_header" function to read the entire header, and return the width, the height, and whether the function was successful. Since functions can return only one value, addresses (via pointers) must be used instead to return additional information. It has already been demonstrated that scanf uses the address of a variable to store information. Similarly, the `read_header` function may accept the

www.manaraa.com

```
#include "print_img.h"
int read_header (int *width, int *height) {
   int maxVal;
   unsigned char c, letter;
   c = fgetc (stdin);
   letter = fgetc (stdin);
   if (c != 'P' || letter != '6') {
      printf ("Incorrect Magic Number\n");
      return 0;
   }
   skip_comments_spaces ();
   scanf ("%d", width);
   skip_comments_spaces ();
   scanf ("%d", height);
   skip_comments_spaces ();
   scanf ("%d%c", &maxVal, &c);
   return 1;
}
```

Algorithm .6: The complete function for reading the ASCII header

addresses of width and height variables and store the values read into those, e.g., a call to `read_header` would be the following: `read_header (&width, &height);`

To accept addresses as parameters, the `read_header` function must specify the data types of the variables to be integer pointers. A pointer is an address in memory that holds a particular type of data. For example, an integer pointer is the address of an integer in memory. The four bytes of data at that address will be interpreted as an integer. To declare a variable to be a pointer, after the data type, use an asterisk (*) before each variable that will be a pointer, e.g., `int *i, *j;`.

The integer returned will specify whether the read was successful. The function will put the width and height in the address locations specified so that the main function will have access to them. The width and height will be read in by scanf, which uses addresses. Since width and height are already addresses, there is no need for & before them.

The function will call the skipping function as needed to skip comments and whitespace, and it will read all the information from the image except the image data. If the file does not begin with the magic number, `read_header` will return a false value to indicate that reading failed. The maximum value will be assumed to be 255 and does not need to be returned. Only one character should occur after the maximum value, and it will be "eaten" as well. See Algorithm .6.

المنارة للاستشارات
www.manaraa.com

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void skip_comments_and_ws (void);
int read_header (int *, int *);
```

Algorithm .7: Header file

```
int main () {
   int width, height;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P6\n%d %d\n255\n", width, height);
   for (i=0; i < width*height*3; ++i) {
      c = fgetc (stdin);
      printf ("%c", c);
   }
   fprintf (stderr, "width: %d\n", width);
   fprintf (stderr, "height: %d\n", height);
   fprintf (stderr, "total pixels: %d\n", width*height);
   return EXIT_SUCCESS;
}
```

Algorithm .8: Invocation of the header-reading function and error testing

4. Add a forward declaration for `read_header` function in "print_img.h." See Algorithm .7.

5. Update the main function to call read_header, passing it the address of locally-created width and height variables. If the header read is unsuccessful, an error message should be printed and the function should be returned.

   Error messages should not be sent to standard out because 1) they will end up in the same file to which standard out is directed and 2) there is a special output stream, usually used for error messages, called "standard error." Data printed to standard error will not be included with standard out (but it can be redirected as well). To output to standard out, use the `fprintf` function. The `fprintf` function accepts the file streams `stderr` and `stdout`, as well as user-created streams. It is followed by the usual arguments used for `printf`. See Algorithm .8.

6. Update the main to read the $width \times height \times 3$ bytes of image and output them to standard out, then

المنارة للاستشارات
www.manaraa.com

print the width, height, and total pixels to standard error. Add text with the output of the width, height, and total pixels to identify each element.

7. Compile and run the program, redirecting the input and output appropriately. The image should be copied to the new location and the width, height, and total number of pixels should be displayed on the screen.

A.6.9 Phase 6

Phase six is reading a PPM image file and modifying to the colors. This phase leads to the final goal of modifying an image's color scheme to match another image. The phase provides practice modifying image data and may take many forms, including changing one channel's value to a fixed value, lightening or darkening all values, converting the image to grayscale, inserting scan lines, fading the colors, converting to monochrome, etc. This is a phase where students can be creative. Required knowledge: information regarding the selected file modification. Channel modification: none; Grayscale: floating point; Scan lines: nested "for" loops, modular arithmetic; conversion to monochrome: chrominance and luminance formulas, capping values, etc.

Update the main to make the appropriate modification to the pixels read, using the previous phase as the starting point for each version.

- Changing a channel to a fixed value.
    1. Choose a channel to modify and a way to modify it. For this example, the blue channel will be changed to zero.
    2. Update the main function to read the three bytes representing the pixel and modify the appropriate channel appropriately. See Algorithm .9.

       The resulting image should clearly be lacking the blue channel.
- Increasing/decreasing all channels by a fixed value is similar to changing an individual channel. Be sure to check for overflow (values over 255) and underflow (values below 0).
- Conversion to grayscale
    1. Change the output image's magic number to be "P5." P5 is the specification for grayscale PPM files. Instead of three bytes of data for each pixel, each pixel is represented by one byte of data,

المنارة للاستشارات
www.manaraa.com

```
int main () {
   int width, height, i;
   unsigned char red, green, blue;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P6\n%d %d\n255\n", width, height);
   for (i=0; i < width * height; ++i) {
      red   = fgetc (stdin);
      green = fgetc (stdin);
      blue  = fgetc (stdin);
      printf ("%c%c%c", red, green, 0);
   }
   return EXIT_SUCCESS;
}
```

Algorithm .9: Modification of the image to exclude blue

which usually contains the "luminance" value of the pixel. Luminance describes the amount of brightness of the pixel: 0 is black and 255 is white. See Algorithm .10.

2. Update the function to read in three bytes (representing one pixel) at a time, convert them to luminance, and output the result. In general, conversion of an RGB color specification to luminance requires knowledge of the intended display device, but a reasonable choice is to assume the NTSC standard display, for which the weighting should be 30% red, 59% green, and 11% blue. Numbers with decimal points are called "floating point values." Floating point values are more difficult to represent than integers, because of precision issues. Therefore, floating point variables are used only when necessary. ".3", ".59", and ".11" will be handled as "doubles" in C. "Doubles" are double-precision, floating-point numbers and require twice as many bytes as the smaller floating point representation, "float." Multiplying the unsigned characters holding the bytes of image data by doubles will produce an answer in double format. Since the data will be printed in unsigned character format, the result will be converted back to an `unsigned char` by the assignment to "lum." All the information after the decimal point will be truncated. e.g. `unsigned char c = 1.9999;` results in 1. Addition of 0.5 followed by truncation is equivalent to rounding, and that is used in the displayed code.

   The output should be a grayscaled image.

- Darkened scan lines (like a TV with a fuzzy signal)

المنارة للاستشارات
www.manaraa.com

```
#include "grayscale.h"
int main () {
   int width, height, i;
   unsigned char red, green, blue;
   unsigned char lum;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P5\n%d %d\n255\n", width, height);
   for (i=0; i < width * height; ++i) {
      red   = fgetc (stdin);
      green = fgetc (stdin);
      blue  = fgetc (stdin);
      lum = red * .3 + green * .59 + blue * .11 + .5;
      printf ("%c", lum);
   }
   return EXIT_SUCCESS;
}
```

Algorithm .10: Image grayscaling

1. In the main function, create variables specifying how much of a gap to have between each darkened line and how wide each darkened line should be. (These should probably be constants.) The line width is the number of pixels wide each darkened line will be, and the gap is the number of pixels between the start of successive darkened lines. Additionally, create three loop counter variables. See Algorithm .11.
2. Update the loop for reading the image data to be three nested for loops: the first loop goes through each row, the second through each pixel, and the third through each channel of each pixel. This step provides a loop counter variable for determining which row is currently being read.
3. The method of determining if a row should have a scan line is to 1) find the remainder after dividing of `i` by the gap and 2) determine if it is a value less than the line gap. For example, if the gap is 4, for rows 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10, the remainders after division are 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2 respectively. Then, if these values are compared to a line width of 2, remainder values of 0 and 1 will be given darkened scan lines. That is, lines 0, 1, 4, 5, 8, and 9. To calculate the remainder, use the modulo operator `%`. $A\%B$ the remainder after dividing integer A by B. After multiplication by a double, the result must be cast back to an unsigned character. (See grayscaling.)

www.manaraa.com

```
int main () {
   int width, height, i, j, k;
   int gap = 8, line_width = 2;
   unsigned char channel;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P6\n%d %d\n255\n", width, height);
   for (i=0; i < height; ++i) {
      for (j=0; j < width; ++j) {
         for (k=0; k < 3; ++k) {
            channel = fgetc (stdin);
            if (i%gap  < line_width) {
               channel = (unsigned char)(channel * .9);
            }
            printf ("%c", channel);
         }
      }
   }
   return EXIT_SUCCESS;
}
```

Algorithm .11: Adding scan lines

- Color fade. The concept behind fading is reducing the amount of color in the image toward grayscale. The image is not darkened, merely faded. The method of fading is to add a weighted grayscale amount to the image. For example, if the image will be faded 50%, 50% of each color channel will be the color value and 50% will be the grayscale value.

  1. Create the necessary variables to hold the channel values, the grayscale value, and the amount to fade. The example is 50%.
  2. Read in the pixels, one at a time, and compute the luminance for each pixel. (See previous description of luminance computation.)
  3. Combine the current value of each channel with the luminance value. Use the specified *amt* to weight the channel value and $1 - amt$ for the luminance.
  4. Output the resulting channel values. The output result should have faded colors. See Algorithm .12.

- Monochrome conversion: converting the image into one with a fixed chrominance and varying luminance. Basically, instead of gray, the base color may be any color. The color value separate from its

المنارة للاستشارات
www.manaraa.com

```
int main () {
   int width, height, i;
   unsigned char red, green, blue, lum;
   float amt = .5;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P6\n%d %d\n255\n", width, height);

   for (i=0; i < width * height; i++) {
      red   = fgetc (stdin);
      green = fgetc (stdin);
      blue  = fgetc (stdin);

      /* compute the grayscale value */
      lum = red * .3 + green * .59 + blue * .11 + .5;
      red   = red   * amt + lum * (1.0-amt);
      green = green * amt + lum * (1.0-amt);
      blue  = blue  * amt + lum * (1.0-amt);

      printf ("%c%c%c", red, green, blue);
   }
   return EXIT_SUCCESS;
}
```

Algorithm .12: Fade program

المنارة للاستشارات
www.manaraa.com

brightness is called chrominance. The output image will be in the 255 possible shades of that color. In order to apply the color in this way, it must be converted to a format that separates chrominance from brightness.

Television signals are transmitted in chrominance and luminance (technically gamma corrected luminance. luminance ("Y") represents the brightness of the image (between black and white) and is the only part used by black and white televisions. Chrominance is represented by two values, sometimes called "Cb" and "Cr." Cb is $blue - Y$, and Cr is $red - Y$. To compute Cb and Cr from RGB values, use the following formulas:

$$Cb = -.147 \times red - .289 \times green + .436 \times blue \tag{1}$$

$$Cr = .615 \times red - .515 \times green - .1 \times blue \tag{2}$$

Once the chrominance of a color is determined, the RGB value of the color with the same chrominance but different luminance(Y) may be obtained using the following formulas:

$$red = Y + 1.14 \times Cr \tag{3}$$

$$green = Y - .395 \times Cb - .581 \times Cr \tag{4}$$

$$blue = Y + 2.032 \times Cb \tag{5}$$

1. Choose a color value to extract Chrominance from. The example is Clemson orange (255, 99, 0). See Algorithm .13.
2. Create variables for holding the width, height, loop counter, the current pixel's RGB values, the current pixel's luminance, temporary integer RGB values with which to perform mathematics, and floating point variables for holding `Cr` and `Cb`.
3. Calculate the chrominance of the chosen color.
4. Read in the RGB values of one pixel at a time and compute the luminance.

www.manaraa.com

```
int main () {
   unsigned char mono_r = 255;
   unsigned char mono_g =  99;
   unsigned char mono_b =   0;
   int width, height, i;
   int r, g, b;
   float cb, cr;
   cb = -.147 * mono_r - .289 * mono_g + .436 * mono_b;
   cr =  .615 * mono_r - .515 * mono_g - .1   * mono_b;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   printf ("P6\n%d %d\n255\n", width, height);
   for (i=0; i < width*height; i++) {
      r = fgetc (stdin);
      g = fgetc (stdin);
      b = fgetc (stdin);
      lum = .3 * r + .59 * g + .11 * b;

      r = lum + 1.14 * cr + .5;
      g = lum - .395 * cb - .581 * cr + .5;
      b = lum + 2.032 * cb + .5;

      if (r > 255) r= 255;
      if (r <   0) r=   0;
      if (g > 255) g= 255;
      if (g <   0) g=   0;
      if (b > 255) b= 255;
      if (b <   0) b=   0;
      printf ("%c%c%c", r, g, b);
   }
   return EXIT_SUCCESS;
}
```

Algorithm .13: Monochrome program

المنارة للاستشارات
www.manaraa.com

```
unsigned char  in[1280*960*3];
```

Algorithm .14: Declaration of an array of unsigned characters to hold image data

5. Calculate new RGB values for the pixel based on the specified chrominance and the newly calculated luminance.
6. Cap the values of the newly-calculated RGB values to be in the range [0,255]. It is possible that the calculated values have exceeded the range of a byte. (Note that these if statements could be improved as if-else statements. It is up to the instructor as to whether to introduce the concept.)
7. Assign the new values to the unsigned characters and output them using printf.

   The resulting image should be in shades of Clemson Orange.

A.6.10 Phase 7

Phase seven is reading a PPM image file into an array and making a modification. The storage of the entire image at one time is necessary for the final goal of reading in two images entirely and manipulating one based on the other. Modifications to the image may now depend on knowledge of more than one pixel at a time. Such modifications include resizing, tiling, flip, half-toning, rotation, blur, sharpen, etc. This is another opportunity for creativity. Required knowledge: arrays, fread, fwrite.

1. Determine the size of the image to manipulate. The size may be found using a viewer or by using the Unix "head" command to look at the first few lines of the input PPM. e.g. `head -3 in.ppm`. The size of the example image is 1280 by 960.

2. Create an array of unsigned characters in the main to hold all the image data. An "array" is a contiguous block of memory with a single name that can hold multiple values of the same data type. An array is declared with a data type and a length, or number of elements of that data type. The notation for declaring an array is "datatype arrayname [number of elements];" e.g. `int array [10];` The block of memory allocated by such an array declaration is the size of the data type (e.g. four bytes for int, one byte for char, eight bytes for double) times the number of elements. In the array for the example image must hold $1280 \times 960 \times 3$ bytes. The multiplication by 3 is necessary because each pixel is represented by 3 bytes. See Algorithm .14.

3. Read data into the array using the "fread" function. The "fread" function reads in the specified number

المنارة للاستشارات
www.manaraa.com

```
fread (in, 1, 1280*960*3, stdin);
```

Algorithm .15: Reading of the entire block of image data

of elements of the specified size from a specified file stream into an array. The benefit of "fread" is that all image data may be read in one command instead of hundreds or thousands. Individually reading bytes of data is much slower than reading the entire block at once. The notation for fread is "fread (location at which to store data, number of bytes for each element, number of elements, input file stream);" Note that the name of the array is equivalent to the address of its first storage location. So, `&in[0]` could have been used instead of `in`. See Algorithm .15.

4. Apply the appropriate modification. Examples are rotate 90 degrees and sharpen.

   - Rotate 90 degrees. The image will be rotated in the clockwise direction 90 degrees. The height will now become the width, and vice versa.

     (a) Create the main with multiple loop counter variables, the width and height, the new width and height (after rotation), an array to hold the input image data, and an array to hold the modified output data. See Algorithm .16.

     (b) After reading in the data, swap the width and the height for the output image.

     (c) Create three nested for loops: one for the rows in the new image, one for the columns, and one for the 3 bytes in each pixel.

     (d) Place in the current location the appropriate byte of data. There are two key parts to handle: 1) getting the current location in the output array and 2) getting the appropriate byte. First, the red byte of the first pixel in the upper left-hand corner is at array element zero. In C, array elements begin with element 0. (This counting scheme is based on the fact that elements are located by adding the element number to the address of the array, and the first element is at the beginning and needs nothing added.) To access element 0, use the notation `in[0]`. The green byte is at array element 1, i.e. `in[1]`, and the blue at `in[2]`. The red byte of the second pixel on the first row is at `in[3]`. The red byte of the first pixel on the second row is at `in[width*3]`. The reason for the multiplication is that the second row occurs after all the first row data is complete, and there are "width" number of pixels with 3 bytes of data. The first red byte on the ith row is at `in[i*width*3]`. Finally, the blue byte on the jth pixel

المنارة للاستشارات
www.manaraa.com

```
int main () {
   int width, height, new_w, new_h, i, j, k;
   unsigned char  in[1280*960*3];
   unsigned char out[960*1280*3];

   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   if (width != 1280 || height != 960) {
      fprintf (stderr, "Error: file is not the correct size.\n");
      return EXIT_FAILURE;
   }
   fread (in, 1, width*height*3, stdin);
   new_w = height;
   new_h = width;
   printf ("P6\n%d %d\n255\n", new_w, new_h);
   for (i=0; i < new_h; i++) {
      for (j=0; j < new_w; j++) {
         for (k=0; k < 3; k++) {
            out[(i*new_w+j)*3+k]=in[((height-j-1)*width+i)*3+k];
         }
      }
   }
   // fwrite included in later code sample
```

Algorithm .16: Rotate 90 degrees

المنارة للاستشارات
www.manaraa.com

on the ith row is at `in[(i*width+j)*3+2]`: each pixel is 3 bytes away from the next (thus the multiplication by 3) and the blue channel is the 3rd byte of the pixel, thus the addition of 2. In the example, the current location in the output data is specified as row i, column j, channel k which is at `out[(i*new_w+j)*3 + k]`.

With the location to copy the data to in hand, where should the data be drawn from in order to result in the rotation of the image? Consider an image with a width of 3 and height of 4. The pixel layout is as follows:

$$\begin{array}{lll} 0_r0_g0_b & 1_r1_g1_b & 2_r2_g2_b \\ 3_r3_g3_b & 4_r4_g4_b & 5_r5_g5_b \\ 6_r6_g6_b & 7_r7_g7_b & 8_r8_g8_b \\ 9_r9_g9_b & 10_r10_g10_b & 11_r11_g11_b \end{array}$$

If the image is rotated by 90 degrees, the layout of the numbered pixels will be as follows:

$$\begin{array}{llll} 9_r9_g9_b & 6_r6_g6_b & 3_r3_g3_b & 0_r0_g0_b \\ 10_r10_g10_b & 7_r7_g7_b & 4_r4_g4_b & 1_r1_g1_b \\ 11_r11_g11_b & 8_r8_g8_b & 5_r5_g5_b & 2_r2_g2_b \end{array}$$

Thus, the pixel in row $height - 1 - j$, column i, channel k maps to the new image's row i, column j, channel k.

- Sharpen image. Sharpening is done by the application of a convolution filter, which is simply a re-weighting of each pixel to distinguish it from surrounding pixels. To compute the final value of a sharpened pixel, floating point values in a fixed 3 by 3 filter grid are multiplied by the corresponding pixel values with the target pixel in the middle. The products are then summed to reach the final pixel value. The filter must be applied to all 3 channels of each pixel. See Algorithm .17.

5. After modification, output the modified image data to standard out using the "fwrite" function. The "fwrite" function outputs the specified number of elements of the specified size from the specified array to the specified file stream. Similar to fread, using fwrite to output the data in one block is more efficient than writing it byte at a time. The notation for fread is "fread (output data, number of bytes for each element, number of elements, output file stream);" See Algorithm .18.

المنارة للاستشارات
www.manaraa.com

```c
float filter [3][3] = { {  0/3.0, -2/3.0,  0/3.0},
                        { -2/3.0, 11/3.0, -2/3.0},
                        {  0/3.0, -2/3.0,  0/3.0} };

/* After the image data is read into image */
for (i=0; i < height; ++i) {
   for (j=0; j < width; ++j) {
      for (k=0; k < 3; ++k) {

         /* multiply each surrounding pixel by the
            corresponding filter value and sum up the result
            to find the sharpened value */
         result = 0.0;
         for (m=0; m < 3; ++m){
            for (n=0; n < 3; ++n){

               /* the row should be offset by m. If the row
                  would then be out of bounds, the modding and
                  addition of height will wrap it around */
               row = (i + m - 1 + height)%height;

               /* modding by width with column will similarly protect
                  from out of bounds. */
               col = (j + n - 1 + width)%width;
               result+=filter[m][n] *
                       image[(row*width+col)*3+k];
            }
         }

         /* cap the value */
         if (result > 255) result = 255;
         if (result < 0) result = 0;
         newImage[(i*width+j)*3+k] = (unsigned char) result;
      }
   }
}
```

Algorithm .17: Applying the sharpen filter

```c
fwrite (out, 1, width*height*3, stdout);
```

Algorithm .18: Output of the entire block of image data

www.manaraa.com

```
int main () {
   unsigned char *image, *newImage;
   int width, height;
   if (!read_header (&width, &height)) {
      fprintf (stderr, "Error: input file is not PPM format.\n");
      return EXIT_FAILURE;
   }
   image = (unsigned char *) malloc (width * height * 3);
   newImage = (unsigned char *) malloc (width * height * 3);
   fread (image, height * width * 3, 1, stdin);

   /* modify data and place in newImage array */

   fwrite (newImage, width*height, 3, stdout);
   free (newImage);
   free (image);
   return EXIT_SUCCESS;
}
```

Algorithm .19: Dynamic memory allocation

A.6.11 Phase 8

Phase eight is reading a PPM image file of any size using dynamic memory allocation. The size of the image is unknown at runtime and is read from the header of the image file. The memory needed is then allocated dynamically. Required knowledge: dynamic memory allocation.

1. Declare two unsigned character pointers for the input and output image data. These pointers will later point to the dynamically allocated blocks of memory to hold image data.

2. Get the width and height of the image header.

3. Dynamically allocate $width \times height \times 3$ bytes for input and output data.

4. Read in the data and place the modified data in the output array.

5. Output the array and free the allocated memory. See Algorithm .19.

A.6.12 Phase 9

Phase nine is reading a PPM image file, converting it to CIELAB colorspace, modifying the colors, and printing it back out. The modification can be anything from modifying one component to produce an interesting effect or a color balancing algorithm. The matrix multiplication for this phase may be covered in

المنارة للاستشارات
www.manaraa.com

```
void multiply3by3 (float m1[3][3], float m2[3][3], float result[3][3]) {
   int i, j, k;
   for (i=0; i < 3; ++i) {
      for (j=0; j < 3; ++j) {
         result[i][j] = 0;
         for (k=0; k < 3; ++k) {
            result[i][j] += m1[i][k] * m2[k][j];
         }
      }
   }
}
```

Algorithm .20: 3 by 3 multiplication

the laboratory setting or provided to the students. Required knowledge: matrix multiplication, knowledge of RGB to CIELAB format.

1. Create a function to multiply two 3 by 3 matrices together. This function will be used to compute the conversion matrices. See Algorithm .20.

2. Create functions to multiply an array of unsigned characters by a conversion matrix of floats producing floats, to multiply an array of floats by a conversion matrix producing an array of unsigned characters, and to multiply an array of floats by a conversion matrix producing an array of floats. These function will be used to convert to and from RGB, LMS and CIELAB formats. (LMS is a color space represented by the response of the three types of cones of the human eye, named after their sensitivity at long, medium and short wavelengths. Because RGB is relative color space (e.g., 255 is full intensity for a channel, but full intensity is not defined), RGB is converted to LMS before conversion to the CIELAB format which is a more visually uniform format.) The parameters are 2D arrays, but they can be left as flat arrays. To use flat (1D) arrays as 2D arrays, cast the pointer to be a 2D array. The number of elements in each row much be constant. See Algorithms .21 and .22.

3. Create functions to convert to and from RGB and LMS. See Algorithm .23.

4. Create functions to convert to and from LMS and CIELAB. See Algorithm .24.

5. Create RGB/CIELAB conversion functions that invoke existing conversion functions. See Algorithm .25.

6. Create utility functions to read and output PPM images.

المنارة للاستشارات
www.manaraa.com

```
void matrix_mult_char2float (float matrix[3][3],
       unsigned char (*in)[3], float (*out)[3], int num_pixels){
   int i, j, k;
   float result;
   for (i=0; i < num_pixels; ++i) {
      for (j=0; j < 3; ++j) {
         result = 0;
         for (k=0; k < 3; ++k)
            result += in[i][k] * matrix[k][j];
         out[i][j] = result;
      }
   }
}
void matrix_mult_float2char (float matrix[3][3], float (*in)[3],
                   unsigned char (*out)[3], int num_pixels){
   int i, j, k;
   float result;
   for (i=0; i < num_pixels; ++i) {
      for (j=0; j < 3; ++j) {
         result = .5; /* add .5 for rounding  */
         for (k=0; k < 3; ++k)
            result += in[i][k] * matrix[k][j];
         if (result > 255) result = 255;
         if (result < 0) result = 0;
         out[i][j] = result;
      }
   }
}
```

Algorithm .21: Character and float conversion

المنارة للاستشارات
www.manaraa.com

```
void matrix_mult_float2float (float matrix[3][3],
                              float (*array)[3], int num_pixels){
   int i, j, k;
   float result;
   float *out = (float *) malloc (num_pixels*3* sizeof(float));
   for (i=0; i < num_pixels; ++i) {
      for (j=0; j < 3; ++j) {
         result = 0;
         for (k=0; k < 3; ++k)
            result += array[i][k] * matrix[k][j];
         out[i*3+j] = result;
      }
   }
   for (i=0; i < num_pixels; ++i) {
      for (j=0; j < 3; ++j) {
         array[i][j] = out [i*3+j];
      }
   }
   free (out);
}
```

Algorithm .22: Character and float conversion, continued.

```
void rgb_to_lms(unsigned char *rgbArray, float *labArray, int num_pixels
   ){
   float matrix [3][3];
   float m1 [3][3] = { {0.3897, 0.6890, -.0787},
                       {-.2298, 1.1834, 0.0464},
                       {0.0000, 0.0000, 1.0000} };
   float m2 [3][3] = { {.5141, .3239, .1604},
                       {.2651, .6702, .0641},
                       {.0241, .1228, .8444} };

   multiply3by3 (m1, m2, matrix);
   matrix_mult_char2float (matrix, (unsigned char (*)[3])rgbArray,
                          (float (*)[3])labArray, num_pixels);
}

void lms2rgb (float *lab, unsigned char *rgb, int num_pixels) {
   float matrix [3][3] = { {4.4679, -3.5873,  0.1193},
                           {-1.2186, 2.3809, -0.1624},
                           {0.0497, -0.2439,  1.2045} };
   matrix_mult_float2char (matrix, (float (*)[3])lab,
                          (unsigned char (*)[3])rgb, num_pixels);
}
```

Algorithm .23: RGB to LMS and LMS to RGB

المنارة للاستشارات
www.manaraa.com

```
void lms2lab (float *labArray, int num_pixels) {
   float matrix [3][3];
   float m1 [3][3] = { {sqrt(3)/3, 0, 0},
                       {0, sqrt(6)/6, 0},
                       {0, 0, sqrt(2)/2} };
   float m2 [3][3] = { {1, 1,  1},
                       {1, 1, -2},
                       {1, -1, 0} };

   multiply3by3 (m1, m2, matrix);
   matrix_mult_float2float (matrix, (float (*)[3])labArray, num_pixels);
}

void lab_to_lms (float *labArray, int num_pixels) {
   float matrix [3][3];
   float m1 [3][3] = { {1, 1,  1},
                       {1, 1, -1},
                       {1, -2, 0} };
   float m2 [3][3] = { {sqrt(3)/3, 0, 0},
                       {0, sqrt(6)/6, 0},
                       {0, 0, sqrt(2)/2} };

   multiply3by3 (m1, m2, matrix);
   matrix_mult_float2float (matrix, (float (*)[3])labArray, num_pixels);
}
```

Algorithm .24: LMS to CIELAB and CIELAB to LMS

```
void rgb_to_lab(unsigned char *rgbArray, float *labArray, int num_pixels
   ){
   rgb_to_lms (rgbArray, labArray, num_pixels);
   lms2lab (labArray, num_pixels);
}

void lab2rgb(unsigned char *rgbArray, float *labArray, int num_pixels){
   lab_to_lms (labArray, num_pixels);
   lms2rgb (labArray, rgbArray, num_pixels);
}
```

Algorithm .25: RGB to CIELAB and CIELAB to RGB

المنارة للاستشارات
www.manaraa.com

```
int getImage (unsigned char **data, int *width, int *height) {
   unsigned char c;
   int maxVal;
   c = fgetc (stdin);

   /* toupper converts the character to uppercase */
   if (toupper(c) != 'P' || (c=fgetc(stdin)) != '6') {
      fprintf (stderr, "Incorrect Magic Number\n");
      return 0;
   }
   skip_comments_spaces ();
   fscanf (stdin, "%d", width);

   skip_comments_spaces ();
   fscanf (stdin, "%d", height);

   skip_comments_spaces ();
   fscanf (stdin, "%d", &maxVal);
   fgetc (stdin);
   *data = (unsigned char *) malloc ((*width) * (*height) * 3);
   fread (*data, (*width) * (*height) * 3, 1, stdin);
   return 1;
}

void skip_comments_spaces () {
   unsigned char c;
   c = fgetc (stdin);
   while (c == '#' || isspace(c)) {
      if (c == '#') {
         while (c != '\n') {
            c = fgetc (stdin);
         }
      }
      while (isspace(c)) {
         c = fgetc (stdin);
      }
   }
   ungetc (c, stdin);
}
void outputPPM (unsigned char *data, int width, int height) {
   fprintf (stdout, "P6\n%d %d\n255\n", width, height);
   fwrite (data, width * height * 3, 1, stdout);
}
```

Algorithm .26: Image reading utilities

المنارة للاستشارات
www.manaraa.com

```
int main () {
   int width, height, i;
   unsigned char *image;
   float * lab;

   if (!getImage (&image, &width, &height)) {
      fprintf (stderr, "Error reading \n");
      return EXIT_FAILURE;
   }
   lab = (float *) malloc(width*height*3*sizeof(float));
   rgb_to_lab (image, lab, width * height);
   for (i=0; i < width*height; ++i) {
      lab[i*3+1] += 20;
   }
   lab2rgb (image, lab, width*height);
   outputPPM (image, width, height);
   return EXIT_SUCCESS;
}
```

Algorithm .27: CIELAB conversion main function

7. Create main method to perform conversion, modify the colors and output. See Algorithm .27.

A.6.13 Phase 10

Phase ten is reading a PPM image file specified on the command-line, converting it to CIELAB format to modify the colors, and printing it back out. Use of command-line arguments is necessary for the final phase, where two images must be specified as input. Required knowledge: Command-line arguments, strings.

1. Update the main function to accept parameters to hold the command-line arguments. The main function, like all functions may specify parameters in the parentheses after its name. The parameters should be specified by a data type and a name assigned to it. When the function is called, the appropriate type and number of arguments must be supplied. “Arguments” are the actual values used when calling the function. “Parameters” are the names for the values used inside the function.

   The first command-line argument (implicitly) passed to the main function is a count of the number of arguments in the command line. The second argument is an array of the strings provided on the command line, beginning with the name of the program, in the following format: `./aout arg1 arg2 arg3` and so on. These arguments are handled as text or what is called in programming terminology “strings.” A “C string” is a sequence of characters with a value of zero (called “NULL”) marking the

المنارة للاستشارات
www.manaraa.com

```
int main (int argc, char *argv[]) {
   int width, height, i;
   unsigned char *image;
   float * lab;
   FILE *in;
   if (argc < 2) {
      fprintf (stderr, "Usage: %s filename\n", argv[0]);
      return EXIT_FAILURE;
   }
   in = fopen (argv[1], "r");
   if (!getImage (in, &image, &width, &height)) {
      fprintf (stderr, "Error reading \n");
      return EXIT_FAILURE;
   }
   lab = (float *) malloc(width * height * 3 * sizeof(float));
   rgb_to_lab (image, lab, width * height);
   for (i=0; i < width*height; ++i) {
      lab[i*3+1] += 20;
   }
   lab2rgb (image, lab, width*height);
   outputPPM (image, width, height);
   return EXIT_SUCCESS;
}
```

Algorithm .28: Reading file name from command line

end of the string. The first string passed in is the name of the executable program. Inside the program, this can be found at `argv[0]` which holds the location of the first character in this string name. The first user-supplied command-line argument is at array location one, `argv[0]`. See Algorithm .28.

2. Update the error message to use the command-line arguments.

3. Open the file passed in on the command line using the “fopen” command. The file should be opened in read mode using the “r” identifier.

4. Pass the file handle (stream identifier) to the image reading functions. See Algorithm .29.

A.6.14 Phase 11

Phase eleven is reading in two PPM image files, converting them to CIELAB format, computing the means and standard deviations on each file, adjusting the values of the first image to the values of the second based on the paper [60], and printing out the resulting image. Required knowledge: computation of mean, standard deviation, math functions, and structures.

www.manaraa.com

```
int getImage (FILE *in, unsigned char **data, int *width, int *height) {
   unsigned char c;
   int maxVal;
   /* read P6 */
   c = fgetc (in);
   if (toupper(c) != 'P' || (c=fgetc(in)) != '6') {
      fprintf (stderr, "Incorrect Magic Number\n");
      return 0;
   }
   skip_comments_spaces (in);
   fscanf (in, "%d", width);
   skip_comments_spaces (in);
   fscanf (in, "%d", height);
   skip_comments_spaces (in);
   fscanf (in, "%d", &maxVal);
   fgetc (in);
   *data = (unsigned char *) malloc ((*width) * (*height) * 3);
   fread (*data, (*width) * (*height) * 3, 1, in);
   return 1;
}

void skip_comments_spaces (FILE *in) {
   unsigned char c;
   c = fgetc (in);
   while (c == '#' || isspace(c)) {
      if (c == '#') {
         while (c != '\n') {
            c = fgetc (in);
         }
      }
      while (isspace(c)) {
         c = fgetc (in);
      }
   }
   ungetc (c, in);
}
```

Algorithm .29: File reading with file handle

www.manaraa.com

```
void logarray (float *fltarray, int num_pixels) {
   int i;

   for (i=0; i < num_pixels * 3; ++i) {
      fltarray[i] = log10 (fltarray[i]);
      if (!finite(fltarray[i])) fltarray[i] = 0;
   }
}
void exparray (float *fltarray, int num_pixels) {
   int i;
   for (i=0; i < num_pixels *3; ++i) {
      fltarray[i] = pow (10, fltarray[i]);
   }
}
```

Algorithm .30: Log (base 10) and Power of Ten functions

```
void rgb_to_lab (unsigned char *rgbArray, float *fltarray, int
    num_pixels) {
   rgb_to_lms (rgbArray, fltarray, num_pixels);
   logarray  (fltarray, num_pixels);
   lms2lab (fltarray, num_pixels);
}
void lab2rgb (unsigned char *rgbArray, float *fltarray, int num_pixels)
    {
   lab_to_lms (fltarray, num_pixels);
   exparray (fltarray, num_pixels);
   lms2rgb (fltarray, rgbArray, num_pixels);
}
```

Algorithm .31: Conversion with minimized skewing

1. Create functions to compute and store the logarithms and exponentials of arrays of floats. The data values are scaled logarithmically to minimize skewing, and after modification they must be converted back. See Algorithm .30.

2. Invoke the logarithm and power functions appropriately. See Algorithm .31.

3. Create a structure to hold all the information about a file, including its name; its RGB, LMS, and CIELAB values; its width and height; and its CIELAB means and standard deviations. A structure is an aggregate, user-defined type that may hold multiple variables of multiple types. Since it is a (user-defined) type, any number of variables of that structure type may be created. See Algorithm .32.

4. Create functions to compute the means and standard deviations of CIELAB L, A, and B values. Mean

www.manaraa.com

```
struct fileinfo {
   char *name;
   unsigned char *rgb;
   float *lms;
   float *lab;
   int width, height;
   float mean[3], std[3];
};
```

Algorithm .32: File information structure

is a simple average. Standard deviation is defined by the following formula:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{6}$$

The standard deviation may be computed with the following steps: 1) Subtract the mean from each value. 2) Square the value. 3) Compute the sum of the squares. 4) Divide each square sum by the number of values. 5) Take the square root the value. See Algorithm .33.

5. Create a function to scale the L, A, and B values of the image data from one file to have the same means and standard deviations as the other. Scaling is done with the following formula:

$$x\prime = (x - \overline{x}_{target})\frac{\sigma^x_{reference}}{\sigma^x_{target}} + \overline{x}_{reference} \tag{7}$$

The steps are as follows: 1) subtract the average value in the target image from each pixel's value. 2) multiply each value by the standard deviation for the reference image over the standard deviation for the target image. 3) add the average value from the reference image to each value. See Algorithm .34.

6. Update the main function to read in two files using the names on the command line, convert them to CIELAB, compute the means and standard deviations, scale the target image, and output the result. See Algorithm .35.

المنارة للاستشارات
www.manaraa.com

```
void lab_mean (struct fileinfo *image) {
   int i, j, k;
   for (k=0; k < 3; ++k) {
      image->mean[k] = 0;
   }
   for (i=0; i < image->height; ++i) {
      for (j=0; j < image->width; ++j) {
         for (k=0; k < 3; ++k) {
            image->mean[k]+= image->lab[i*image->width*3+j*3+k];
         }
      }
   }
   for (k=0; k < 3; ++k) {
      image->mean[k] /= image->width * image->height;
   }
}
void lab_stddev (struct fileinfo *image) {
   int i, j;
   float sq_sum[3] = {0,0,0};
   for (i=0; i < image->height * image->width; i++) {
      for (j=0; j < 3; j++) {
         sq_sum[j] += (image->lab[i*3+j]-image->mean[j]) *
                      (image->lab[i*3+j]-image->mean[j]);
      }
   }
   for (j=0; j < 3; j++) {
      sq_sum[j] /= image->height * image->width;
      image->std[j] = sqrt (sq_sum[j]);
   }
}
```

Algorithm .33: Mean and standard deviation

```
void scale (struct fileinfo *target, const struct fileinfo *reference) {
   int i, k;

   /* for 2D referencing of the array */
   float (*twoD)[3] = (float (*)[3])target->lab;
   for (i=0; i < target->height * target->width; i++) {
      for (k=0; k < 3; k++) {
         twoD[i][k] -= target->mean[k];
         twoD[i][k] *= reference->std[k]/target->std[k];
         twoD[i][k] += reference->mean[k];
      }
   }
}
```

Algorithm .34: Image scaling

المنارة للاستشارات
www.manaraa.com

```
#define SCENE 0
#define REFERENCE  1

int main (int argc, char **argv) {
   struct fileinfo files[2];
   int i;
   if (argc < 3) {
      fprintf (stderr, "Usage: %s target reference\n", argv[0]);
      return EXIT_FAILURE;
   }
   for (i=SCENE; i <= REFERENCE; ++i) {
      files[i].name = argv[i+1];
      if (!getImage (&files[i])) {
         fprintf (stderr, "Error reading %s \n", files[i].name);
         return EXIT_FAILURE;
      }

      files[i].lab = (float *) malloc(files[i].width *
                      files[i].height * 3 * sizeof(float));
      rgb_to_lab (files[i].rgb, files[i].lab, files[i].width *
                                          files[i].height);

      lab_mean (&files[i]);
      lab_stddev (&files[i]);
    }

   scale (&files[SCENE], &files[REFERENCE]);

   lab2rgb (files[SCENE].rgb, files[SCENE].lab,
            files[SCENE].width * files[SCENE].height);

   outputPPM (&files[SCENE], stdout);
   free_struct (&files[SCENE]);
   free_struct (&files[REFERENCE]);

   return EXIT_SUCCESS;
}
```

Algorithm .35: Color transfer main function

المنارة للاستشارات
www.manaraa.com

Appendix B CS2 Guide

B.1 Credits

4 (3 hour lecture and 2 hour lab)

B.2 Prerequisites

Students taking this course should have had one semester (or equivalent) of programming and are expected to have experience with the following:

- Basic C programming
- Programming logic (loops and branches)
- Functions and parameter passing
- Pointers
- Built-in data structures (integers, characters, strings, and floats)
- User-defined data structures (structures, enumerated types, typedef)
- Dynamic memory allocation
- Command-line arguments
- File input/output (opening, reading, and writing)

B.3 Course Goals

This course teaches the following computer science skills and techniques:

- Standard programming techniques including recursion.
- Elementary data structures: arrays, linked lists, stacks, and queues, as well as a basic understanding of the complexity of these structures.
- Large program organization and modularization.
- Function pointers and unions to approximate object-oriented code.

المنارة للاستشارات
www.manaraa.com

- Object-oriented programming and design.
- Incorporation of mathematics into programming.

B.4 Course Description

Each course is structured around a large, semester-long, graphical project. CS2 is structured around writing a raytracer: a technique for rendering realistic images by modeling the path from a given starting point to defined objects on a scene. An image can be raytraced by beginning each ray at a given starting point (eye point) and shooting the rays in the direction of each pixel, computing the color resulting from any intersections, and print the resulting color.

To do something as large as the raytracer, programmers must break it up into smaller phases. This raytracer can be broken down into as many as 15 phases, which can then be grouped into assignments as the instructor wishes. In the following lesson guide are descriptions of each phase, including examples solutions for the instructor's sake. These solutions are in C/C++ and are merely for guidance and not meant to imply that there are not other, better ways to raytrace.

B.5 Resources

The classic reference on raytracing is Glassner's *An Introduction to Ray Tracing* (1989. Academic Press Ltd.).

B.6 Lesson Guide

B.6.1 Suggested Course Policies

1. Requirement for on time work.
2. Extra credit for early work.
3. Maximum grade for simply meeting guidelines be lower than 100%.
4. Allowance of problem discussion and minor debugging with other students.
5. Prohibition of code sharing, whether verbally or electronically.

للاستشارات المنارة
www.manaraa.com

```
void raytrace (unsigned char *image, int width, int height) {
   int i;
   for (i=0; i < width*height; ++i) {
      image[i*CHANNELS+0]=255; /*CHANNELS=3 for red,green,blue*/
      image[i*CHANNELS+1]=0;   /* 255,0,0 produces red */
      image[i*CHANNELS+2]=0;
   }
}
```

Algorithm .36: Stub raytrace method to fill the image data array with red pixels

B.6.2 Selling the Assignment

This is an important opportunity for the instructor to sell students on the idea of investing time into an assignment with exciting results. Selling the assignment might include the display of images the students will be able to create, description of the technique, and explanation of the impact of this technique in industry.

B.6.3 Phase 1

The suggested first phase is the creation of a solid-colored PPM format image file output to standard out. Although it is not yet raytracing, the program can still be given a structure compatible with later raytracing. Requires knowledge: arrays, array access, data (bytes) functions, data and ASCII output, PPM image format, and IO redirection.

1. Create a `raytrace` function that fills the provided `unsigned char` array with the appropriate number of pixels (RGB values) needed. Each channel of each pixel is represented by a byte in the range of [0,255], which is the size and range of unsigned characters, making them the best representation choice. As a note, this method is of course not raytracing yet; instead, it is more of a "stub" method, providing valid output to test before the next step. e.g. Algorithm .36.

2. Create an `output` method to create the PPM format image file using the flat pixel array generated by `raytrace`. PPM format requires the first information in the header to be the "magic number" indicating the file type; in this case: `P6`. After the magic number is the specification of the integers representing width of the image, the height, and finally the maximum value for each channel (red, green, and blue) of each pixel in the image. (Each channel will be represented by one byte with maximum values of 0 255.) Each piece of information can be separated by any amount of whitespace and comment lines. (A comment is a line beginning with a `#`.) After the magic number, dimensions, and maximum value is a

المنارة للاستشارات
www.manaraa.com

```
void output (unsigned char *image, int width, int height) {
   printf ("P6\n%d %d\n255\n", width, height);
   fwrite (image, width*height*CHANNELS, 1, stdout);
}
```

Algorithm .37: Function for printing a PPM format image to standard out

```
int main () {
   unsigned char image[WIDTH*HEIGHT*CHANNELS];
   raytrace (image, WIDTH, HEIGHT);
   output(image, WIDTH, HEIGHT);
   return EXIT_SUCCESS;
}
```

Algorithm .38: Main function for invoking the appropriate functions to produce an image file

single (1) whitespace character. (Caution: some systems use two characters to represent end of line.) Finally, output the entire array of pixels. e.g. Algorithm .37.

3. Finally, create a main method to create the unsigned char array of the appropriate size, call raytrace, and call output. WIDTH and HEIGHT can be any typical image size, such as 800 by 600. e.g. Algorithm .38.

B.6.4 Phase 2

The second phase is a creating an image of any specified size. Required knowledge: command-line arguments, dynamic memory allocation, and string (char*) to integer conversion.

1. Accept command-line arguments in the main. See Algorithm .39.
2. Create an `unsigned char` pointer to hold the address of the image data.
3. Use the first two command-line arguments as the width and height of the image, if they are provided. The arguments are character arrays and must be converted to integers using the `atoi` function (Ascii TO Int). If no arguments are provided, use a default width and height.
4. Dynamically allocate an array to hold the data for the image, based on the specified size. Call `raytrace` and `output` as before using the image data. Note that in this case freeing the memory is unnecessary but is included for completeness.

المنارة للاستشارات
www.manaraa.com

```c
int main (int argc, char *argv[]) {
   unsigned char *image;
   int width=WIDTH, height=HEIGHT;
   if (argc > 2) {
      width  = atoi(argv[1]);
      height = atoi(argv[2]);
   }
   image = (unsigned char *)malloc (sizeof (unsigned char) *
                                    width * height * CHANNELS);
   raytrace (image, width, height);
   output(image, width, height);
   free (image);
   return EXIT_SUCCESS;
}
```

Algorithm .39: Main function with parameters for accepting command-line arguments

B.6.5 Phase 3

Phase three is creating an image of any size of a sky. Although it is a basic step, this phase introduces the fundamental structure of the raytracer. Required knowledge: header files, forward declarations, structures, enumerated types, function pointers, scaling, procedural texturing, and pixel location to 3D coordinate conversion.

1. Create 3D "virtual world" dimensions. The output image of the raytracer is 2D, but the objects traced in the image are defined in a 3D world space to create correct effects. Therefore the locations and dimensions of objects in the raytracer should be defined as 3D coordinates to be later projected onto the 2D image. The sample output images will reflect the common monitor viewing ratio of 4:3 (e.g. 1024,768); therefore, the 3D world coordinates also have a 4:3 ratio. In the example code, the x component width of the image will be 4 in the range of [-2,2]. The y component height will be 3 in the range of [-1.5,1.5]. The z component will be zero at the screen with positive and negative values of z behind or ahead of the screen. The dimensions can be stored as global or local constants, or as part of a scene structure for storing scene information. See Algorithm .40.

2. Create a `point` structure to store floating point x, y, z coordinates in the 3D virtual world. They may be stored in an array or as named variables, or (using a `union`) both. Array storage opens the possibility for iteration. On the other hand, named variables tend to be easier to read. The example code uses an array with an enumerated type to improve readability.

المنارة للاستشارات
www.manaraa.com

```
struct scene {
   int width; /* will be set to 4 */
   int height; /* will be set to 3 */
};

enum coord {X, Y, Z};
struct point {
   double coords[3];
};

struct point eye = {{0.0, 1.5, 4.0}}; /* high and well back. */

enum channel {RED, GREEN, BLUE};
struct color {
   double channels[3];
};
```

Algorithm .40: Declaration of the scene structure

3. Create an "eye" point to be the starting point of all the rays to be traced. Each ray shot to create the image is defined by a starting point and a direction point. At this stage of raytracing, all rays will start at the eye point, which should be place in a position a little back from the screen and high enough to see everything.

   Raytracers can use either the "left-handed" or the "right-handed" coordinate system. If you are using the right-handed coordinate system, point your right index finger toward positive x (likely the right side of the screen). Keeping the right index finger toward positive x, point the right middle toward positive y (up). The right thumb is now pointing to positive z (toward your chest). Under the left-handed system, your thumb will point away from you (toward the screen). See Figure 24

   Either system may be used and may be switched for each semester to prevent code reuse among students. If you are using the right-handed system, the eye point should have a positive z value, and the objects should have negative z values.

4. Create a function to compute a direction point for each pixel. Each pixel traced will have a different direction point to aim for, based on its location in the image. The directions are based on pixel image coordinates. For example, in an image with dimensions of 800 by 600, each pixel has a row value in the range [0,599] and a column value in the range [0,799]. Each pixel's row and column values are converted to x, y, z coordinates. The z value is always going to be zero, so only the x and y values must

المنارة للاستشارات
www.manaraa.com

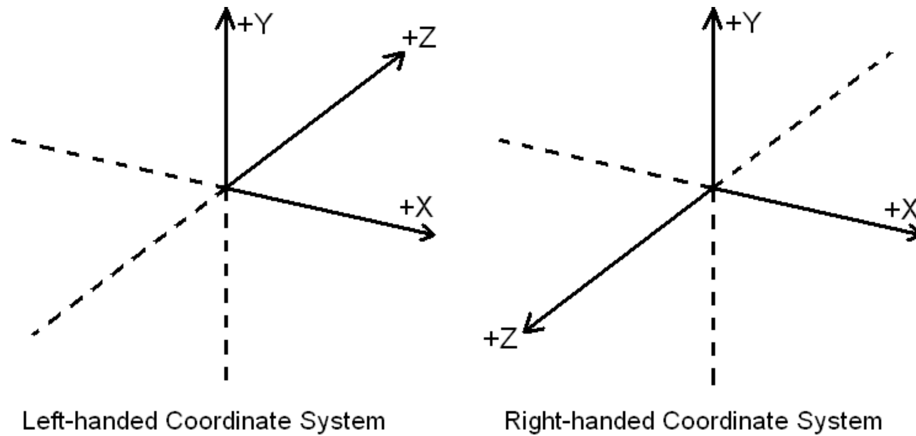


Figure 24: Demonstration of the left-handed and right-handed coordinate systems

be computed. Convert the column value to an x value requires scaling by the width. e.g.

$$width_{scene} \times \frac{column}{width - 1} \tag{8}$$

This scaling should produce a value in the range $[0, width_{scene}]$. However, the value needs to be in the range $[-\frac{width_{scene}}{2}, \frac{width_{scene}}{2}]$. Thus computing the x value requires the scaling and subsequent shift to the appropriate the range using subtraction. Computing y is similar, but y values start positive and work down to negative, so the subtraction is reversed. e.g.

$$x = width_{scene} \times \frac{column}{width - 1} - \frac{width_{scene}}{2} y = \frac{height_{scene}}{2} - height_{scene} \times \frac{row}{height - 1} z = 0 \tag{9}$$

5. Create a `color` structure to store the color values generated for a pixel by the raytracer. During raytracing, color values will be scaled and added to, making the [0,255] range a little hard to work with. Instead, each channel of each color will be stored as a floating point value in the range [0.0, 1.0] and be scaled after raytracing is complete into the byte [0, 255] range. The sample color structure uses the same format as the point structure.

6. Create a `sky` structure to store information needed to calculate the color value of the sky at a given point. The color of the sky will never be contingent on the lighting, so the color of the sky is considered the “ambient” color.

   The sample method of creating a sky is using “procedural texturing.” That is, the sky’s color is gener-

المنارة للاستشارات
www.manaraa.com

```
/* forward declaration of the sky's ambient computation function */
struct color sky_ambient (struct point spot);

struct sky {
   float blue; /* blue contribution */
   float base; /* minimum red, green contribution */
   float horizon; /* variable red, green contribution */
};

struct scene {
   int width; /* will be set to 4 */
   int height; /* will be set to 3 */
   struct sky sky;
};

struct color sky_ambient(const struct sky *sky,
                         const struct point *spot){
 struct color color;
 color.channels[RED] =(1-spot->coords[Y])*sky->horizon+sky->base;
 color.channels[GREEN]=(1-spot->coords[Y])*sky->horizon+sky->base;
 color.channels[BLUE] =sky->blue;
 return color;
}

struct sky {
   float blue, base, horizon;
   struct color(*ambient)(const struct sky*, const struct point*);
};
```

Algorithm .41: Declaration of the sky structure

ated "on the fly" by a function. A blue sky my be textured using a constant blue contribution with the red and green contribution varying based on the height of the sky. The higher areas of the sky will be more blue with the lower areas more white. The actual values used for blue contribution, base red and green contribution, and varying contribution will be stored in a `sky` structure and are something the students will want to decide. Some students may prefer to have other predominant colors in the sky, such as gray. The values relating to the sky should be stored in one structure. See Algorithm .41

7. Add `sky` to the `scene` structure.

8. Create a function to perform the sky's ambient color computation. The sky's ambient value is dependent on the `y` value at the ray's intersection point with the sky. However, since intersection with the sky is impossible, instead the normalized ray from the starting point to the direction will be used. Unfortu-

www.manaraa.com

```
struct scene set_scene () {
   /* 4 is the virtual-world width.
    * 3 is the virtual-world height.
    * 1.0 is sky's blue contribution.
    * .5 is the sky's base red, green contribution.
    * .4 is the sky's varying red, green contribution.
    * sky_ambient is a function pointer to the ambient function.
    */
   struct scene scene = {4, 3, {1.0, .5, .4, sky_ambient}};
   return scene;
}
```

Algorithm .42: Scene set up function

nately, normalization requires many added steps; temporarily, merely the direction point may be used. A sky will still be produced, giving students interesting output, and the later addition of normalization to produce a better sky. The y value will be subtracted from 1, because 1 is the maximum normalized y value.

9. Add a function pointer to the `sky` structure to the `sky_ambient` function. Storing a pointer in the sky structure to its function will allow to later approximation of object orientation and prepare students for object in C++.

10. Create a function to set up the scene to trace. The sample program's virtual world width and height are 4 and 3, respectively. The sky's blue contribution is as much as possible, i.e. 1.0. Red and green have a starting contribution of .5. Red, green values in the sky closer to the horizon will have added on up to .4. Finally, the `sky` structure holds a function pointer to its ambient function, e.g. `sky_ambient`. e.g. Algorithm .42.

11. Complete the function for converting pixels into 3D virtual-world coordinate. e.g. Algorithm .43.

12. Create a function to trace an individual pixel, using the existing starting point (eye) and computing the direction point. Currently, the only object to trace is the sky, which is always visible from any direction (unless it is blocked by another object). Therefore, the `trace_pixel` merely needs to call the ambient function associated with the `sky` variable. At this point the eye point is not being used but, it will be when later with direction ray normalization. Most of the parameters in the example code are pointers merely to improve runtime efficiency. For readability, "in" parameters are always marked as `const`. See Algorithm .44. Notice that, although `ambient` is a function pointer, it is not being de-referenced.

المنارة للاستشارات
www.manaraa.com

```
struct point virtual_coord (int row, int col, int height,
                            int width, const struct scene *scene){
   struct point point;
   point.coords[X] = scene->width * (col/((double)(width-1))) -
                     scene->width/2.0;
   point.coords[Y] = scene->height/2.0 -
                     scene->height * row/((double)(height-1));
   point.coords[Z] = 0;
   return point;
}
```

Algorithm .43: Conversion from pixel coordinates to world coordinates

```
struct color trace_pixel(const struct point *eye,
                         const struct point *dir,
                         const struct scene *scene) {
   return scene->sky.ambient(&scene->sky, spot);
}
```

Algorithm .44: Function to compute a specified pixel's values

Depending on the version of C, the function pointers may need to be dereferenced. e.g. Algorithm .45.

13. Create a method to convert the generated color's channels to the [0,255] range. Values should be scaled by 255 and then capped to the range, in case of minor overflow. e.g. Algorithm .46.

14. Finally, update the `raytrace` to reflect the changes to the raytracer. e.g. Algorithm .47.

    The resulting image should look similar to the Figure 25.

B.6.6 Phase 4

The forth phase is creating an image of any specified size with a sky and any number of spheres. Required knowledge: unions, definition of a sphere, ray-sphere intersection, the quadratic equation, macros, the dot product, point subtraction, comparing doubles to zero.

1. Create a sphere structure to hold the necessary data for tracing a sphere. A sphere is defined by a

```
   return (*scene->sky.ambient)(&scene->sky, spot);
```

Algorithm .45: Dereferencing of function pointer

المنارة للاستشارات
www.manaraa.com

```
void to_byte_range (struct color *color) {
   enum channel i;
   int channel;
   for (i=RED; i <= BLUE; ++i) {
      channel = (int)(color->channels[i] * 255 + 0.5);
      if (channel > 255) channel = 255;
      if (channel <   0) channel =   0;
      color->channels[i] = channel;
   }
}
```

Algorithm .46: Conversion of the pixel's channel values to the [0,255] range

```
void raytrace (unsigned char *image, int width, int height) {
   int i, j, k;
   struct point eye = {{0, 1.5, 4.0}};
   struct point dir;
   struct color color;
   struct scene scene = set_scene ();

   for (i=0; i < height; ++i) {
      for (j=0; j < width; ++j) {
         dir = virtual_coord (i, j, height, width, &scene);
         color = trace_pixel (&eye, &dir, &scene);
         to_byte_range (&color);
         for (k=RED; k <=BLUE; ++k) {
            image[(i*width +j)*CHANNELS + k] = color.channels[k];
         }
      }
   }
}
```

Algorithm .47: Complete program for creating a gradient sky pattern

المنارة للاستشارات
www.manaraa.com

Figure 25: Gradient sky

location (x, y, z coordinate), a radius size (floating point), and an ambient color function. Additionally, unlike the sky, the sphere will be intersected by only some of the rays shot. Thus the sphere will need an intersection test function.

2. Create the `sphere_ambient` function to return the ambient color at a given point on the sphere. For now, the spheres will have solid color. An attribute in the sphere structure will store the color of the sphere, or the color can be hard coded into the ambient function. See Algorithm .48.

3. Create a structure for holding any type of object. With the scene now containing any number of objects in the scene, these objects must be stored in a way that allows iteration. Since variables of different types(e.g. sky and sphere) may not be in one array, a generic `object` type must be made to hold either sky or sphere objects. Use of a union provides space for either a sky or a sphere, but not both. The ambient and intersection functions can be removed from the sky and sphere structures and put in the object structure. See Algorithm .49.

4. Update the sphere and sky ambient function to be compatible with the new structure: Algorithm .50.

5. Update the scene structure to hold the address of the objects array. Once definition of the sphere

المنارة للاستشارات
www.manaraa.com

```
struct sphere {
   float radius;
   struct point center;
   struct color color; /* representation of this object's color */
   struct color (*ambient)(const struct sphere*, const struct point*);
   int (*intersection) ();
};

struct color sphere_ambient(const struct sphere* sphere,
                            const struct point* spot) {
   return sphere->color;
}
```

Algorithm .48: Addition of sphere's color function

```
struct sky {
   float blue, base, horizon;
};
struct sphere {
   float radius;
   struct point center;
   struct color color;
};
struct object {
   struct color (*ambient)(const struct obj*, const struct pt*);
   int (*intersection) ();
   union geometry {
      struct sky sky;
      struct sphere sphere;
   } type;
};
```

Algorithm .49: Consolidation of shared attributes in the object structure

المنارة للاستشارات
www.manaraa.com

```
struct color sphere_ambient (const struct object* obj,
                             const struct point* spot){
   return obj->type.sphere.color;
}

struct color sky_ambient (const struct object *obj,
                          const struct point *spot) {
   struct color color;
   color.channels[RED]   = (1-spot->coords[Y]) *
                     obj->type.sky.horizon + obj->type.sky.base;
   color.channels[GREEN] = (1-spot->coords[Y]) *
                     obj->type.sky.horizon + obj->type.sky.base;
   color.channels[BLUE]  = obj->type.sky.blue;
   return color;
}

struct scene {
   int width, height;
   int num_objs; /* number of objects in the array */
   struct object *objects;
};
```

Algorithm .50: Modification of color functions to match new object structure

functions is complete, the objects may be added to the scene's array.

6. Create the ray-sphere intersection test formula. A ray is defined as the set of all points p along the line defined by $startingpoint \times (1 - t) + directionpoint \times t$.

$$\{p|sp(1-t) + dp \times t\} \tag{10}$$

t is a sort of ratio of the distance along the ray. If t is 1, the point is at the direction point. If t is 0, the point is at the start point. t is not a true distance value, but useful for distance comparison and for determining whether a ray intersects an object. The distance formula may be used when exact distances are needed.

Along with the definition of a ray, the definition of a sphere must be used to determine intersection. If c is the center point of the sphere and r is the radius, a sphere is defined as

$$\{p|(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 = r^2\} \tag{11}$$

المنارة للاستشارات
www.manaraa.com

A ray intersects a sphere when both formulas are satisfied. Thus, plug the ray formula into the sphere formula. i.e.

$$((sp_x(1-t)+dp_xt)-c_x)^2+((sp_y(1-t)+dp_yt)-c_y)^2+((sp_z(1-t)+dp_zt)-c_z)^2=r^2 \tag{12}$$

To calculate the t value, put the above equation in the the form $ax^2+bx+c=0$, where $x=t$ and solve for t using the quadratic equation. i.e.

$$((dp_x-sp_x)t+sp_x-c_x)^2+((dp_y-sp_y)t+sp_y-c_y)^2+((dp_z-sp_z)t+sp_z-c_z)^2-r^2=0 \tag{13}$$

$$\begin{aligned}
&(dp_x-sp_x)^2t^2+(dp_y-sp_y)^2t^2+(dp_z-sp_z)^2t^2\\
&+2(dp_x-sp_x)(sp_x-c_x)t+2(dp_y-sp_y)(sp_y-c_y)t+2(dp_z-sp_z)(sp_z-c_z)t\\
&+(sp_x-c_x)^2+(sp_y-c_y)^2+(sp_z-c_z)^2-r^2=0
\end{aligned} \tag{14}$$

$$\begin{aligned}
&((dp_x-sp_x)^2+(dp_y-sp_y)^2+(dp_z-sp_z)^2)t^2\\
&+2t((dp_x-sp_x)(sp_x-c_x)+(dp_y-sp_y)(sp_y-c_y)+(dp_z-sp_z)(sp_z-c_z))\\
&+(sp_x-c_x)^2+(sp_y-c_y)^2+(sp_z-c_z)^2-r^2=0
\end{aligned} \tag{15}$$

$$\begin{aligned}
a=&\quad(dp_x-sp_x)^2+(dp_y-sp_y)^2+(dp_z-sp_z)^2\\
b=&\quad 2((dp_x-sp_x)(sp_x-c_x)+(dp_y-sp_y)(sp_y-c_y)+(dp_z-sp_z)(sp_z-c_z))\\
c=&\quad(sp_x-c_x)^2\times(sp_y-c_y)^2\times(sp_z-c_z)^2-r^2
\end{aligned} \tag{16}$$

Now use the quadratic equation to solve for t:

$$t=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{17}$$

This equation could generate two values of t. These two values represent the two points at which the ray interests the sphere. If the ray is tangent to the sphere, there will be only one unique value of t.

If both values of t are positive, the ray intersects the sphere in two places ahead of the ray (see Figure 26, and the closer point is the one to use. If both t values are negative, the sphere is behind the ray, and thus is not visible to this ray. If only one t value is negative, the ray was shot from inside the sphere. In this case, the positive t value is the only usable one. Thus the ray-sphere intersection is based on the

المنارة للاستشارات
www.manaraa.com

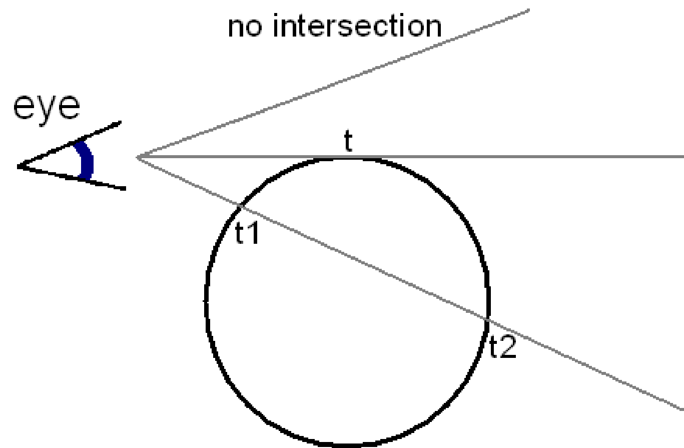


Figure 26: Ray-sphere intersection

```
#define SQUARE(x)((x)*(x))
```

Algorithm .51: Square macro

closer positive t value generated by the quadratic equation.

In some cases, no t value may exist. If $b^2 - 4ac$ is negative, t cannot be computed, because $\sqrt{b^2 - 4ac}$ would be imaginary. This case occurs if the ray never intersects the sphere. $b^2 - 4ac$ is called the discriminant and determines if an intersection occurs at all.

7. Create mathematical utility functions to aid in the writing of the ray-sphere intersection function: square, point subtraction, and dot product. These functions, as well as other small utility functions throughout this project (e.g. bounce, normalize, multiply, etc.) can be integrated as lab assignments.

   The square function is needed throughout the intersection test and is very simple to write as a C macro. Be sure to use enough parentheses to prevent incorrect results in cases involving expressions like `SQUARE(x-y)`. See Algorithm .51. Point subtraction is merely a component by component subtraction: Algorithm .52. Finally, the dot product of two vectors (or points) is the sum of the product of each component of the two vectors: $p1 \cdot p2 = p1_x p2_x + p1_y p2_y + p1_z p2_z$. The dot product is useful in determining the cosine of the angle between two vectors, because $X \cdot Y = |X||Y| cos\theta$. If vectors X and Y are normalized, the dot product results in $cos\theta$. In this phase, although no cosines of angles need to be computed, the dot product is handy for computing the values for a, b, and c in sphere intersection. All three computations require multiplication and summation the components.

المنارة للاستشارات
www.manaraa.com

```
struct point subtract (const struct point *p1, const struct point *p2) {
   struct point sum;
   enum coord i; /* an integer counter will also suffice */
   for (i=X; i <= Z; ++i) {
      sum.coords[i] = p1->coords[i] - p2->coords[i];
   }
   return sum;
}

double dot_prod (const struct point *p1, const struct point *p2) {
   double result=0.0;
   enum coord i;
   for (i=X; i <= Z; ++i) {
      result += p1->coords[i] * p2->coords[i];
   }
   return result;
}
```

Algorithm .52: Point subtraction function

```
struct intersection {
   double t;
   struct point location;
   struct object *obj;
};
```

Algorithm .53: Intersection structure

8. Create an intersection structure to be used with the ray-sphere intersection function. Intersection may be computed using the sphere's center and radius, the starting point and direction. The resulting t and intersection point additionally need storage locations. It is suggested that the t value, location, and a pointer to the object hit (in this case, the sphere passed in) are all stored in one `intersection` structure to minimize overhead. See Algorithm .53.

9. Write the ray-sphere intersection function. The function accepts a sphere object to test for intersection, a ray starting point, and direction point. The function will return a boolean value indicating whether an intersection occurred, as well as an intersection structure variable (via pointer). Note that the `struct object *` parameter will not be altered, but it cannot be declared constant and have its address stored in the intersection struct variable. See Algorithm .54. Now the a, b, and c values may be computed for use in the quadratic equation. Their computation will be simplified with the use of the utility functions. The computed a, b, and c values are now used in the quadratic equation. First, it must be determined

www.manaraa.com

```
int sphere_intersect (struct object *obj, const struct point *sp,
           const struct point *dp, struct intersection *intersect) {
   enum coord i; /* used in the for loop */
   double t;
   struct point dp_minus_sp = subtract (dp, sp);
   struct point sp_minus_c  = subtract (sp, &obj->type.sphere.center);

   double a = dot_prod (&dp_minus_sp, &dp_minus_sp);
   double b = 2.0 * dot_prod (&dp_minus_sp, &sp_minus_c);
   double c = dot_prod (&sp_minus_c, &sp_minus_c) -
              SQUARE (obj->type.sphere.radius);

   double discr = b*b - 4.0 * a * c;
   if (discr >= 0) {
      t = (-b - sqrt(discr))/(2.0*a);
      if (t < MIN) {
         t = (-b + sqrt (discr))/(2.0*a);
      }
      if (t > MIN) { /* if t is positive, there is an intersection */
         for (i=X; i <= Z; ++i) {
            intersect->location.coords[i] = sp->coords[i]*(1-t)
                                          + dp->coords[i]*t;
         }
         intersect->t = t;
         intersect->obj = obj;
         return 1;
      }
   }
   return 0;
}
```

Algorithm .54: Sphere intersection function

www.manaraa.com

```
/* in the header */
#define MAX 60

/* in the main code */
int sky_intersect (struct object *obj, const struct point *sp,
       const struct point *dp, struct intersection *intersect) {
   intersect->location = *dp;
   intersect->t = MAX;
   intersect->obj = obj;
   return 1;
}
```

Algorithm .55: Sky intersection function

whether the determinant. If it is not, the ray never intersects the sphere. Next is the computation of the first t value. If t too small to be significant, the other t value should be computed and used. Because floating point values are not infinitely precise, t value cannot simply be compared to zero. It is possible for the error in a floating point number to give a false positive. Instead, t should be compared to some minimum value to allow room for error. A suggested value for MIN is .000001.

If the ray intersects the sphere, the t value can be used to calculate the intersection point (x, y, z coordinates). e.g. $location = sp(1 - t) + dp \times t$.

Finally, if t was not greater than zero, there was no intersection, an the function should return false.

10. Create a ray-sky intersection function that always returns true. From this point on, all objects raytraced will need to have ambient functions and intersection functions. If sky has an intersection function, object intersections will be simplified by the parallelism. For each object in the scene, will will check for intersection and choose the closest object. Sky will always be intersected, but at the furthest distance: `MAX`. `MAX` is a constant defining the largest distance any ray will travel. A safe large distance is 60. There may be larger or smaller distances that work better, depending on the implementation. Sky's intersection point will still be the direction point, but once normalization is available, sky's intersection point should be the normalized ray from the starting point to the direction point. See Algorithm .55.

11. Update the object structure to have the full signature of the intersection functions. This is important to add, because with a full signature, the C compiler can identify any mistakes in calling the functions. See Algorithm .56.

12. Set up the scene. The example scene in Algorithm .57 has two spheres and a sky.

المنارة للاستشارات
www.manaraa.com

```
struct object {
   struct color (*ambient)(const struct obj*, const struct pt*);
   int (*intersection) (struct object *, const struct point *,
                  const struct point *, struct intersection *);
   union geometry {
      struct sky sky;
      struct sphere sphere;
   } type;
};
```

Algorithm .56: Signature information added to the intersection function pointer declaration

13. Update `trace_pixel` to locate the closest object and return its ambient color: Algorithm .58. The raytracer should now create an image with blue sky, a red circle, and a cyan circle (Figure 27).

    The spheres are very flat, because there is no diffuse lighting yet. Additionally, the sphere on the edge of the scene may appear slightly stretched: Figure 28.

    This stretching effect is caused by the projection of the 3D scene onto a 2D image. Rays traced toward the edges intersect objects that are outside of the 2D image dimensions. Thus the edges of the image tend to be stretched. This effect can be reduced by moving the eye point further back from the image.

B.6.7 Phase 5

The fifth phase is creating an image of any specified size with a sky, any number of spheres, and a horizontal floor. Required knowledge: Ray-floor intersection.

1. Create a floor structure to store all floor-related information. The example floor is an infinite, horizontal plane with a solid color. Therefore, The floor's definition is merely a color and a y value called "height." See Algorithm .59.

2. Add the floor structure to the union in the object structure, as in Algorithm .60.

3. Create a `floor_ambient` function to return the color of the floor, as in Algorithm .61.

4. Create a `floor_intersect` function to determine whether a ray intersects the floor. See Algorithm .62. Again, a ray is defined as $\{p|sp(1-t) + dp \times t\}$. The floor is an infinite plane at a specified height. Therefore, the floor is defined as the set of all points whose y values equal the floor's height.

المنارة للاستشارات
www.manaraa.com

```
struct scene set_scene () {
   struct scene scene;
   int i=0;
   scene.width  = 4;
   scene.height = 3;
   scene.num_objs = 3;
   scene.objects = (struct object *) malloc
                   (sizeof (struct object) * scene.num_objs);

   /* first object: a dark red sphere on the lower right */
   scene.objects[i].type.sphere.center.coords[X] = 1.4;
   scene.objects[i].type.sphere.center.coords[Y] = -1.25;
   scene.objects[i].type.sphere.center.coords[Z] = -1.5;
   scene.objects[i].type.sphere.radius = .75;
   scene.objects[i].type.sphere.color.channels[RED]=139.0/255.0;
   scene.objects[i].type.sphere.color.channels[GREEN] = 0.0;
   scene.objects[i].type.sphere.color.channels[BLUE]  = 0.0;
   scene.objects[i].ambient = sphere_ambient;
   scene.objects[i].intersection = sphere_intersect;

   /* second object: a cyan sphere on the left */
   ++i;
   scene.objects[i].type.sphere.center.coords[X] = -1.5;
   scene.objects[i].type.sphere.center.coords[Y] = -0.25;
   scene.objects[i].type.sphere.center.coords[Z] = -2.25;
   scene.objects[i].type.sphere.radius = .75;
   scene.objects[i].type.sphere.color.channels[RED]   = 0.0;
   scene.objects[i].type.sphere.color.channels[GREEN] = 1.0;
   scene.objects[i].type.sphere.color.channels[BLUE]  = 1.0;
   scene.objects[i].ambient = sphere_ambient;
   scene.objects[i].intersection = sphere_intersect;

   /* third object: a blue sky */
   ++i;
   scene.objects[i].type.sky.blue = 1.0;
   scene.objects[i].type.sky.base = .5;
   scene.objects[i].type.sky.horizon = .4;
   scene.objects[i].ambient = sky_ambient;
   scene.objects[i].intersection = sky_intersect;
   return scene;
}
```

Algorithm .57: Specification of the objects in the scene

المنارة للاستشارات
www.manaraa.com

```
struct color trace_pixel (const struct point *eye,
                          const struct point *dir,
                          const struct scene *scene) {
   int i;
   struct intersection curr, closest;
   closest.t = MAX+1;

   /* find the closest intersected object (i.e. the intersected
      object with the smallest t value. */
   for (i=0; i < scene->num_objs; ++i) {
      if (scene->objects[i].intersection (&scene->objects[i],
           eye, dir, &curr) && curr.t < closest.t) {
         closest = curr;
      }
   }
   return closest.obj->ambient (closest.obj, &closest.location);
}
```

Algorithm .58: Addition of nearest object search

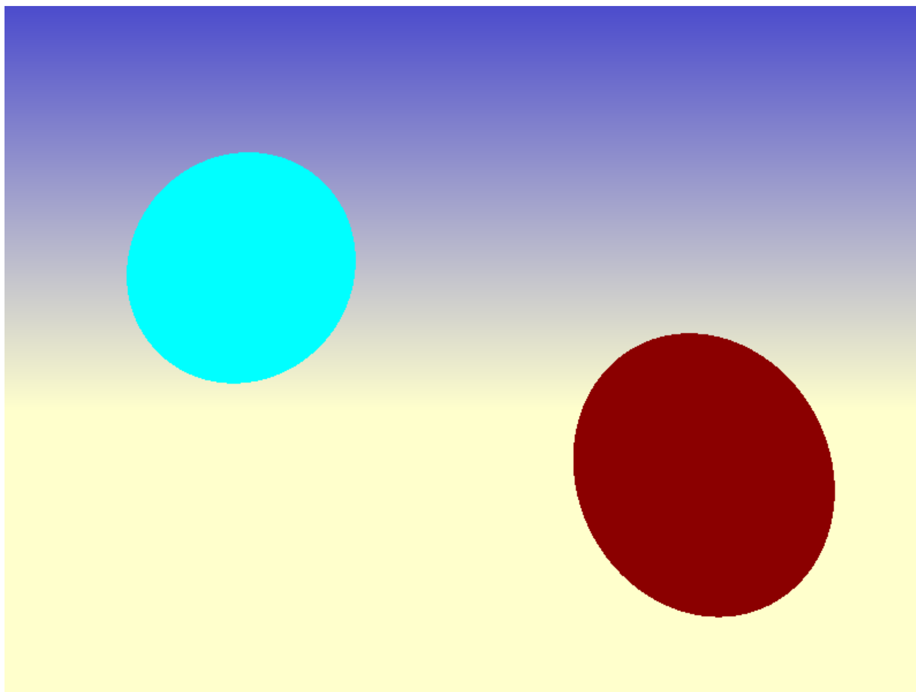


Figure 27: Blue sky and filled circles

المنارة للاستشارات
www.manaraa.com

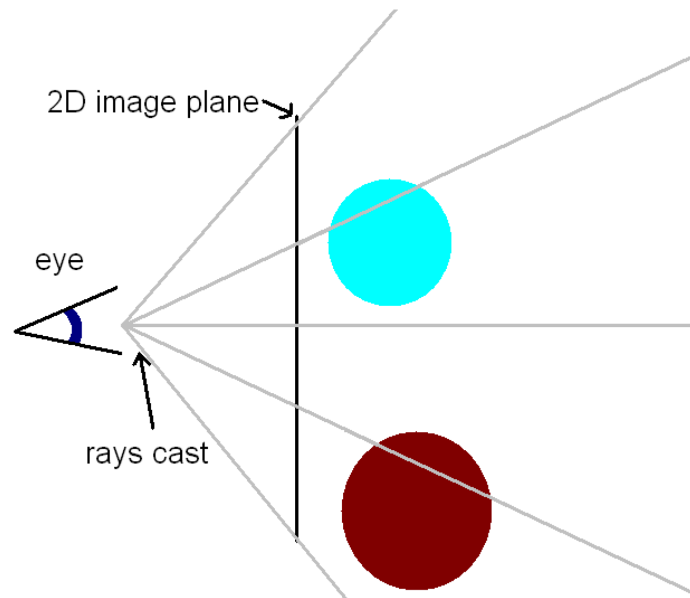


Figure 28: Ray scene projection

```
struct floor {
   int height;
   struct color color;
};
```

Algorithm .59: Specification of the floor structure

```
struct object {
   struct color (*ambient)(const struct obj*, const struct pt*);
   int (*intersection) (struct object *, const struct point *,
                        const struct point *, struct intersection *);
   union geometry {
      struct sky sky;
      struct sphere sphere;
      struct floor floor;
   } type;
};
```

Algorithm .60: Addition of floor structure to the geometry union

```
struct color floor_ambient(const struct object* obj,
                           const struct point* spot){
   return obj->type.floor.color;
}
```

Algorithm .61: Floor's color function

للاستشارات المنارة
www.manaraa.com

```
int floor_intersect (struct object *obj, const struct point *sp,
         const struct point *dp, struct intersection *intersect) {

   /* in the floor_intersection function */
   if (fabs (dp->coords[Y]-sp->coords[Y]) < MIN) return 0;

   intersect->t = (obj->type.floor.height - sp->coords[Y])/
                  (dp->coords[Y] - sp->coords[Y]);
   if (intersect->t < MIN) return 0;
   intersect->obj = obj;
   intersect->location.coords[X] = sp->coords[X] *
              (1.0-intersect->t) + dp->coords[X] * intersect->t;
   intersect->location.coords[Y] = obj->type.floor.height;
   intersect->location.coords[Z] = sp->coords[Z] *
              (1.0-intersect->t) + dp->coords[Z] * intersect->t;
   return 1;
}
```

Algorithm .62: Floor intersection function

i.e. $\{p|p_y = height\}$. The ray intersects the floor where both equations are satisfied:

$$(sp_y(1-t) + dp_y t = height \\ (dp_y - sp_y)t + sp_y = height \tag{18}$$

$$t = \frac{height - sp_y}{dp_y - sp_y} \tag{19}$$

If the t value is negative, the intersection with the ray occurs before the starting point and is not visible. if $dp_y = sp_y$, the ray is parallel to the floor and will not intersect it. If the values are the same as the floor's height, then technically the ray is traveling inside the floor, and will be ignored. Unfortunately, determining whether $dp_y = sp_y$ is difficult, because floating point representation makes it unlikely that the two values will ever be identical. Instead of direct comparison, the absolute difference between the values should be compared to a minimum value MIN, and if the difference is less, the two points will be treated as equal.

Once ray-floor intersection is confirmed, the intersection point must be computed. Once again, $location = sp(1-t) + dpt$. The y value, however, will always be the height of the floor. Thus only x and z and must be computed.

5. Add a floor to the scene in the `set_scene` function: Algorithm .63. The resulting image will have cyan

المنارة للاستشارات
www.manaraa.com

```
struct scene set_scene () {
   struct scene scene;
   int i=0;
   scene.width  = 4;
   scene.height = 3;
   scene.num_objs = 4;
   scene.objects = (struct object *) malloc (sizeof
                   (struct object) * scene.num_objs);

   /* first object: a beige floor */
   scene.objects[i].type.floor.height = -2;
   scene.objects[i].type.floor.color.channels[RED]   = 1.0;
   scene.objects[i].type.floor.color.channels[GREEN] = 235.0/255.0;
   scene.objects[i].type.floor.color.channels[BLUE]  = 205.0/255.0;
   scene.objects[i].ambient = floor_ambient;
   scene.objects[i].intersection = floor_intersect;

   /* second object: a dark red sphere on the lower right */
   ++i;
   /* . . . */
```

Algorithm .63: Addition of the floor to the scene

```
struct floor {
   int height;
   struct color color1, color2;
};
```

Algorithm .64: Two colors in the floor structure

and dark red circles, a blue sky, and a beige floor: Figure 29.

B.6.8 Phase 6

The six phase creates an image of any specified size with a sky, any number of spheres, and a checkered floor. The checkered floor will have two alternating colors applied procedurally. Required knowledge: Mathematical function *floor* and modular arithmetic.

1. Update the `floor` structure to have two colors. See Algorithm .64.
2. Update the `set_scene` to set both floor colors. See Algorithm .65.
3. Update the `floor_ambient` function to create a checkered pattern. The checkering will be based on the x and z value of the intersection location. (The y value will always be the floor height.) Each checker

المنارة للاستشارات
www.manaraa.com

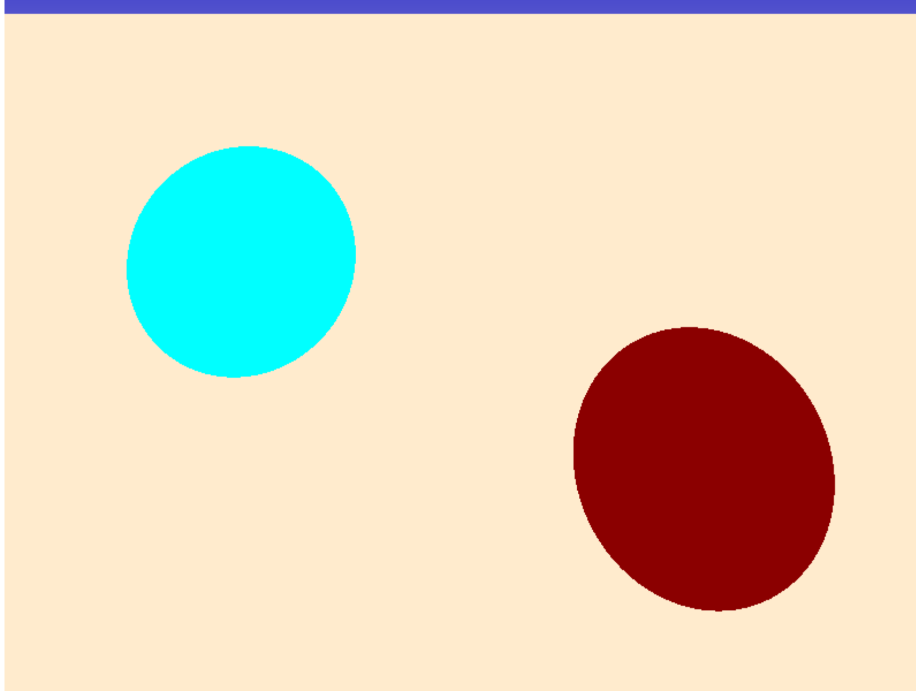


Figure 29: Blue sky, circles, and floor

```
/* first object: a checkered floor */
scene.objects[i].type.floor.height = -2;
scene.objects[i].type.floor.color1.channels[RED]   = 1.0;
scene.objects[i].type.floor.color1.channels[GREEN] = 235.0/255.0;
scene.objects[i].type.floor.color1.channels[BLUE]  = 205.0/255.0;
scene.objects[i].type.floor.color2.channels[RED]   = 139.0/255.0;
scene.objects[i].type.floor.color2.channels[GREEN] =  69.0/255.0;
scene.objects[i].type.floor.color2.channels[BLUE]  =  19.0/255.0;
scene.objects[i].ambient = floor_ambient;
scene.objects[i].intersection = floor_intersect;
```

Algorithm .65: Specification of the floor colors

المنارة للاستشارات
www.manaraa.com

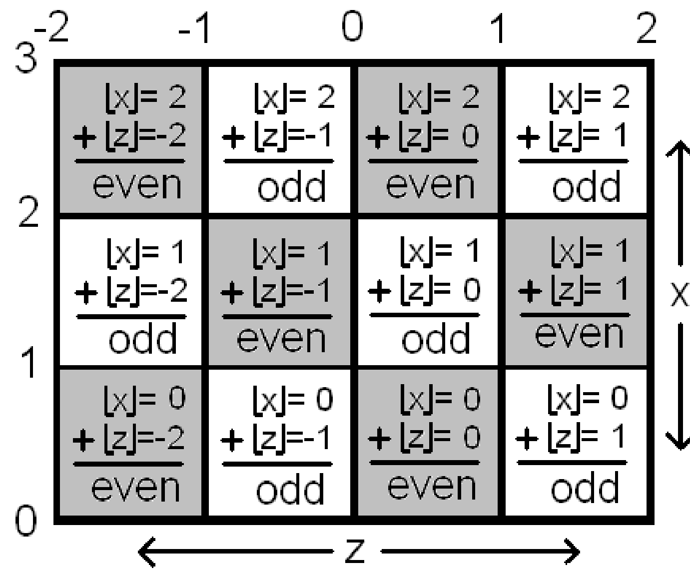


Figure 30: Checker algorithm

will be a square whose width and height are each size 1 in world coordinates. Each color begins at a round world coordinate value (-2.0, -1.0, 0.0, 1.0, etc.) and ends at the next integer value.

The color of the floor at a given location will be based on the mathematical floor of the x and z values. The floor of a number x ($\lfloor x \rfloor$) is the largest integer less than or equal to x. The floor function is used for both x and z to generate two integers: $\lfloor x \rfloor$ and $\lfloor z \rfloor$. At every other square, the sum of these integers is even. See Figure 30.

Therefore, a simple way to apply the checkered texture procedurally is to apply one color whenever the sum of $\lfloor x \rfloor$ and $\lfloor z \rfloor$ is even and the other when the sum is odd. Of course, even and odd values can be determined using modulo 2. See Algorithm .66 and Figure 31.

B.6.9 Phase 7

The seventh phase is the use of an Object-Oriented program in an OO language to create an image of any specified size with a sky, any number of spheres, and a checkered floor. Converting to C++ can be postponed, but it is not suggested. At this point, students have actually implemented Object-Oriented programs, but have done so with unions and function pointers. Staying with a procedural programming from this point on will not add any knowledge, and beginning with an OO language will introduce many new concepts with ample time for practicing programming in a new paradigm. Additionally, an early conversion to C++ reduces the amount of code that will need to be rewritten in C++. Covered knowledge: C++ classes,

المنارة للاستشارات
www.manaraa.com

```
struct color floor_ambient(const struct object* obj,
                           const struct point* spot){
   int floorX = (int) floor (spot->coords[X]);
   int floorZ = (int) floor (spot->coords[Z]);

   if ((floorX+floorZ)%2==0) {
      return obj->type.floor.color1;
   } else {
      return obj->type.floor.color2;
   }
}
```

Algorithm .66: Functional floor texture

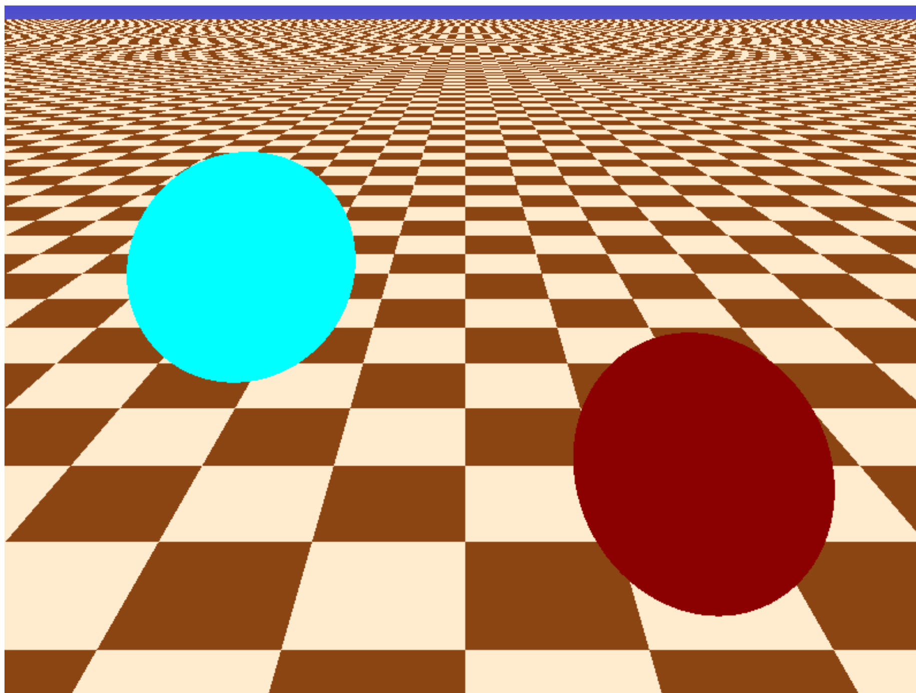


Figure 31: Sky, circles, and checkered floor

المنارة للاستشارات
www.manaraa.com

```
#ifndef POINTCLASS
#define POINTCLASS
// all of the Point header code
#endif
```

Algorithm .67: If-not-defined preprocessor directive

C++ inheritance, virtual methods, purely virtual methods, references, static methods, destructors, operator overloading, anonymous structures, constructor initialization, default parameters, constant member functions, typedef, iostreams, and make files.

Not all covered knowledge needs to be introduced at this phase. More advanced information is provided for future direction.

1. Design the raytracer. The sample raytracer has many functions and variables that should be grouped appropriately. Most structures will become classes, and functions that operate on those structures will become methods of those classes. The structures point, color, object, sky, sphere, floor, and scene should all become classes, as also may be the raytracer.

2. Create the Point class. The Point class will hold x, y, z coordinates, methods for initializing, accessing, and modifying those coordinates, the maximum coordinate value in the raytracer, the minimum coordinate value considered significant, a method for 2D to 3D conversion, and point-related operations, such as subtraction and dot product.

   Note that since the Point header will likely be included in multiple files, it may be helpful to surround the entire header with the notation in Algorithm .67. The components in the Point class can be named doubles, an array of doubles or both. Not all C++ compilers will allow anonymous structures as shown in Algorithm .68. If not, name them. Declare the maximum distance allowed in the scene and the minimum distance considered significant.

   The x, y, z values will be initialized in the point class constructor. The constructor must accept up to 3 values, but it is fine to have default component values if the user does not wish to specify the components initially. In Object-Oriented programming, instance variables, such as the coordinates, are typically private to allow the class to control any access and modification. Therefore, accessor methods should be provided to access these components. Since these methods are accessors only and do not alter the state of the object (i.e. they do not change any instance variables), they should be declared constant methods. Note that putting `"void"` in the parameters is not necessary. It is C notation for confirming

المنارة للاستشارات
www.manaraa.com

```
// in the header file Point.h
typedef enum {X, Y, Z} coord;

class Point {
   private:
      union {
         struct {
            double x, y, z;
         };
         double xyz[3];
      };
   public:
      static const double MAX = 60.0;
      static const double MIN = 0.000001;
      Point (double x=0.0, double y=0.0, double z=0.0);
      double get_x(void) const { return x; }
      double get_y(void) const { return y; }
      double get_z(void) const { return z; }
      double  operator[] (int i) const { return xyz[i];}
      double &operator[] (int i) { return xyz[i];}
      double dot_prod (const Point &pt) const;
      Point subtract (const Point &pt) const;
      Point operator-(const Point &pt) const;
};
```

Algorithm .68: Point class

المنارة للاستشارات
www.manaraa.com

that the method does not have parameters. Another more intuitive way to access the components is using bracket notation. e.g. `value = point[X];` Bracket notation is very natural and can be use for accessing or modifying values. Thus, the Point class is a good place to use operator overloading. There are two bracket operators the Point class can overload: the constant accessor and the accessor/mutator. The constant accessor operator is a constant method (as evidenced by the modifier `const` occurring after the method signature) that returns a copy of the value at the specified location.

The accessor/mutator bracket operator is not constant and returns a reference to the value at the specified location. References (specified with ampersands (&)) are new in C++. They are similar to pointers, but they require less notation. If a reference is returned from a method in C++, the compiler determines whether the value should be used as a reference or as the actual value. Therefore the bracket operator can work with both of the following code snippets: `point[X] = 5;` and `value = point[X];`

Since they operate exclusively on Point objects, dot product and subtract will be part of the Point class. These methods will act on the object the method was called upon and the object passed in. For efficiency, parameters can be passed as references, since references require less space than Point objects. Since these parameters will not be modified, they should be declared constant. As before, since these methods are not altering the state of the object, they should also be declared constant.

The built-in subtraction operator may be overloaded for the point class, instead of naming the subtraction method "subtract." Overloading allows more intuitive operations: `Point diff = point1 - point2;` The subtraction operator may be overloaded by merely changing the method name `operator-`. To comply with the standard definition of the subtraction operator, the method must return a copy of the result of the subtraction (a copy, because the result is temporary) and may not alter the state of the objects involved. Therefore the method should be constant, as should be the parameter. The Point result is returned by copy. The completed implementation of the Point class is shown in Algorithm .69. The constructor can set the x, y, z values through assignments (e.g. `x = x_in`), or by using initialization, as shown in the so-called member initialization list in Algorithm .69. Initialization is more efficient than assignment, because the values of the attributes are set only once. With assignment, x, y, and z would be initialized to default values before the assignment occurred. An object can access private members of another object of the same type.

3. Create the Color class. The Color class will hold red, green, blue values as before. The structure of the Color class is very similar to the Point class, and therefore needs little explanation. Again, not all C++

المنارة للاستشارات
www.manaraa.com

```
#include "Point.h"

Point::Point (double x_in, double y_in, double z_in):
              x(x_in), y(y_in), z(z_in) {}
double Point::dot_prod (const Point &pt) const {
   return x * pt.x + y * pt.y + z * pt.z;
}
Point Point::operator- (const Point &pt) const {
   return Point (x-pt.x, y-pt.y, z-pt.z);
}
```

Algorithm .69: Point class implementation

compilers will allow anonymous structures as shown in Algorithm .70. The color class should also handle the conversion from range [0.0, 1.0] to [0.0, 255.0]. The example method converts "this" object to [0,255] and returns a reference to this. After class definition is implementation, begun in Algorithm .71.

The `to_byte_range` method uses the bracket operator to access the RGB values. The bracket operator must be used on an object of the `Color` class. The "this" pointer to refer to the object the method is called on. Since `this` is a pointer, it must first be dereferenced before accessing values with the bracket operator.

4. Create the generic Object class to be the base class from which all the objects in the scene will inherit. The Object class uses the intersection structure, as well as the Point class and Color class.

   Since no methods need to be associated with intersection, it can remain a structure. This may be a good time to begin using `typedef`, but it is not necessary. See Algorithm .72. Now the Object class can be defined. Since the Object class will be the base class, its methods must be marked `virtual` to allow them to be overridden by inheriting classes. Additionally (if you wish) the methods can be "purely virtual," meaning they must be overridden by a class that has instantiateable objects. Otherwise, an error is generated. To make a method purely virtual, mark it as `virtual` and place `=0;` after the signature. See Algorithm .73.

5. Create the Scene class to handle all Scene related information, such as the 3D world width and height, and an array of scene objects. An additional method suggested is a "first visible" method that returns the first intersection by a given ray. This method belongs in Scene, because the Scene class actually contains the objects.

المنارة للاستشارات
www.manaraa.com

```
// in Color.h
typedef enum {RED, GREEN, BLUE} channel;

class Color {
   private:
      union {
         struct {
            double red, green, blue;
         };
         double rgb[3];
      };
   public:
      Color (double r=0.0, double g=0.0, double b=0.0);
      double get_red  (void) const { return red; }
      double get_green(void) const { return green; }
      double get_blue (void) const { return blue; }
      double &operator[] (int i) { return rgb[i];}
      double  operator[] (int i) const { return rgb[i];}
      Color &to_byte_range (void);
};
```

Algorithm .70: Color class definition

```
// in Color.cpp
#include "Color.h"
Color::Color (double r, double g, double b): red(r), green(g), blue(b)
    {}

Color &Color::to_byte_range (void) {
   for (int i=RED; i <= BLUE; ++i) {
      (*this)[i] = (*this)[i] * 255 + 0.5;
      if ((*this)[i] > 255) (*this)[i] = 255;
      if ((*this)[i] <   0) (*this)[i] =   0;
   }
   // Since "this" is a pointer, it must be dereferenced first
   //    in order to be converted to a reference.
   return *this;
}
```

Algorithm .71: Beginning of Color class implementation

المنارة للاستشارات
www.manaraa.com

```
// in Object.h
#include "Color.h"
#include "Point.h"

// forward declaration to allow intersection to reference Objects.
class Object;

typedef struct {
   Object *obj;
   double t;
   Point spot;
} intersection;
```

Algorithm .72: Definition of intersection structure

```
class Object {
   public:
      virtual Color get_ambient (const Point &spot) const = 0;
      virtual bool get_intersect(const Point &sp, const Point &dp,
                                 intersection &intersect) =0;
};
```

Algorithm .73: Definition of purely virtual Object methods to be overridden by child classes

Any methods or data attributes in Scene that do not depend on instance variables should be declared static. Only one copy of each static method and attribute exists and can be accessed (without an object) using the class name and scope resolution operator, e.g. `Scene::WIDTH`. Since Scene has an array of objects, it must include the Object header. See Algorithm .74. The array of Objects will actually hold Object pointers. Since Object cannot be instantiated (it has purely virtual methods), it is best to have the array be pointers to Objects.

The array of Objects will be dynamically allocated and filled with dynamically allocated elements. Therefore, the Scene class needs a destructor to perform clean up after Scene objects are deleted. A destructor is responsible for deleting any memory allocated by the object on which it is called.

Once all the allocated memory for all elements in the array has been de-allocated, the array itself must be de-allocated. Arrays must be deleted using the bracket operator to specify to the environment to free the entire array.

The `first_visible` method accepts a starting point and a direction point and returns an intersection structure holding a pointer to the closest object intersected (closest to starting point), the intersection

المنارة للاستشارات
www.manaraa.com

```
#include <stdlib.h>
#include "Object.h"
class Scene {
   private:
      static const int WIDTH = 4;
      static const int HEIGHT= 3;
      const int NUM_OBJS;
      Object **objects;

   public:
      Scene (void);
      ~Scene (void) {
         for (int i=0; i < NUM_OBJS; ++i) {
            delete objects[i];
         }
         delete [] objects;
      }
      intersection first_visible (const Point &, const Point &) const;
      static Point virtual_coord (int row, int col, int h, int w);
};
```

Algorithm .74: Scene class header file

point, and the distance t along the ray from the starting point to the intersection point. Note: $sp + dp$ could be included in the intersection structure.

The `virtual_coord` method is static, because its result does not rely on any instance variables.

Completion of the Scene class in the `Scene.cpp` file will have to wait until all scene object classes are defined.

6. Create the Sky class to inherit from the Object class. The Sky class must define values for the blue, horizon, and base red/green values. Additionally, Sky will override Object's purely virtual `get_ambient` and `get_intersect` methods. The Sky class must include the Object header file in order to access the Object class. See Algorithm .75. The implementation of the Sky class is as would be expected, as seen in Algorithm .76. The `get_ambient` method returns a new Color object by copy.

   As usual, intersection with the sky is always possible and occurs at the largest possible distance. In the future, the intersection spot should be the normalized ray from sp to dp.

7. Create the Sphere class to inherit from the Object class. The Sphere class defines radius, center, and color instance variables, a constructor, and overrides the virtual Object methods. See Algorithm .77. The constructor and ambient methods are straightforward in `Sphere.cpp`, as in Algorithm .78. The

المنارة للاستشارات
www.manaraa.com

```
// in Sky.h
#include "Object.h"

class Sky : public Object {
   private:
      double horizon, base, blue;
   public:
      Sky (double, double, double);
      virtual Color get_ambient (const Point&) const;
      virtual bool get_intersect(const Point&, const Point&,
         intersection&);
};
```

Algorithm .75: Definition of the Sky class

```
// in Sky.cpp
#include "Sky.h"

Sky::Sky (double hor, double ba, double bl) :
          horizon (hor), base (ba), blue (bl) {}
Color Sky::get_ambient (const Point &spot) const {
   double redgreen = (1-spot[Y])*horizon + base;
   return Color(redgreen, redgreen, blue);
}
bool Sky::get_intersect(const Point &sp, const Point &dp,
                        intersection &intersect) {
   intersect.t = Point::MAX;
   intersect.spot = dp;
   intersect.obj = this;
   return true;
}
```

Algorithm .76: Sky class code file

www.manaraa.com

```
// in Sphere.h
#include "Object.h"
#include <math.h>

class Sphere : public Object {
   private:
      double radius;
      Point center;
      Color color;
   public:
      Sphere (double, const Point &, const Color &);
      virtual Color get_ambient (const Point &spot) const;
      virtual bool get_intersect(const Point &sp, const Point &dp,
                                 intersection &intersect);
};
```

Algorithm .77: Sphere class definition

intersection method is much neater than the C version, due to operator overloading and access to instance variables.

Since the Point class overloads the subtraction operator, point subtraction uses much simpler notation.

The `dot_product` function is now a member of the Point class and must be called on a Point object.

The Point class's bracket operator cleans up access to the point components.

8. Create the Floor class to inherit from the Object class. The Floor class defines height and color attribute and a constructor, and it overrides the Object's virtual methods. See Algorithm .79.

   The example `get_ambient` method uses the ternary operator, which is available in C as well. See Algorithm .80. If the condition is true, the expression returns the first value. If not, the expression returns the second value.

9. Create the Scene.cpp file to complete the Scene class definition. In C++, the `new` operator is used to dynamically allocate memory. See Algorithm .81. The `first_visible` method uses static declarations to define the intersection variables. These variables do not need to be static, but declaring them static ensures that they are created only once at the beginning of the program, rather than every time `first_visible` is invoked. The are initialized once at the beginning of execution, and then `closest.t` is re-set to `MAX+1` every time the method is invoked. See Algorithm .82.

10. Create the Raytracer class to pull it all together. C++ iostreams may be used to output the image. (Of

www.manaraa.com

```
// in Sphere.cpp
#include "Sphere.h"

Sphere::Sphere (double r, const Point &cen, const Color &col) :
                radius (r), center (cen), color (col) {}
Color Sphere::get_ambient (const Point &spot) const {
   return color;
}
bool Sphere::get_intersect(const Point &sp, const Point &dp,
                               intersection &intersect) {
   double t;
   Point dp_minus_sp = dp - sp;
   Point sp_minus_c  = sp - center;
   double a = dp_minus_sp.dot_prod (dp_minus_sp);
   double b = 2.0 * dp_minus_sp.dot_prod (sp_minus_c);
   double c = sp_minus_c.dot_prod (sp_minus_c) - radius * radius;
   double discr = b*b - 4.0 * a * c;

   if (discr >= 0) {
      t = (-b - sqrt(discr))/(2.0*a);

      if (t < MIN) {
         t = (-b + sqrt (discr))/(2.0*a);
      }
      if (t > MIN) {
         for (int i=X; i <= Z; ++i) {
            intersect.spot[i] = sp[i]*(1-t)+ dp[i]*t;
         }
         intersect.t = t;
         intersect.obj = this;
         return true;
      }
   }
   return false;
}
```

Algorithm .78: Sphere class

المنارة للاستشارات
www.manaraa.com

```
#include "Object.h"
#include <math.h>

class Floor : public Object {
   private:
      double height;
      Color color1, color2;
   public:
      Floor (double, const Color &, const Color &);
      virtual Color get_ambient (const Point &spot) const;
      virtual bool get_intersect(const Point &sp, const Point &dp,
                                 intersection &intersect);
};
```

Algorithm .79: Floor class definition

```
#include "Floor.h"

Floor::Floor (double ht, const Color &c1, const Color &c2) :
              height (ht), color1(c1), color2(c2) {}

Color Floor::get_ambient (const Point &spot) const {
   int floorX = (int) floor (spot[X]);
   int floorZ = (int) floor (spot[Z]);
   return ((floorX+floorZ)%2==0)? color1 : color2;
}

bool Floor::get_intersect(const Point &sp, const Point &dp,
                          intersection &intersect) {

   if (fabs (dp[Y]-sp[Y]) < Point::MIN) return false;
   intersect.t = (height - sp[Y])/(dp[Y] - sp[Y]);
   if (intersect.t < Point::MIN) return false;
   intersect.spot[X] = sp[X] * (1.0-intersect.t) + dp[X] * intersect.t;
   intersect.spot[Y] = height;
   intersect.spot[Z] = sp[Z] * (1.0-intersect.t) + dp[Z] * intersect.t;
   intersect.obj = this;
   return true;

}
```

Algorithm .80: Floor class implementation

المنارة للاستشارات
www.manaraa.com

```
#include "Scene.h"
Scene::Scene (void) : NUM_OBJS(4) {
   int i=0;
   objects = new Object* [NUM_OBJS];
   objects[i++] = new Floor (-2, Color (1.0, 235.0/255.0, 205.0/255.0),
                         Color (139.0/255.0, 69.0/255.0, 19.0/255.0));
   objects[i++] = new Sphere (.75, Point (1.4, -1.25, -1.5),
                          Color (139.0/255.0, 0.0, 0.0));
   objects[i++] = new Sphere (.75, Point (-1.5, -.25, -2.25),
                          Color (0.0, 1.0, 1.0));
   objects[i++] = new Sky (.4, .5, 1.0);
}
```

Algorithm .81: Initialization of Objects in the Scene

```
intersection Scene::first_visible (const Point &sp, const Point &dp)
    const {
   static intersection curr = {NULL, Point::MAX+1, Point()};
   static intersection closest (curr);
   closest.t = Point::MAX+1;

   for(int i=0; i < NUM_OBJS; ++i) {
      if (objects[i]->get_intersect (sp, dp, curr) && curr.t < closest.t
         ) {
         closest = curr;
      }
   }
   return closest;
}

Point Scene::virtual_coord(int row,int col,int img_h,int img_w){
   return Point (WIDTH * (col/(double)(img_w-1)) - WIDTH/2.0,
                 HEIGHT/2.0 - HEIGHT*row/(double)(img_h-1),0.0);
}
```

Algorithm .82: Scene class's first intersected method and coordinate computation method

المنارة للاستشارات
www.manaraa.com

```
#include <iostream>
#include "Scene.h"
using std::cout;
using std::endl;

class Raytracer {

   private:
      static const int CHANNELS=3;
      static const int DEFAULT_WIDTH=800;
      static const int DEFAULT_HEIGHT=600;

      int width, height;
      Point eye;
      Scene scene;
      unsigned char *image;
      Color trace_pixel (const Point &sp, const Point &dp);

   public:
      Raytracer (int w=DEFAULT_WIDTH, int h=DEFAULT_HEIGHT);
      ~Raytracer (void) {
         delete []image;
      }
      void create_image(void);
      void trace (void);
      void output (void) const;
};
```

Algorithm .83: Raytracer class definition

course printf and fwrite will still work.) Since cout and endl are part of the std namespace, their declarations must be specified with the std name and the scope resolution operator. See Algorithm .83. Since the raytracer allocates an array of image data, the destructor must deallocate the memory when the raytracer object is deleted.

The Raytracer class constructor will initialize the dimensions of the image, the eye point location, the scene, and will dynamically allocate memory to store the resulting image data, as in Algorithm .84. The given method is not necessary, but it simplifies the creation of raytraced images. The example output method uses iostreams to output the image. (Of course, printf and fwrite are still available.) There must be only one character after the 255. For this reason $\backslash n$ is used instead of endl. In some operating systems, endl would produce 2 characters: $\backslash r \backslash n$.

The trace method invokes the static virtual_coord method. A static method in another class can

المنارة للاستشارات
www.manaraa.com

```
// in Raytracer.cpp
#include "Raytracer.h"

Raytracer::Raytracer (int w, int h) : width (w), height(h),
                      eye (0.0, 1.5, 4.0), scene() {
  image = new unsigned char [width*height*CHANNELS];
}
void Raytracer::create_image (void) {
   trace();
   output();
}
void Raytracer::output (void) const {
   cout <<"P6"<<endl<<width << " " << height << endl << "255\n";
   cout.write ((char *)image, width * height * CHANNELS);
}
Color Raytracer::trace_pixel (const Point &sp, const Point &dp) const {
   intersection intersect = scene.first_visible (sp, dp);
   return intersect.obj->get_ambient (intersect.spot);
}
```

Algorithm .84: Raytracer implementation

by called by the class name and use of the scope resolution operator. See Algorithm .85.

11. Create a main function to create and run the raytracer.

12. Create a make file to compile all the files together. Of course, the make file is not required, but it is very helpful. See Algorithm .86. The image should come out the same as before the conversion.

B.6.10 Phase 8

The eighth phase is an object-oriented program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, and shadows. The lights that cast the shadows will be child classes of the Sphere class. Required knowledge: protected attributes, static local variables, and diffuse lighting.

1. Update the Sphere class to allow for inheritance. Since a light is defined as a round object with a radius, location, and color, Sphere is the perfect base class for Light. Every attribute that Light needs to inherit must be marked `protected`, and every method Light needs to override must be virtual. See Algorithm .87.

2. Create the Light class. Light will inherit from Sphere but will always have a white color. (In the future, colored lights are an option.) Additionally, since light locations are central to the creation of shadows,

المنارة للاستشارات
www.manaraa.com

```
void Raytracer::trace (void) {
   Point dir;
   Color color;
   for (int i=0; i < height; ++i) {
      for (int j=0; j < width; ++j) {
         dir = Scene::virtual_coord (i, j, height, width);
         color = trace_pixel (eye, dir);
         color.to_byte_range ();
         for (int k=RED; k <=BLUE; ++k) {
            image[(i*width +j)*CHANNELS + k] = (unsigned char)color[k];
         }
      }
   }
}
int main (int argc, char *argv[]) {
   Raytracer *tracer;
   if (argc > 2) {
      tracer = new Raytracer (atoi(argv[1]), atoi(argv[2]));
   } else {
      tracer = new Raytracer ();
   }
   tracer->create_image ();
   delete tracer;
}
```

Algorithm .85: Raytracer main function and trace method

```
HEADERS = Raytracer.h Color.h Point.h Floor.h Sky.h Scene.h \\
          Sphere.h Object.h
OBJS    = Raytracer.o Color.o Point.o Floor.o Sky.o Scene.o Sphere.o

# Conversion rule: source_extension -> object extension
#
.cpp.o: $(HEADERS) Makefile
        g++ -c -Wall $*.cpp

a.out: $(HEADERS) $(OBJS)
        g++ -Wall $(OBJS)

clean:
        rm -f *.o core a.out *.ppm *.gch
```

Algorithm .86: Makefile

المنارة للاستشارات
www.manaraa.com

```
class Sphere : public Object {
   protected:
      double radius;
      Point center;
      Color color;
   public:
      Sphere (double, const Point &, const Color &);
      virtual Color get_ambient (const Point &spot) const;
      virtual bool get_intersect(const Point &sp, const Point &dp,
                                 intersection &intersect);
};
```

Algorithm .87: Sphere class definition

```
// in Light.h
#include "Sphere.h"

class Light : public Sphere {
   public:
      Light (double, const Point &);
      const Point & location () const { return center; }
};

// in Light.cpp
#include "Light.h"
Light::Light (double r, const Point &cen) :
             Sphere (r, cen, Color (1,1,1)) {}
```

Algorithm .88: Light class definition

and shadows are calculated, at least initially, by sending rays from intersection points to light centers, the Light class will provide a method for accessing its center point. See Algorithm .88.

3. Create an array of in the Scene class to hold all the lights in the scene. See Algorithm .89.

4. Update the destructor to clean up the lights array as well as the objects array.

5. Add lights to the scene light array. Algorithm .90 adds three lights.

6. Update `first_visible` to iterate through both the objects and the lights arrays.

7. Create a method to determine which lights are visible from a given point. Casting shadows amounts to adding extra brightness for each visible light. A light is visible from a given point if a ray from the given point toward the center of the light does not intersect any other objects before reaching the light.

المنارة للاستشارات
www.manaraa.com

```
// in Scene.h
class Scene {
   private:
      static const int WIDTH = 4;
      static const int HEIGHT= 3;
      const int NUM_OBJS;
      const int NUM_LIGHTS;
      Object **objects;
      Light  **lights;

      ~Scene (void) {
         for (int i=0; i < NUM_OBJS; ++i) {
            delete objects[i];
         }
         delete [] objects;
         for (int i=0; i < NUM_LIGHTS; ++i) {
            delete lights[i];
         }
         delete [] lights;
      }
```

Algorithm .89: Scene class definition

If one or more lights are blocked and thus not visible, their lighting effects are not added to that point, causing a shadow.

The difficulty with creating this visible lights method is returning the list of visible lights in an efficient manner. A separate array of all visible lights can be generated each time the method is called. Alternatively the reference to the light array and an array of integers representing the light elements that are visible could be returned. Another possibility is to write the method to behave in a manner similar to the string tokenizer method: if a Point is provided, the method returns the first light visible from that location. If a Point is not provided (i.e. is `NULL`), the method returns the next light visible from the last point provided, or `NULL` if no more lights are visible. Any of these approaches may be used. In the example code, notice that the Point is passed by pointer instead of reference. Pointer use is necessary, because there is no `NULL` reference. Similarly, the return value is a pointer, because if there are no more visible lights, `NULL` must be returned. See Algorithm .91.

8. Update the Object class to have a method return the diffuse color value of the object. The diffuse color value of an object is the color amount to add for each light visible to the given object. Typically, as scenes become more sophisticated with more light objects, diffuse lighting will take on a more

المنارة للاستشارات
www.manaraa.com

```
Scene::Scene (void) : NUM_OBJS(4), NUM_LIGHTS(3) {
   int i=0;
   objects = new Object* [NUM_OBJS];
   objects[i++] = new Floor (-2, Color (1.0, 235.0/255.0, 205.0/255.0),
                      Color (139.0/255.0, 69.0/255.0, 19.0/255.0));
   objects[i++] = new Sphere (.75, Point (1.4, -1.25, -1.5),
                      Color (139.0/255.0, 0.0, 0.0));
   objects[i++] = new Sphere (.75, Point (-1.5, -.25, -2.25),
                      Color (0.0, 1.0, 1.0));
   objects[i++] = new Sky (.4, .5, 1.0);

   i=0;
   lights = new Light* [NUM_LIGHTS];
   lights[i++] = new Light (.25, Point (-1.5, 2.5, 0.5));
   lights[i++] = new Light (.25, Point ( 1.5, 2.5, 0.5));
   lights[i++] = new Light (.25, Point ( 0.0, 3.5, 0.5));
}
intersection Scene::first_visible (const Point &sp,
                                   const Point &dp) const {
   static intersection curr = {NULL, Point::MAX+1, Point()};
   static intersection closest (curr);
   closest.t = Point::MAX+1;
   for(int i=0; i < NUM_OBJS; ++i) {
      if (objects[i]->get_intersect(sp, dp, curr) && curr.t < closest.t)
         {
         closest = curr;
      }
   }
   for(int i=0; i < NUM_LIGHTS; ++i) {
      if (lights[i]->get_intersect(sp, dp, curr) && curr.t < closest.t)
         {
         closest = curr;
      }
   }
   return closest;
}
```

Algorithm .90: Addition of Lights to the Scene

المنارة للاستشارات
www.manaraa.com

```
const Light * Scene::next_light_visible (const Point *pt) const {
   static Point cur_pt (0,0,0);
   static int i=0;
   intersection first;

   if (pt != NULL) {
      cur_pt = *pt;
      i=0;
   }
   while (i < NUM_LIGHTS) {
      first = first_visible (cur_pt, lights[i]->location());
      if (first.obj == lights[i++]) {
         return (Light *)first.obj;
      }
   }
   return NULL;
}
```

Algorithm .91: Method to iteratively return the next light visible from a given point

```
    virtual Color get_diffuse (const Point &spot) const = 0;
```

Algorithm .92: Addition of a diffuse color method in the Object class

prominent role and ambient lighting a less important one. For now, the diffuse color value for the objects can be the same as the ambient values. As students become more creative and wish to exert more control over the raytracer, they can modify the ambient and diffuse values of each object. See Algorithm .92.

9. Update the Sphere class, Floor class, Sky class, and Light class to override the `get_diffuse` method. Sky and Light should not react at all to lighting effects and should merely return black. For now, Sphere and Floor can just return the ambient values. See Algorithm .93.

10. Update the Object class to have a method specifying whether an object responds to lighting effects. By default, an object does respond to lighting. Update Sky and Light to override the `allows_lighting` method to return false, since neither reacts to diffuse (or specular) lighting. See Algorithm .94.

11. Update the Color class to allow channel-by-channel addition of Color objects. This addition will be used to add diffuse color onto ambient color. Since the diffuse value will be added onto the ambient value, the Color class should overload the += operator. This operator adds the values passed in to the local values. In overloading any operator, it is best to comply to the standard use of that operator.

www.manaraa.com

```
// in Sphere.h
      virtual Color get_diffuse (const Point &spot) const {
         return get_ambient (spot);
      }

// in Floor.h
      virtual Color get_diffuse (const Point &spot) const {
         return get_ambient (spot);
      }

// in Sky.h
      virtual Color get_diffuse (const Point &spot) const {
         return Color (0,0,0);
      }

// in Light.h
      virtual Color get_diffuse (const Point &) const {
         return Color (0,0,0);
      }
```

Algorithm .93: Creation of diffuse color methods in child classes

```
// in Object.h
      virtual bool allows_lighting (void) const { return true; }

// in Sky.h
      virtual bool allows_lighting (void) const { return false; }

// in Light.h
      virtual bool allows_lighting (void) const { return false; }
```

Algorithm .94: Addition of boolean method specifying whether lighting affects this object

المنارة للاستشارات
www.manaraa.com

```
// in Color.h
      const Color &operator+= (const Color &);

// in Color.cpp
const Color &Color::operator+= (const Color &color) {
   for (int i=RED; i <= BLUE; ++i) {
      (*this)[i] += color[i];
   }
   return *this;
}
```

Algorithm .95: Color add-to operator overloading

```
Color Raytracer::trace_pixel (const Point &sp, const Point &dp) const {
   intersection intersect = scene.first_visible (sp, dp);
   Color pixel = intersect.obj->get_ambient (intersect.spot);

   if (intersect.obj->allows_lighting()) {
      const Light *light = scene.next_light_visible(&intersect.spot);
      while (light) {
         pixel += intersect.obj->get_diffuse(intersect.spot);
         light = scene.next_light_visible (NULL);
      }
   }
   return pixel;
}
```

Algorithm .96: Addition of diffuse lighting to color computation

For example, C++ allows the following expression: `a = (b += c);`. Therefore, the += operator should return a reference to the result of the addition. Similarly, since `(a += b) = c;` is not valid, the reference returned by the += operator must be constant. Finally, this method alters the state of the object and cannot be declared constant. See Algorithm .95.

12. Update `Raytracer::trace_pixel` to add diffuse lighting to the ambient value for each light that is visible. Of course, lighting will be added only for objects that allow lighting. See Algorithm .96.

    The resulting image will have shadows (Figure 32. The next step is to modify the diffuse lighting based on the angle and distance to the center of the light.

www.manaraa.com

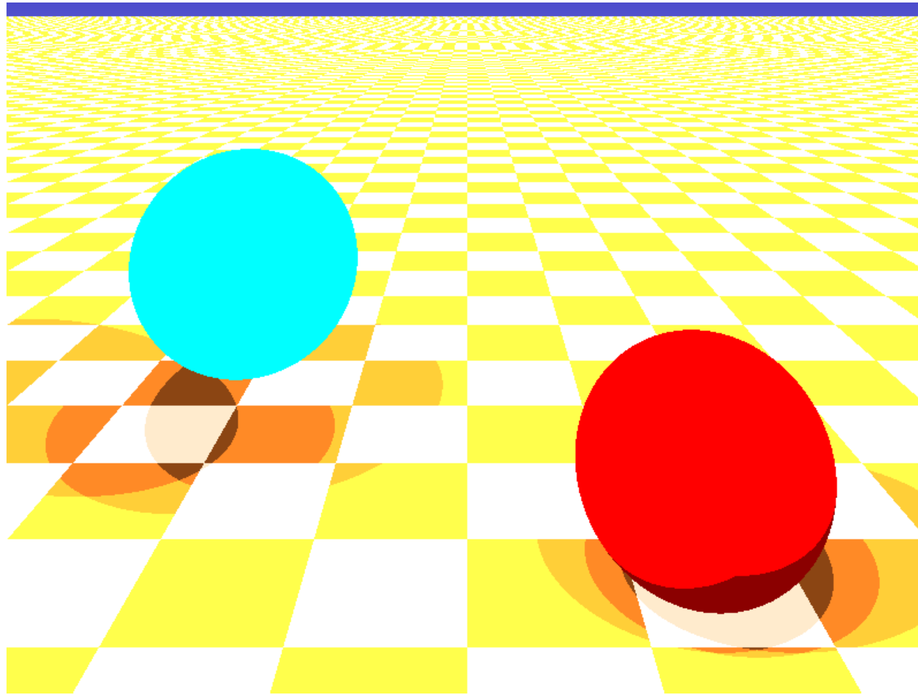


Figure 32: Shadows on a bright scene

B.6.11 Phase 9

The ninth phase is an OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, and diffuses light contribution based on light distance and surface normal. The diffuse contribution is proportional to the cosine of the angle between at any surface point, the surface normal at that point and a unit vector pointing toward the light source. The cosine of the angle to the light will be obtained using the dot product method. Required knowledge: distance formula, normalizing a vector, Sphere normal, and Lambert's cosine law.

1. Create a distance method in the Point class. This method is necessary for determining the distance to a light from a given location. The method should return the distance from the point the method is called to its parameter, also a point. The distance between two points is the Euclidean distance, given by

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{20}$$

   Using the previously-overloaded `operator-` and `dot_prod` methods in the Point class, computing distance is simple. See Algorithm .97.

المنارة للاستشارات
www.manaraa.com

```
// in Point.h
      double distance (const Point &pt) const;
      Point operator/ (double divisor) const;
      Point get_unit_vector (const Point &dp) const;

// in Point.cpp
double Point::distance (const Point &pt) const {
   Point diff = *this - pt;
   return sqrt (diff.dot_prod (diff));
}
Point Point::operator/ (double divisor) const {
   return Point (x/divisor, y/divisor, z/divisor);
}
Point Point::get_unit_vector (const Point &dp) const {
   double dist = distance (dp);
   if (fabs(dist) < MIN) {
      // if the distance is 0, return a 0-length vector
      return Point (0, 0, 0);
   } else {

      Point diff = dp - *this;
      return diff/dist;
   }
}
```

Algorithm .97: Point distance, division and unit vector

المنارة للاستشارات

www.manaraa.com

```
bool Sky::get_intersect(const Point &sp, const Point &dp,
                        intersection &intersect) {
   intersect.t = Point::MAX;
   intersect.spot = sp.get_unit_vector(dp);
   intersect.obj = this;
   return true;
}
```

Algorithm .98: Definition of Sky intersection

2. Create a method in the Point class for normalizing a vector. A normalized vector is a vector whose length is one, also known as a unit vector. This normalizing method will be used to assist in computation of the angle to the light. You may recall that

$$X \cdot Y = |X||Y|cos\theta \tag{21}$$

where θ is the angle between vectors X and Y. If vectors X and Y are normalized (i.e. unit vectors), the dot product produces simply $cos\theta$.

The unit vector of a given vector can be computed using the following formula: $unit = \frac{vector}{|vector|}$ If "distance" is the distance from the starting point to the direction point, the unit vector is computed as follows:

$$\begin{aligned} unit_x &= \frac{dp_x - sp_x}{distance} \\ unit_y &= \frac{dp_y - sp_y}{distance} \\ unit_z &= \frac{dp_z - sp_z}{distance} \end{aligned} \tag{22}$$

Since normalization is dependent upon division of the components by a scalar, it is a good idea to first define a division method by overloading `operator/`. With Point subtract and division available, normalizing is simple. One error condition to handle is the possibility that the distance between the points is zero (or close to it).

3. Update the `Sky::get_intersect` to compute the intersection point as the normalized vector from the starting point to the direction point. Until now, the sky's intersection point has been simply the direction point. However, the blueness of the sky is based on its angle. Now that vector normalization is available, this issue can be corrected. See Algorithm .98.

4. Update the object classes to return the surface normal vector at a given point for the object. The

المنارة للاستشارات
www.manaraa.com

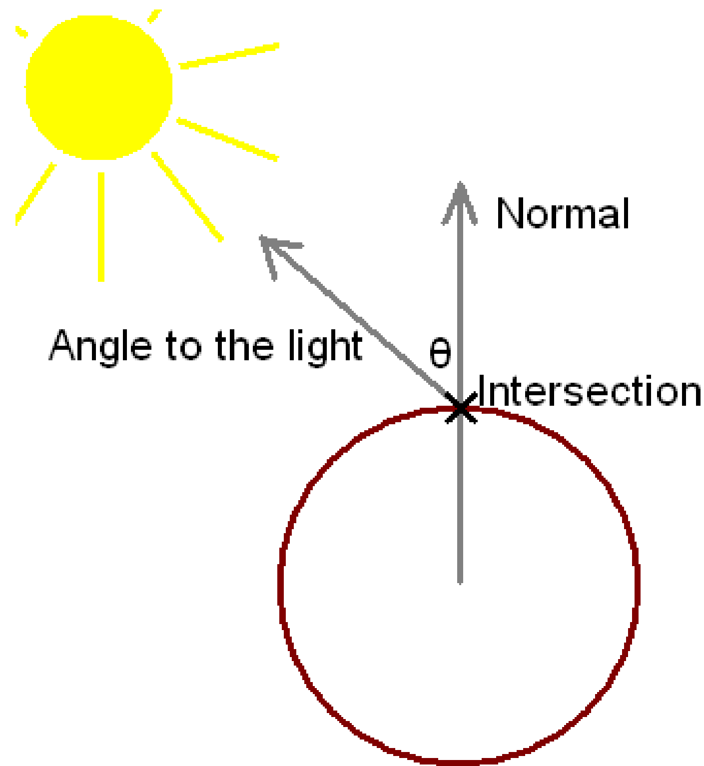


Figure 33: Angle to the light

```
virtual Point get_normal (const Point &) const = 0;
```

Algorithm .99: Purely virtual normal computation method in Object class

"normal" to a surface at a given point is a vector perpendicular to it. Normal vectors are often made unit length and thus called "unit normals." Unit surface normals are needed for the objects in order to compute the diffuse contribution to lighting at each surface point. The surface normal and the unit vector to the light provide the angle that determines the extent of the diffuse contribution. The dot product of the normal and unit vector toward the light is the cosine of the angle between them. (Figure 33).

As usual, the Object class will not directly implement the method for computing the normal and will be overridden by the inheriting classes. See Algorithm .99.

The normal for the floor is the easiest to compute since it will always be straight up: (0, 1, 0). See Algorithm .100.

Since the surface of the sphere is curved, every point on the sphere has a different surface normal.

المنارة للاستشارات
www.manaraa.com

```
// Floor get_normal
virtual Point get_normal (const Point &) const {
   static Point normal (0, 1, 0);
   return normal;
}
// Sphere get_normal
virtual Point get_normal (const Point &pt) const {
   return center.get_unit_vector (pt);
}
// Sky get_normal
virtual Point get_normal (const Point &pt) const {
   return pt; // irrelevant
}
```

Algorithm .100: Normal computation

```
Color operator* (double mult) const {
   return Color (red*mult, green*mult, blue*mult);
}
```

Algorithm .101: Color scaling method

Fortunately computation of the normal is very simple here too: a line drawn from the center of the sphere through the given point is perpendicular to the surface at that point. Therefore, to compute the unit normal at a given point for a sphere, simply subtract the center point from the intersection point.

The normals for Sky and Light are irrelevant, since diffuse lighting does not impact them. However, since the method is purely virtual, it must be overridden in all child classes. Light does not need to override it, since Sphere handles it for light. Since sky's normal could be anything, returning merely the passed in point will suffice.

5. Update the Color class to overload the multiplication operator. Since the brightness of the lighting will be scaled by the distance and angle to the light, Colors will need to be scaled, component by component, using multiplication with a double. See Algorithm .101.

6. Update `Raytracer::trace_pixel` to diminish the contribution of each light based on its distance and the cosine of the angle from the intersection point. (Lambert's cosine law says that the total radiant power observed from a "Lambertian" surface is directly proportional to the cosine of the angle between the observer's line of sight and the surface normal.) The cosine of the angle to the light will be obtained using the dot product method, since $X \cdot Y = |X||Y|cos\theta$. If the object reacts to lighting, the first visible

المنارة للاستشارات
www.manaraa.com

```
Color Raytracer::trace_pixel (const Point &sp, const Point &dp) const {
   intersection intersect = scene.first_visible (sp, dp);
   Color pixel = intersect.obj->get_ambient (intersect.spot);
   if (intersect.obj->allows_lighting()) {
      const Light *light = scene.next_light_visible(&intersect.spot);
      Point unit_dir;
      double dist, weight;
      Object *obj = intersect.obj;

      while (light) {
         unit_dir = intersect.spot.get_unit_vector (light->location());
         dist = intersect.spot.distance (light->location());
         weight = obj->get_normal (intersect.spot).dot_prod(unit_dir);
         weight /= dist;
         pixel += obj->get_diffuse(intersect.spot) * weight;
         light = scene.next_light_visible (NULL);
      }
   }
   return pixel;
}
```

Algorithm .102: Raytracer's pixel trace method

light in the scene should be located. The distance to each visible light is computed.

Once again, the dot product of the two unit vectors (the normal and the unit vector toward the light) is the cosine of the angle. The resulting cosine is stored as "weight." The cosine "weight" will now be divided by the distance to the light, since a far light does not have the intensity of a close light. Light intensity attenuates with the square of the distance, but for short distances found in raytraced scenes, linear attenuation often provides better visual results. Finally, using the overloaded $+=$ and $*$ operators, scale the diffuse value by the weight and add the result to the final pixel value.

The resulting image is shown in Figure 34.

B.6.12 Phase 10

The tenth phase is an OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent diffuse lighting contribution, and overall light reduction attenuation with distances. Although the actual attenuation of light is not necessary, the addition of a double tracking the distance light has traveled is necessary for specular lighting. (If you do not wish to attenuate light, you could instead drastically reduce the ambient contribution.) Required knowledge:

المنارة للاستشارات
www.manaraa.com

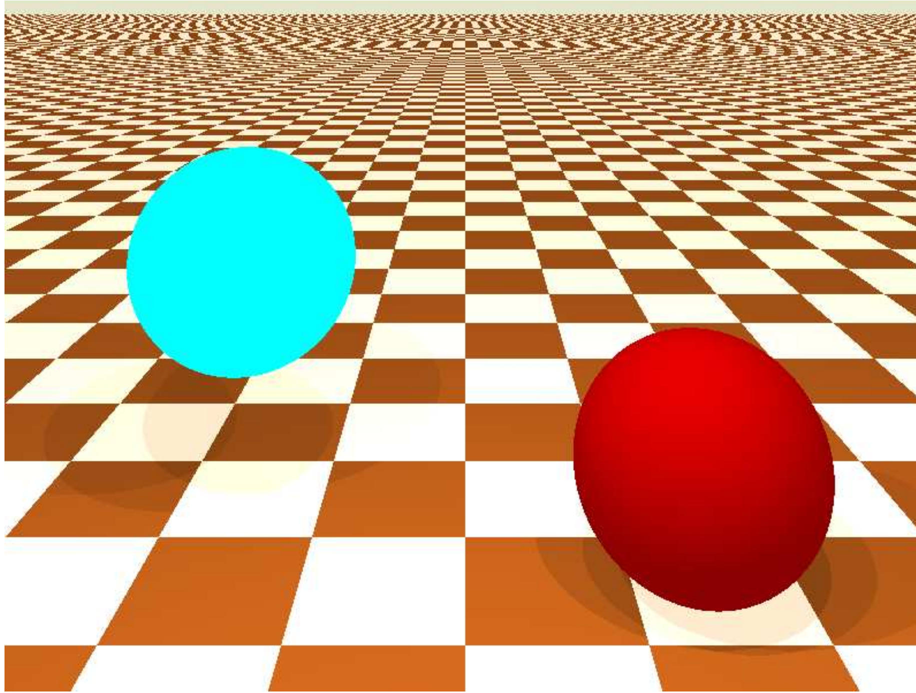


Figure 34: Lighting with diffuse component

nothing new.

1. Overload the division/assignment operator (/=) in the Color class to allow the channels of a color to be divided by a scalar double. This division is necessary for attenuating the light value with distance. Since the operator involves assignment, it is a mutator method and cannot be constant. Like previous similar operators, it should return a result of the division that cannot be modified. See Algorithm .103.

2. Light intensity will decrease with scaled distance. Create a constant float representing the scale factor on distance. A suggested starting weight is .4, and each student can alter the value as appropriate. See Algorithm .104.

3. Add a parameter to `Raytracer::trace_pixel` specifying how far the light has traveled at the instant the method is called. At this point, the traveled value is always the same (zero), because `trace_pixel` will be invoked only once for each pixel. However, with the later addition of specular lighting, which requires recursion, the distance traveled will vary. See Algorithm .105.

4. Add to the "traveled" variable the distance traveled from the starting point to the current intersection point. It is fine to update the variable for intersections with any object; however, only objects that

المنارة للاستشارات
www.manaraa.com

```
// in Color.h
      const Color &operator/= (double);

// in Color.cpp
const Color &Color::operator/= (double divisor) {
   for (int i=RED; i <= BLUE; ++i) {
      (*this)[i] /= divisor;
   }
   return *this;
}
```

Algorithm .103: Sphere class normal computation

```
// in Raytracer.h
   private:
      static const float DIST_WEIGHT=.4f;
```

Algorithm .104: Declaration of the weight that distance has in this Raytracer

```
// in Raytracer.cpp
Color Raytracer::trace_pixel (const Point &sp, const Point &dp,
                                          double traveled) const {
```

Algorithm .105: Addition of a distance-traveled-so-far parameter

المنارة للاستشارات
www.manaraa.com

```
   intersection intersect = scene.first_visible (sp, dp);
   Color pixel = intersect.obj->get_ambient (intersect.spot);

   if (intersect.obj->allows_lighting()) {
      const Light *light = scene.next_light_visible(&intersect.spot);
      Point unit_dir;
      double dist, weight;
      Object *obj = intersect.obj;

      traveled += sp.distance (intersect.spot);
      while (light) {
         unit_dir = intersect.spot.get_unit_vector (light->location());
         dist = intersect.spot.distance (light->location());
         weight = obj->get_normal (intersect.spot).dot_prod(unit_dir);

         weight /= dist;

         pixel += obj->get_diffuse(intersect.spot) * weight;
         light = scene.next_light_visible (NULL);
      }
      pixel /= DIST_WEIGHT * traveled;
   }
   return pixel;
}
```

Algorithm .106: Raytracer::trace_pixel update

respond to lighting will have their brightness reduced over distances. That is, sky and lights will always return the same brightness values. See Algorithm .106.

5. At the end of `Raytracer::trace_pixel`, divide the pixel by the scaled distance traveled.

6. Update the `Raytracer::trace` method to pass the distance the light has traveled (zero). If the resulting image is too dark, the distance traveled could be given a smaller initial value, say -1. See Algorithm .107.

   The resulting image should look more realistic. The scaling weight on the distance, the starting distance, the eye position, and the ambient and diffuse weights can all be adjusted to get more realistic images. See Figure 35.

B.6.13 Phase 11

The eleventh phase is an OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent lighting contribution, overall light

www.manaraa.com

```
void Raytracer::trace (void) {
   Point dir;
   Color color;

   for (int i=0; i < height; ++i) {
      for (int j=0; j < width; ++j) {
         dir = Scene::virtual_coord (i, j, height, width);
         color = trace_pixel (eye, dir, 0.0);
         color.to_byte_range ();
         for (int k=RED; k <=BLUE; ++k) {
            image[(i*width +j)*CHANNELS + k] = (unsigned char)color[k];
         }
      }
   }
}
```

Algorithm .107: Addition of initial distance traveled argument to the pixel trace invocation.

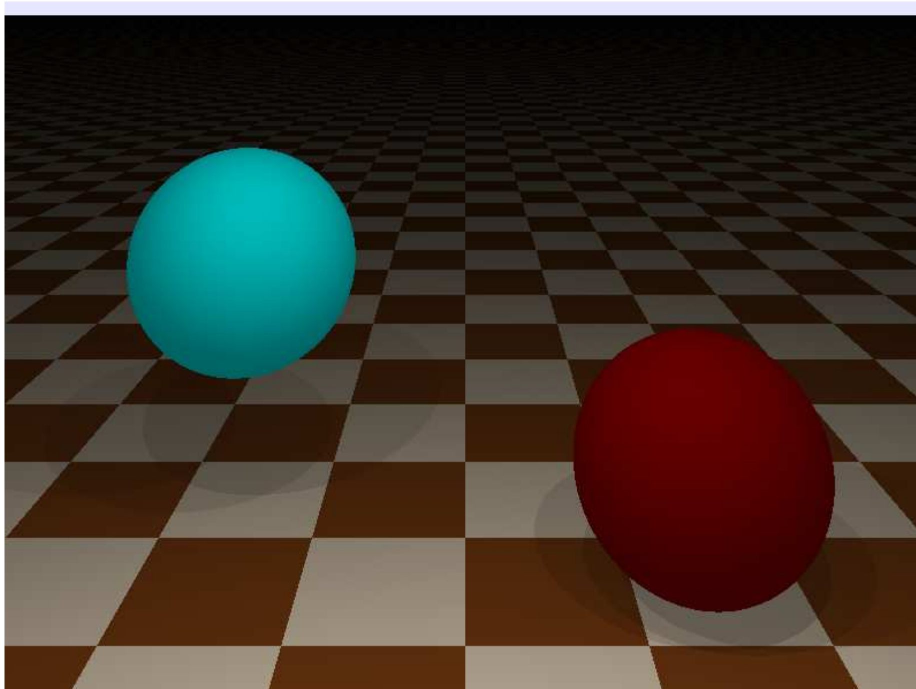


Figure 35: Light attenuation with distance

المنارة للاستشارات
www.manaraa.com

```
// in Point.h
      Point operator* (double mult) const;
      const Point &operator-= (const Point &pt);
      const Point &operator+= (const Point &pt);

// in Point.cpp
Point Point::operator* (double mult) const {
   return Point (x*mult, y*mult, z*mult);
}

const Point &Point::operator-= (const Point &sub) {
   for (int i=X; i <=Z; ++i) {
      (*this)[i] -= sub[i];
   }
   return *this;
}

const Point &Point::operator+= (const Point &add) {
   for (int i=X; i <=Z; ++i) {
      (*this)[i] += add [i];
   }
   return *this;
}
```

Algorithm .108: Point arithmetic methods

attenuation with distance, and specular reflectivity. Reflectivity is produced in a raytracer by bouncing rays off reflective objects and recursively tracing their paths. Required knowledge: bouncing a ray with the law of reflection, recursion.

1. Update the Point class to overload the multiplication operator, subtraction/assignment operator, and the addition/assignment operator to be used with bouncing rays. See Algorithm .108.

2. Update the Point class to have a method to reflect an incoming ray around the surface normal. The angle of the incoming ray to the normal (angle of incidence) and the angle of the outgoing, reflected ray (angle of reflection) are equal, with the normal bisecting them. To compute the angle of reflection, use the reversed incoming ray, $in_{reversed}$, and the normal N. Note that $reflected \cdot N = in_{reversed} \cdot N$, as desired.

$$reflected = (2N \times (in_{reversed} \cdot N)) - in_{reversed} \quad (23)$$

   The bounce method assumes that the Point object on which it is called is the normalized, reversed incoming ray. See Algorithm .109.

المنارة للاستشارات
www.manaraa.com

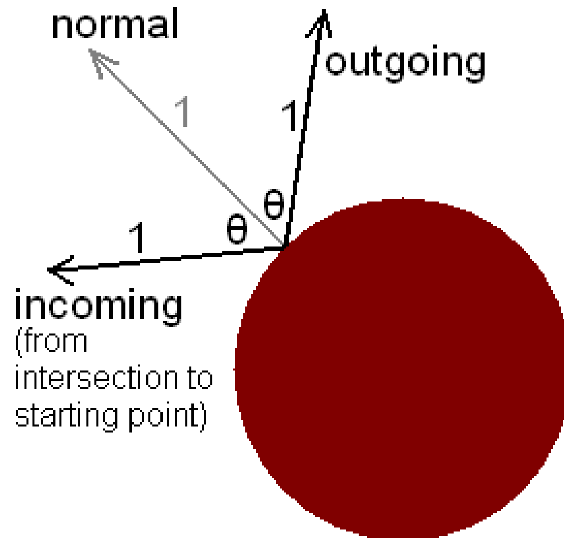


Figure 36: Vector bounce illustration

```
// in Point.h
      Point bounce (const Point &normal) const;

// in Point.cpp
Point Point::bounce (const Point &normal) const {
   Point outgoing = normal * (2 * dot_prod (normal));
   return outgoing -= *this;
}
```

Algorithm .109: Vector bounce function

المنارة للاستشارات
www.manaraa.com

```
// in Object.h
class Object {
   protected:
      Color reflect;
   public:
      Object () : reflect (0.0,0.0,0.0) {}
      Object (const Color &r) : reflect (r) {}
      const Color &reflectivity () const { return reflect; }

// no change in Sky.cpp

// in Floor.cpp
Floor::Floor (double ht, const Color &c1, const Color &c2,
   const Color &r) : Object (r), height (ht), color1(c1), color2(c2) {}

// in Sphere.cpp
Sphere::Sphere (double r, const Point &cen, const Color &col,
   const Color &ref) :
   Object (ref), radius (r), center (cen), color (col) {}

// in Light.cpp
Light::Light (double r, const Point &cen) :
              Sphere (r, cen, Color (1,1,1), Color(0,0,0)) {}
```

Algorithm .110: Addition of reflectivity attributes to Scene Objects

3. Update the objects to have reflectivity components. The amount of reflectivity an object has is a value in the range [0.0, 1.0] and represents the percentage of the object's color that is based on reflectance. The complement, $1 - reflectivity$, is the amount of ambient/diffuse light that the object's color is based on. Each rgb channel can have a different reflectance weight. Since reflectance must be represented by three doubles, the reflectivity for each object will be represented as a Color object. Reflectivity does not need to depend on the location on the object, though it could. In there examples here, reflectivity does not require the location of the intersection. The Object class can implement reflectivity for all objects. Sky will rely on Object's default constructor (and therefore will not need to be altered), while Floor, Sphere, and Light will use the other constructor for setting reflectivity to values other than 0. See Algorithm .110.

4. Update the Scene objects to now have reflectivity. It is best that all objects have at least some reflectivity, since most natural objects do. In the example, the red (`Color (139.0/255.0, 0.0, 0.0)`) sphere has extra red reflectivity, and the cyan sphere is completely reflective. See Algorithm .111.

www.manaraa.com

```
Scene::Scene (void) : NUM_OBJS(4), NUM_LIGHTS(3) {
   int i=0;
   objects = new Object* [NUM_OBJS];
   objects[i++] = new Floor (-2, Color (1.0, 235.0/255.0, 205.0/255.0),
                            Color (139.0/255.0, 69.0/255.0, 19.0/255.0),
                            Color (.02, .02, .02));
   objects[i++] = new Sphere (.75, Point (1.4, -1.25, -1.5),
                            Color (139.0/255.0, 0.0, 0.0),
                            Color (.6, .2, .2));
   objects[i++] = new Sphere (.75, Point (-1.5, -.25, -2.25),
                            Color (0.0, 1.0, 1.0),
                            Color (1.0, 1.0, 1.0));
   objects[i++] = new Sky (.4, .5, 1.0);
```

Algorithm .111: Definition of objects to have reflective components

5. Create a `Raytracer::add_diffuse` method to simplify the structure of the `Raytracer::trace_pixel` method. With all the work being done for each pixel, the `trace_pixel` method is getting long and complicated. See Algorithm .112.

6. Create a `Raytracer::compute_specular` method to compute and return the reflected color at a given point. The method of computing the specular/reflected value is to *recursively* call `trace_pixel` starting at the intersection point toward the direction of the bounce angle. Therefore, the first step in computing specular reflectivity is to compute the normal at the intersection point, as well as the unit vector from the intersection toward the starting point. These unit vectors are needed for computing the reflection vector. See Algorithm .113.

   The outgoing unit direction vector will be used in the reflection trace with the intersection point used as the starting point. Since `trace_pixel` depends on a starting point and direction point (not a unit direction vector), the starting point should be added to the unit direction vector in order to generate a direction point. Thus the starting point will be the intersection point, and the direction point is the bounced unit vector plus the intersection point.

7. Update the Color class to support the operations needed to add in specular. The specular contribution will be weighted by the object's reflectivity component. The diffuse/ambient contribution will be weighted by the complement of the object's reflectivity component, $1.0 - reflectivity$. Therefore, to compute specular, the Raytracer needs the ability to multiply colors by colors (per-component multiplication) and to find the negative or complement of a color. See Algorithm .114.

www.manaraa.com

```
// in Raytracer.h
      void add_diffuse (Color &, const intersection &) const;

// in Raytracer.cpp
void Raytracer::add_diffuse (Color &pixel,
                              const intersection &intersect) const {
   const Light *light = scene.next_light_visible(&intersect.spot);
   Point unit_dir;
   double dist, weight;
   Object *obj = intersect.obj;

   while (light) {
      unit_dir = intersect.spot.get_unit_vector (light->location());
      dist = intersect.spot.distance (light->location());
      weight = obj->get_normal (intersect.spot).dot_prod(unit_dir);

      weight /= dist;

      pixel += obj->get_diffuse(intersect.spot) * weight;
      light = scene.next_light_visible (NULL);
   }
}
```

Algorithm .112: Separation of diffuse color computation

```
Color Raytracer::compute_specular (const intersection &intersect,
                           const Point &sp, double traveled) const {

   Point normal = intersect.obj->get_normal(intersect.spot);
   Point incoming = intersect.spot.get_unit_vector (sp);
   Point outgoing = incoming.bounce (normal);
   outgoing += intersect.spot;
   Color specular = trace_pixel (intersect.spot, outgoing, traveled);
   return specular;
}
```

Algorithm .113: Computation of specular reflectivity

المنارة للاستشارات
www.manaraa.com

```
// in Color.h
      const Color &operator*= (const Color &);
      Color complement () const {
         return Color (1.0-red, 1.0-green, 1.0-blue);
      }

// in Color.cpp
const Color &Color::operator*= (const Color &color) {
   for (int i=RED; i <= BLUE; ++i) {
      (*this)[i] *= color[i];
   }
   return *this;
}
```

Algorithm .114: Color scaling methods

8. Update trace_pixel to halt tracing after the maximum distance has been traveled. Since trace_pixel will be called recursively, there must be a base case. The base case occurs when the light has traveled past the maximum allowable distance, Point::MAX. If the passed-in distance traveled exceeds Point::MAX, black is returned. See Algorithm .115.

9. Update trace_pixel to appropriately invoke the new methods and include specular reflectivity.

   The resulting image should have reflective spheres and a slightly reflective floor: Figure 37.

B.6.14 Phase 12

The twelfth phase is an OO program that creates an image of any specified size with a sky, any number of spheres, a checkered floor, shadows, distance/angle dependent diffuse lighting contribution, overall light distance attenuation, specular reflectivity, and anti-aliasing. *Anti-aliasing is the technique of minimizing the distortion artifacts known as aliasing when representing a high-resolution image at a lower resolution* The edges of the spheres, and the reflections in them have sharp, boxy edges that do not adequately represent the appropriate round shapes. Anti-aliasing techniques will smooth the boxy, pixelated edges in the images. The method of anti-aliasing used here is performing multiple traces for each pixel with pseudo-randomly jittered direction points. These multiple jittered traces are averaged to determine the final pixel value. The result of using the average of multiple, jittered traces is a blended final pixel value that smoothes transitions between colors in the image.

Covered knowledge: random number generation, a method of generating arbitrary direction points.

المنارة للاستشارات
www.manaraa.com

```
// in Raytracer.cpp
Color Raytracer::trace_pixel (const Point &sp, const Point &dp,
                                        double traveled) const {
   static const Color BLACK;
   if (traveled > Point::MAX) return BLACK;
   intersection intersect = scene.first_visible (sp, dp);
   Color pixel = intersect.obj->get_ambient (intersect.spot);

   if (intersect.obj->allows_lighting()) {
      add_diffuse (pixel, intersect);

      traveled += sp.distance (intersect.spot);
      pixel /= DIST_WEIGHT * traveled;
      Color specular = compute_specular (intersect, sp, traveled);
      specular *= intersect.obj->reflectivity();
      pixel    *= intersect.obj->reflectivity().complement();
      pixel += specular;
   }
   return pixel;
}
```

Algorithm .115: Update of trace_pixel

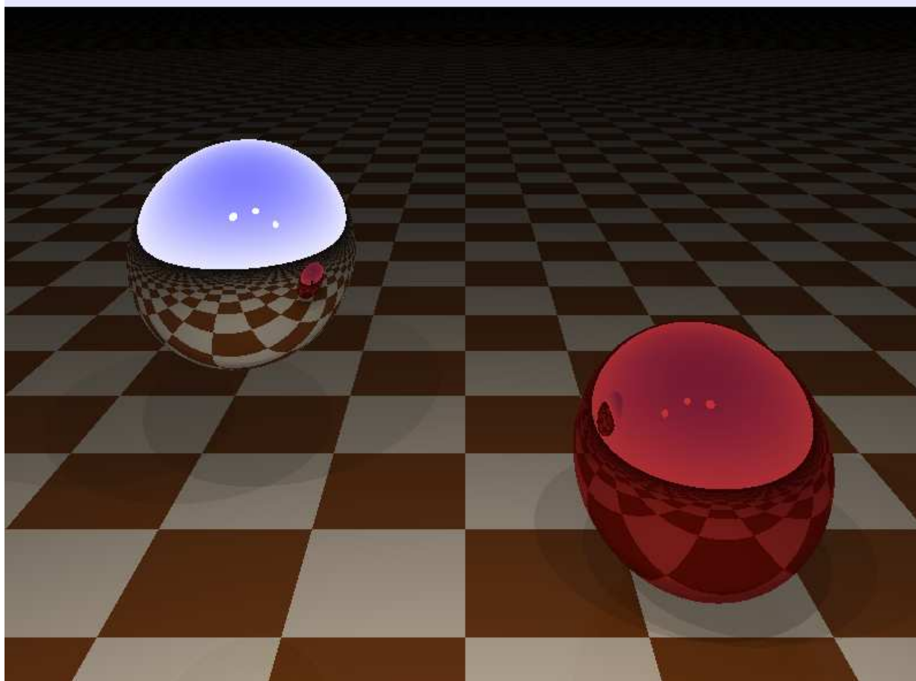


Figure 37: Scene with reflectivity

المنارة للاستشارات
www.manaraa.com

```
// in Raytracer.h
private:
      static double jitter (int base, int number);

// in Raytracer.cpp
double Raytracer::jitter (int base, int number) {
   double radical_inverse = 0.0;
   double digit_place = 1.0/base;
   while (number > 0) {
      radical_inverse += digit_place * (number%base);
      number /= base;
      digit_place *= 1.0/base;
   }
   return radical_inverse;
}
```

Algorithm .116: Pseudo-random jitter method

1. Create a `jitter` method to "randomly" generate values in the range [0.0, 1.0]. The pseudo-random numbers will be generated using the Halton sequence. These pseudo-random numbers will assist in generating jittered directions for the rays. Varying the direction points slightly will lower the aliasing effects. The code for generating the Halton sequence may be provided verbatim to students, but the algorithm is explained here for the instructor's sake:

   (a) Choose a prime base (passed-in). Typically a 2 or 3.

   (b) Find the radical inverse of the provided number in the chosen base:

      i. Convert the provided number to that base. (e.g. 4 in base 2=100)

      ii. Reverse the bits of the number. (e.g. 100 becomes 001)

      iii. Put a decimal in the front. (e.g. 001 becomes .001)

      iv. Convert back to base 10. (e.g. .001 becomes 1/8)

   The jitter method is not dependent in any way on the state of the raytracer and can therefore be static. See Algorithm .116. This converts the given number, place-by-place, to the radical inverse by computing the remainder after division by the given base and adding it at the appropriate decimal place to the final result.

   Each time through the loop, the new digit place is calculated. e.g. if `base` is 2, the digit place is $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, etc.

المنارة للاستشارات
www.manaraa.com

```
// in Raytracer.h
   private:
      static const int NUM_TRACES=8;
      Color antialias_trace(const Point &,int,int,double) const;

// in Raytracer.cpp
Color Raytracer::antialias_trace (const Point &sp, int row,
                           int col, double traveled) const {
   Point dir = Scene::virtual_coord (row, col, height, width);
   Color colorSum = trace_pixel (sp, direction, traveled);
   for (int i=1; i < NUM_TRACES; ++i) {
      dir = Scene::virtual_coord((int)(row-.5+jitter(2,i)),
                 (int)(col-.5+jitter(3,i)), height, width);
      colorSum += trace_pixel (sp, dir, traveled);
   }
   return colorSum /= NUM_TRACES;
}
```

Algorithm .117: Raytracer's anti-aliasing trace

2. Create a constant representing how many traces to perform per pixel in order to perform anti-aliasing. A suggested number is 8, but during testing, students may wish to use smaller numbers to lower runtime. See Algorithm .117.

3. Create a method to perform multiple traces for each pixel using jittered direction points. First perform the normal trace.

   Next perform the `NUM_TRACES-1` more traces at jittered direction points. Both the x and the y directions are jittered in the range [-.5, .5] using bases of 2 and 3 and the current value of the loop counter.

4. Update the `Raytracer::trace` method to call `antialias_trace` instead of `trace_pixel`. See Algorithm .118. The resulting program will be approximately `NUM_TRACES` slower than before. However the aliasing effects will be greatly reduced. See Figure 38.

B.6.15 Phase 13

The thirteenth phase is an OO program that creates an image of any specified size with a sky, any number of spheres, *any number of boxes*, a checkered floor, shadows, distance/angle dependent diffuse lighting contribution, overall light attenuation with distance, specular reflectivity, and anti-aliasing. Boxes in this raytraces are defined as 3D cubes whose sides are aligned with the x, y, and z axis. Thus, a box is defined by merely two x, y, z coordinates: a minimum xyz value designating the left, lower, far corner, and the maximum

المنارة للاستشارات
www.manaraa.com

```
void Raytracer::trace (void) {
   Color color;
   for (int i=0; i < height; ++i) {
      for (int j=0; j < width; ++j) {
         color = antialias_trace (eye, i, j, 0.0);
         color.to_byte_range ();
         for (int k=RED; k <=BLUE; ++k) {
            image[(i*width +j)*CHANNELS + k] = (unsigned char)color[k];
         }
      }
   }
}
```

Algorithm .118: Invocation of anti-aliasing trace from Raytracer loop

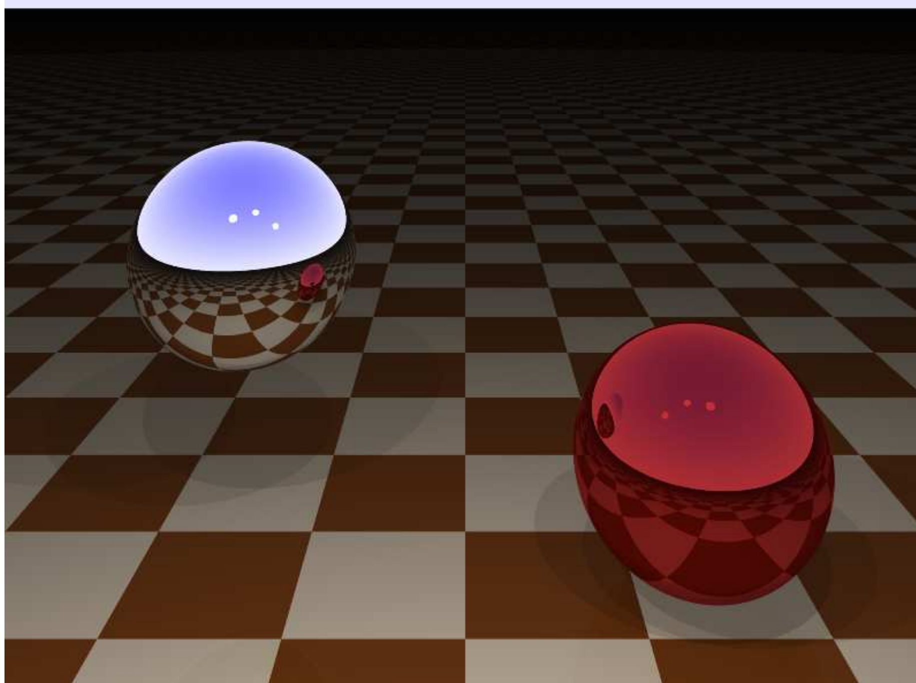


Figure 38: Anti-aliased image

المنارة للاستشارات

www.manaraa.com

```
#include <math.h>
#include "Object.h"

class Box : public Object {
   private:
      Point left_low_far;
      Point right_high_near;
      Color color;
      static bool is_equal (double d1, double d2) {
         return (fabs (d1-d2) < Point::MIN);
      }
      static void swap (double &d1, double &d2) {
         static double temp;
         temp = d1;
         d1 = d2;
         d2 = temp;
      }

   public:
      Box (const Point &, const Point &, const Color &, const Color &);
      virtual Color get_ambient (const Point &) const;
      virtual Color get_diffuse (const Point &spot) const {
         return get_ambient (spot);
      }
      virtual bool get_intersect(const Point &sp, const Point &dp,
                                 intersection &intersect);
      virtual Point get_normal (const Point &) const;
};
```

Algorithm .119: Box class definition

xyz value designating the right, upper, near corner. Required knowledge: ray-box intersection, comparison of doubles.

1. Create the `Box.h` header. The Box class is very similar to other object classes but it has two Points defining its location: a point on the left, lower, far corner, and a point on the right, upper, near corner. See Algorithm .119. In order to simplify the box intersection later, create a private, static `is_equal` method to determine whether two doubles are approximately equal, and create a private, static swap method for swapping two doubles.

2. Implement the simple parts of the Box class in `Box.cpp`. See Algorithm .120.

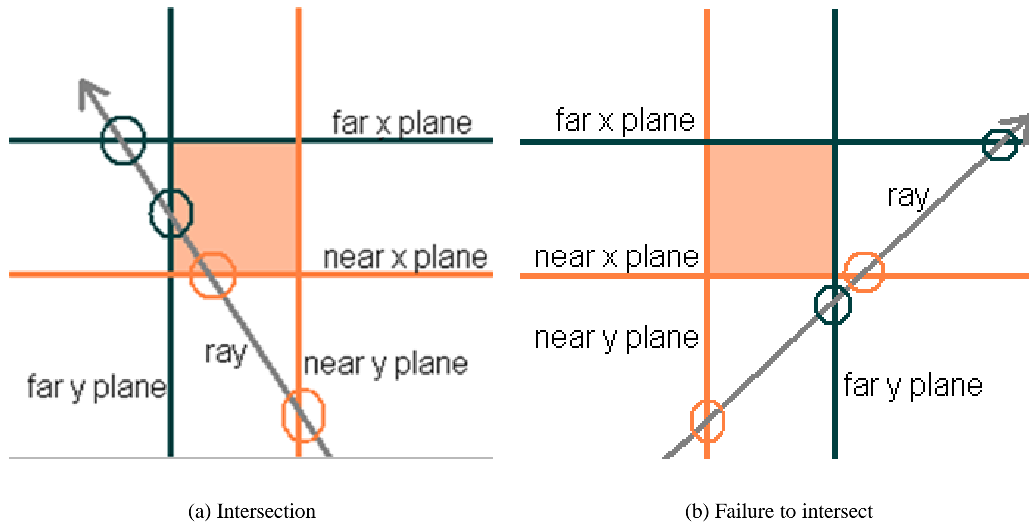
3. Implement the Box intersection method. The ray/box intersection method tests for intersection of the ray with all 6 planes defining the box: left (min x plane), right (max x plane), bottom (min y plane), top

المنارة للاستشارات
www.manaraa.com

```
#include "Box.h"

Box::Box(const Point &min, const Point &max, const Color &c,
         const Color &r) : Object (r), left_low_far(min),
         right_high_near(max), color(c){}
Color Box::get_ambient (const Point &spot) const {
   return color;
}
```

Algorithm .120: Box constructor and ambient color methods



(a) Intersection (b) Failure to intersect

Figure 39: Intersection tests

(max y plane), back (min z plane), and front (max z plane). The order in which the ray intersects these planes determines whether a ray intersects the box described by the planes. *A ray intersects a box iff the three planes of the box closest to the ray are intersected before the three planes of the box furthest from the ray* Therefore, if the ray intersects any of the far planes at distances less than any of the near planes, the ray does not intersect the box. The images demonstrating box intersection here use only x and y planes for simplicity. See Figure 39.

If the ray is parallel to an axis, say axis $c(= x, y, z)$, then the start point and direction point for the ray have equal c components. If this common value of the c component is less than min c plane or greater than the max c plane, the ray misses the box.

Since intersection occurs when all of the near sides are intersected before all of the far sides, the key

المنارة للاستشارات
www.manaraa.com

```
bool Box::get_intersect(const Point &sp, const Point &dp,
                          intersection &intersect) {

   double tnear = -Point::MAX;
   double tfar  =  Point::MAX;
   double t1, t2;
   for (int i=X; i <= Z; ++i) {
      if (is_equal(sp[i], dp[i])) {
         if (sp[i]<left_low_far[i] || sp[i]>right_high_near[i]){
            return false;
         }
      } else {
         t1 = (left_low_far[i]    - sp[i]) / (dp[i] - sp[i]);
         t2 = (right_high_near[i] - sp[i]) / (dp[i] - sp[i]);
         if (t1 > t2) swap (t1, t2);
         if (t1 > tnear) { tnear = t1; }
         if (t2 < tfar)  { tfar  = t2; }
         if (tnear > tfar || tfar < 0.0) return false;
      }
   }
   if (tnear < Point::MIN) return false;
   intersect.spot  = sp * (1-tnear);
   intersect.spot += dp * tnear;
   intersect.t = tnear;
   intersect.obj = this;
   return true;
}
```

Algorithm .121: Initialization of Box intersection computation variables

values to locate are the furthest intersection with a near plane and the closest intersection with a far plane. To accommodate this search, variables `tnear` and `tfar` will be initialized to extremely high and extremely low values, respectively. Then, the intersection of the ray within the two planes for each component will be calculated and used to update `tnear` and `tfar`. Since each component has two planes to test, there are two t variables to store the distances along the ray to the points of intersection. See Algorithm .121.

Equality of start point and direction point components must be tested to determine whether the ray is parallel to an axis. If they are equal and if the starting point's component is outside of the range of the `left_low_far` point and `right_high_near` point, there is no intersection.

If starting point's corresponding component is not out of the range, testing on this component is complete. However, if the starting point component and direction point component are not equal, we must

المنارة للاستشارات
www.manaraa.com

```
Point Box::get_normal (const Point &spot) const {
   Point normal (0,0,0);
   for (int i=X; i <=Z; ++i) {
      if (is_equal (spot[i], left_low_far[i])) {
         normal[i] = -1;
         return normal;
      }
   }
   for (int i=X; i <=Z; ++i) {
      if (is_equal (spot[i], right_high_near[i])) {
         normal[i] = 1;
         return normal;
      }
   }
   return normal;
}
```

Algorithm .122: Box class normal computation

calculate the distance along the ray to intersection with the two planes, using the same formula for calculating the `t` value used with floor intersection.

Once the two distances for this axis have been calculated, the closer distance must be stored in `t1` with the farther in `t2`. Now that the distances to the two planes have been calculated, `tnear` and `tfar` must be updated to hold the furthest near plane and the closest far plane, respectively. Once again, this calculation is to confirm that all intersections with near planes of the box occur before all intersections with far planes of the box. Otherwise, the ray misses the box.

If the loop through the x, y, and z components completes and `tfar >= 0.0`, then the plane intersections occurred in proper order (near planes before far) and the ray does indeed intersect the box.

Next, the intersection point with the nearest plane should be calculated using the `tnear` value and the information in the intersection structure variable should be updated.

4. Implement the Box `get_normal` method. Since the box is aligned with the x, y, and z axes, the normals are relatively simple: if the intersection was with the left side, the normal is (-1, 0, 0); for the right, (1, 0, 0); for the bottom, (0, -1, 0); for the top, (0, 1, 0); for the back, (0, 0, -1), for the front, (0, 0, 1). The normals for boxes are thus determined by which plane is hit, and so they can be computed in a loop. See Algorithm .122.

5. Add box objects to the scene (Algorithm .123.

المنارة للاستشارات
www.manaraa.com

```
3Scene::Scene (void) : NUM_OBJS(6), NUM_LIGHTS(3) {
   int i=0;
   objects = new Object* [NUM_OBJS];
   objects[i++]=new Floor(-2,Color(1.0,235.0/255.0, 205.0/255.0),
                     Color (139.0/255.0,69.0/255.0, 19.0/255.0),
                     Color (.02, .02, .02));
   objects[i++]=new Sphere (.75, Point (1.4, -1.25, -1.5),
                       Color (139.0/255.0, 0.0, 0.0),
                       Color (.6, .2, .2));
   objects[i++]=new Sphere (.75, Point (-1.5, -.25, -2.25),
                       Color (0.0, 1.0, 1.0),
                       Color (1.0, 1.0, 1.0));
   objects[i++]=new Sky (.4, .5, 1.0);
   objects[i++]=new Box(Point(-3, -2, -4.5), Point (3, -0.5, -3),
                        Color (0, 1, 1), Color (.01, .01, .01));
   objects[i++]=new Box(Point (-2, -2, -3.0), Point (-1, -1, -1),
                        Color (0, 1, 1), Color (.01, .01, .01));
```

Algorithm .123: Addition of boxes to the scene

The resulting image should have two boxes: Figure 40.

B.6.16 Phase 14

The fourteenth phase changes the storage of the objects in the scene to be a linked list. The purpose for this data structure alteration is support for the next phase, in which the objects in the scene are read from a file. Required knowledge: linked lists, iteration.

1. Create a linked list Node class that contains Objects. As is the case with most linked list nodes, this class will contain a pointer to the Object and a pointer to the next node. Additionally, the node class has constructors, a destructor, and an `add_after` method for inserting a passed-in node after this node. This example implementation has the node class be a private, inner class of the Object linked list class. See Algorithm .124.

   If this node has any nodes after it, the `add_after` method is a little tricky, especially if the passed in node is actually the first in a list of nodes. The passed-in node(s) should be inserted between this node and its next. Thus, if this node has a node after it, 1) find the last node in the passed-in list, 2) update the next of that last node (in the passed-in list) to point to this node's next, and 3) change this node's next to be the node passed in. See Algorithm .125.

2. Create an iterator class for stepping through the nodes in the linked list. Of course, this class is not

www.manaraa.com

```
// in ObjList.h
#include "Object.h"
#include "Light.h"
#include <stdlib.h>

class ObjList {
   private:
      class Node {
         public:
            Object *object;
            Node *next;
            Node (Object *object);
            Node (Object *object, Node *next);
            ~Node () {
               delete object;
            }
            void add_after (Node *new_node);
      };

// in ObjList.cpp
ObjList::Node::Node (Object *obj) {
   this->object = obj;
   next = NULL;
}

ObjList::Node::Node (Object *obj, Node *next) {
   this->object = obj;
   this->next = next;
}
```

Algorithm .124: Linked list node class

```
void ObjList::Node::add_after (Node *new_node) {
   // have the new node's next point to what this's next previously held
      .
   if (this->next != NULL) {
      Node *current = new_node;
      while (current->next != NULL) current = current->next;
      current->next = this->next;
   }
   this->next = new_node;
}
```

Algorithm .125: Linked list node add after method

www.manaraa.com

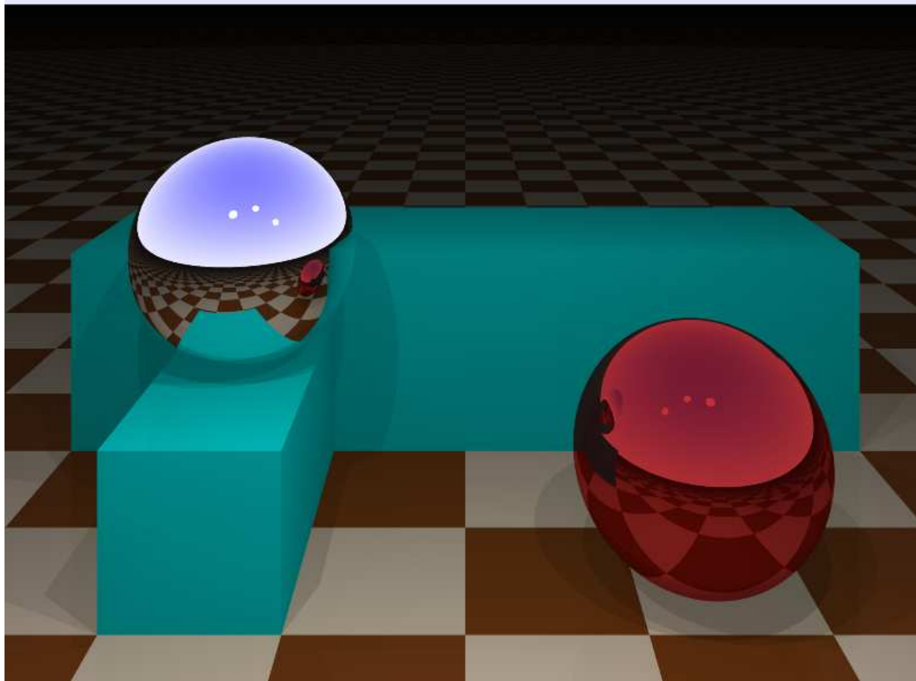


Figure 40: Scene with boxes

necessary, but it introduces iterators and also results in simpler code. The iterator should hold the node that will be returned next, and should provide operations to access the Object in that node, determine if there are any more objects, and move to the next object. Two operators typically overloaded for an iterator are `operator*` and `operator++`. The dereferencing operator is used to obtain the object in the current node, and the increment operator moves on to the next node in the list. This class can be an inner class of the Object linked list, but it must be public. See Algorithm .126.

The iterator begins with the node passed in and completes when the current node is NULL. See Algorithm .127.

Since "current" always holds the next node to be returned, the `next` method must move to the next node and then return the one before it. See Algorithm .128.

Dereferencing the iterator returns the object and the current node, and incrementing the iterator moves the current to the next node. See Algorithm .129.

3. Create the Object linked list based on the node and iterator classes. Since this linked list is custom built, it may have features that would not be available otherwise. Specifically, special handling for lights. When the raytracer iterates through objects to locate the first object intersected by a ray, all

المنارة للاستشارات
www.manaraa.com

```
// in ObjList.h
   public:
      class ObjectIter {
         private:
            Node *current;
         public:
            ObjectIter (Node *start);
            bool has_next(void) const;
            Object *next(void);
            Object *operator*(void) const;
            const ObjectIter &operator++ (void);
      };
```

Algorithm .126: Linked list iterator class

```
// in ObjList.cpp
ObjList::ObjectIter::ObjectIter (Node *start) {
   current = start;
}

bool ObjList::ObjectIter::has_next(void) const {
   return current != NULL;
}
```

Algorithm .127: Linked list iterator methods

```
Object *ObjList::ObjectIter::next() {
   Node *toReturn = current;
   current = current->next;
   return toReturn->object;
}
```

Algorithm .128: Iterator method for getting the next Object

```
Object *ObjList::ObjectIter::operator*(void) const {
   return current->object;
}
const ObjList::ObjectIter &ObjList::ObjectIter::operator++(void){
   next();
   return *this;
}
```

Algorithm .129: Iterator dereferencing and incrementing

المنارة للاستشارات
www.manaraa.com

```
// in ObjList.h
#include "Object.h"
#include "Light.h"
#include <stdlib.h>

class ObjList {
   private:
      class Node { /* . . . see above code  . . . */ };
      Node *head, *tail, *first_light;

   public:
      class ObjectIter { /* . . . see above code . . . */ };

      ObjList (void);
```

Algorithm .130: Beginning of Linked list code

```
      ~ObjList (void) {
         Node *current = head, *prev;
         while (current) {
            prev = current;
            current = current->next;
            delete prev;
         }
      }
      void add (Light *object);
      void add (Object *object);
      ObjectIter iterator (void) const;
      ObjectIter light_iterator (void) const;
};
```

Algorithm .131: Remainder of linked list class

objects and lights should be included. However, when the raytracer is determining how many lights are visible at a given spot, it should iterate only through the lights. Therefore, the example version of the linked list has lights always added to the tail of the list and other objects added to the head. In this manner, a list of all objects or a list of only lights is available within the same linked list. Thus, the linked list will have a pointer to head, tail, and the first light. See Algorithm .130.

The destructor for a linked list must delete all memory used for each node. See Algorithm .131.

The linked list can return an object iterator or a light iterator. See Algorithm .132.

4. Modify the Scene constructor to use the object linked list. The instance variables of the Scene must be changed as well to use the linked list instead of an array. Using a linked list removes the need for

المنارة للاستشارات
www.manaraa.com

```
// in ObjList.cpp
#include "ObjList.h"
ObjList::ObjList () : head(NULL), tail(NULL), first_light(NULL){}

// Lights are appended to the tail
void ObjList::add (Light *obj) {
   Node *new_node = new Node (obj);
   if (head == NULL) {
      head = tail = new_node;
   } else {
      tail->add_after (new_node);
      tail = tail->next;
   }
   if (first_light == NULL) first_light = new_node;
}

// All other objects are added at the head
void ObjList::add (Object *obj) {
   Node *new_node = new Node (obj);
   if (head == NULL) {
      head = tail = new_node;
   } else {
      new_node->add_after (head);
      head = new_node;
   }
}

ObjList::ObjectIter ObjList::iterator () const {
   return ObjList::ObjectIter (head);
}

ObjList::ObjectIter ObjList::light_iterator () const {
   return ObjList::ObjectIter (first_light);
}
```

Algorithm .132: Creation of linked list functions

المنارة للاستشارات
www.manaraa.com

```
// in Scene.h
class Scene {
   private:
      ObjList objects;

// in Scene.cpp
Scene::Scene (void) {
   objects.add(new Floor(-2,Color (1.0,235.0/255.0,205.0/255.0),
                    Color (139.0/255.0, 69.0/255.0, 19.0/255.0),
                    Color (.02, .02, .02)));
   objects.add (new Sphere (.75, Point (1.4, -1.25, -1.5),
                          Color (139.0/255.0, 0.0, 0.0),
                          Color (.6, .2, .2)));
   objects.add (new Sphere (.75, Point (-1.5, -.25, -2.25),
                          Color (0.0, 1.0, 1.0),
                          Color (1.0, 1.0, 1.0)));
   objects.add (new Sky (.4, .5, 1.0));
   objects.add(new Box(Point(-3.0,-2.0,-4.5),Point(3.0,-0.5,-3.0)
                     , Color (0, 1, 1), Color (.01, .01, .01)));
   objects.add(new Box(Point(-2.0,-2.0,-3.0),Point(-1.0,-1.0,-1.0),
                      Color (0, 1, 1), Color (.01, .01, .01)));

   objects.add (new Light (.25, Point (-1.5, 2.5, 0.5)));
   objects.add (new Light (.25, Point ( 1.5, 2.5, 0.5)));
   objects.add (new Light (.25, Point ( 0.0, 3.5, 0.5)));
}
```

Algorithm .133: Scene objects in a linked list

variables for indicating how many lights and objects the scene has. See Algorithm .133.

5. Modify `Scene::first_visible` to use the linked list. The address of the object in the current node is needed during iteration through the elements, and may be obtained by dereferencing the iterator. Calling any methods on the object obtained by iteration dereferencing must be done by dereferencing the object. See Algorithm .134.

6. Update `Scene::next_light_visible` to use the light iterator, as in Algorithm .135.

   If the use of the linked list, iterator, or operator overloading is seems too complex for students, they could instead learn about linked lists in lab and use the standard template library list.

المنارة للاستشارات
www.manaraa.com

```
intersection Scene::first_visible(const Point &sp,
                                  const Point &dp) const {
   static intersection curr = {NULL, Point::MAX+1, Point()};
   static intersection closest (curr);
   ObjList::ObjectIter iter = objects.iterator();
   closest.t = Point::MAX+1;

   for (;iter.has_next(); ++iter) {
      if ((*iter)->get_intersect (sp, dp, curr) &&
           cur.t < closest.t) {
         closest = curr;
      }
   }
   return closest;
}
```

Algorithm .134: First intersection method with a linked list

```
const Light * Scene::next_light_visible (const Point *pt) const {
   static Point cur_pt (0,0,0);
   static ObjList::ObjectIter iter(NULL);
   intersection first;

   if (pt != NULL) {
      cur_pt = *pt;
      iter = objects.light_iterator();
   }

   while (iter.has_next()) {
      first = first_visible (cur_pt, ((Light*)*iter)->location());
      if (first.obj == *iter) {
         ++iter;
         return (Light *)first.obj;
      }
      ++iter;
   }
   return NULL;
}
```

Algorithm .135: Next light method with a linked list

المنارة للاستشارات
www.manaraa.com

B.6.17 Phase 15

In the fifteenth phase, the scene to raytrace is read in from a file. Optionally, students can use the input operator `operator>>` to read in information. Required knowledge: file IO, (friend functions and the input operator).

1. Create a file specifying the scene to trace. As usual, the example will be the same scene used throughout. The format of the file should be one that is easy to understand and simple to read. The sample file specifies the object to be read and then lists each attribute (in any order) and its value. Spacing is unimportant. See Algorithm .136.

2. Update the Point class to overload the input operator. As information is read from the scene file, attributes will be read in using the input operator. e.g. `in >> radius;` will read in the next double and store it in radius. Therefore, object reading may be simplified by overloading the input operator for the Point class to allow similar notation: `in >> center;`. Again, this feature is optional. Alternatively, a "set" method may accept the file reader object and read in the values. (The C++ file reading class is `std::ifstream`).

   The tricky part about overloading the input (or output) operator is that the first object to the operator is the ifstream object and not the Point object. Therefore, if one overloaded the input operator as a member function (`std::ifstream &Point::operator>>(std::ifstream &);`), the resulting notation would be backward: `center >> in;`. Instead, the input operator must be overloaded *outside* of the Point class and accept two parameters: the ifstream object and the Point object. Unfortunately, if the operator method is outside of the class, it cannot access Point's private members. Therefore, *an overloaded input or output operator must be a friend function to the class for which the operator is being overridden*

   Once the notation is covered, overloading the input operator for the Point class is simple. Since `ifstream` objects can already read doubles, specifying how to read the three components is all that is necessary. See Algorithm .137.

3. Similarly, and for the same reasons, overload the input operator for the Color class. See Algorithm .138.

4. Create a constructor for Sphere that accepts an ifstream object from which to read in the object data. Since the attributes for Sphere can be in any order, a loop should be used to read in the four of them.

www.manaraa.com

```
Sky
horizon .4
base .5
blue 1.0

Sphere
center 1.4 -1.25 -1.5
radius .75
color .545 0 0
reflect .6 .2 .2

Sphere
center -1.5 -.25 -2.25
radius .75
color 0 0 0
reflect 1 1 1

Floor
height -2
color1 .545 .27 .0745
color2 1 .92 .8
reflect .02 .02 .02

Box
min -3 -2 -4.5
max 3 -.5 -3.0
color 0 1 1
reflect .01 .01 .01

Box
min -2 -2 -3
max -1 -1 -1
color 0 1 1
reflect .01 .01 .01

Light
center -1.5 2.5 .5
radius .25

Light
center 1.5 2.5 .5
radius .25

Light
center 0 3.5 .5
radius .25
```

Algorithm .136: Scene specification file

المنارة للاستشارات
www.manaraa.com

```
// in Point.h
class Point {
      /* . . . */
      friend std::ifstream &operator>> (std::ifstream &in,
                                        Point &pt) {
         in >> pt.x >> pt.y >> pt.z;
         return in;
      }
      /* . . . */
};
```

Algorithm .137: friend function for reading in a Point

```
// in Color.h
class Color {
      /* . . . */
      friend std::ifstream &operator>> (std::ifstream &in,
                                        Color &c) {
         in >> c.red >> c.green >> c.blue;
         return in;
      }
      /* . . . */
};
```

Algorithm .138: Friend function for reading in a color

www.manaraa.com

```
// in Sphere.h
      Sphere (std::ifstream &);

// in Sphere.cpp
Sphere::Sphere (std::ifstream &in) {
   std::string attribute;
   for (int i=0; i < 4; ++i) {
      in >> attribute;
      if (attribute == "center") {
         in >> center;
      } else if (attribute == "radius") {
         in >> radius;
      } else if (attribute == "color") {
         in >> color;
      } else if (attribute == "reflect") {
         in >> reflect;
      }
   }
}
```

Algorithm .139: Sphere constructor for reading a sphere

The first input is the string name of the next attribute. That string may be matched to one of Sphere's attributes and read in using the input operator. See Algorithm .139.

5. Similarly create constructors in Box, Floor, and Sky to read in their own attributes. See Algorithm .140, .141, and .142.

6. Update Light similarly to the others. There is one minor difference with the Light class. Since the Light class inherits from the Sphere class, the Sphere class constructor is called for the object before the Light constructor. Since the values for Light are read in the Light's constructor, they cannot be provided to the Sphere constructor at the time of invocation. Thus, the Sphere class must now provide a default constructor. See Algorithm .143.

7. Update the Scene constructor to accept a character array string specifying the input file name. Using the specified filename, create a file reader object (`std::ifstream`). Additionally, declare a method for reading in the objects. See Algorithm .144.

8. Create `Scene::read_objects`. Object reading is dependent upon each object's ability to read itself in. As long as the reader is not at the end of the file, it will read the name of the next object and call that object's constructor. See Algorithm .145.

المنارة للاستشارات
www.manaraa.com

```
// in Box.h
      Box (std::ifstream &in);
// in Box.cpp
Box::Box (std::ifstream &in) {
   std::string attribute;
   for (int i=0; i < 4; ++i) {
      in >> attribute;
      if (attribute == "min") {
         in >> left_low_far;
      } else if (attribute == "max") {
         in >> right_high_near;
      } else if (attribute == "color") {
         in >> color;
      } else if (attribute == "reflect") {
         in >> reflect;
      }
   }
}
```

Algorithm .140: Box input constructor

```
// in Floor.h
      Floor (std::ifstream &in);
// in Floor.cpp
Floor::Floor (std::ifstream &in) {
   std::string attribute;
   for (int i=0; i < 4; ++i) {
      in >> attribute;
      if (attribute == "height") {
         in >> height;
      } else if (attribute == "color1") {
         in >> color1;
      } else if (attribute == "color2") {
         in >> color2;
      } else if (attribute == "reflect") {
         in >> reflect;
      }
   }
}
```

Algorithm .141: Floor input constructor

المنارة للاستشارات
www.manaraa.com

```
// in Sky.h
      Sky (std::ifstream &in);
// in Sky.cpp
Sky::Sky (std::ifstream &in) {
   std::string attribute;
   for (int i=0; i < 3; ++i) {
      in >> attribute;
      if (attribute == "horizon") {
         in >> horizon;
      } else if (attribute == "base") {
         in >> base;
      } else if (attribute == "blue") {
         in >> blue;
      }
   }
}
```

Algorithm .142: Sky input constructor

```
// in Sphere.h
class Sphere : public Object {
   public:
      Sphere () : Object () {}

// in Light.h
      Light (std::ifstream &in);

// in Light.cpp
Light::Light (std::ifstream &in) {
   std::string attribute;
   reflect.red = reflect.green = reflect.blue = 0.0;
   color.red = color.green = color.blue = 1.0;

   for (int i=0; i < 2; ++i) {
      in >> attribute;
      if (attribute == "center") {
         in >> center;
      } else if (attribute == "radius") {
         in >> radius;
      }
   }
}
```

Algorithm .143: Light constructor for reading input

المنارة للاستشارات
www.manaraa.com

```
// in Scene.h
class Scene {
   private:
      static const int WIDTH = 4;
      static const int HEIGHT= 3;
      ObjList objects;
      void read_objects (std::ifstream &);

   public:
      Scene (char *);

// in Scene.cpp
Scene::Scene (char *file) {
   std::ifstream in (file, std::ios::in);
   read_objects (in);
}
```

Algorithm .144: Added object reading method

```
void Scene::read_objects (std::ifstream &in) {

   std::string type;
   in >> type;
   while (!in.eof()) {
      if (type == "Sphere") {
         objects.add (new Sphere (in));
      } else if (type == "Light") {
         objects.add (new Light (in));
      } else if (type == "Box") {
         objects.add (new Box (in));
      } else if (type == "Floor") {
         objects.add (new Floor (in));
      } else if (type == "Sky") {
         objects.add (new Sky (in));
      }
      in >> type;
   }
}
```

Algorithm .145: Scene object reading

المنارة للاستشارات
www.manaraa.com

```
// in Raytracer.h
class Raytracer {
   public:
      Raytracer(char *,int w=DEFAULT_WIDTH,int h=DEFAULT_HEIGHT);

// in Raytracer.cpp
Raytracer::Raytracer (char *name, int w, int h) : width (w),
                   height(h), eye (0.0, 1.5, 4.0), scene(name) {
  image = new unsigned char [width*height*CHANNELS];
}
```

Algorithm .146: Raytracer scene name parameter

```
int main (int argc, char **argv) {
   Raytracer *tracer;
   if (argc > 3) {
      tracer=new Raytracer(argv[1],atoi(argv[2]),atoi(argv[3]));
   } else if (argc > 1) {
      tracer = new Raytracer (argv[1]);
   } else {
      std::cerr << "Usage: " << argv[0]
                << " input file [width height]" << std::endl;
      return EXIT_FAILURE;
   }
   tracer->create_image ();
   delete tracer;
}
```

Algorithm .147: Main function that accepts an input file

9. Update the Raytracer constructor to accept a character array specifying the filename. See Algorithm .146.

10. Update the main function to require the filename of the scene to raytrace. If the user does not provide a filename, print an error message. See Algorithm .147. The above described raytracer should now be able to read in scenes from an input file.

المنارة للاستشارات
www.manaraa.com

Appendix C Algorithms and Data Structures Course Guide

C.1 Credits

4 (3 hour lecture and 2 hour lab)

C.2 Prerequisites

CPSC 102 or 210 with a C or better.

C.3 Course Goals

This course covers the following computer science knowledge and skills:

- Abstract data types.
- Fundamental data structures (lists, trees, heaps).
- Fundamental algorithms (searching, sorting, tree balancing, etc.).
- Ability to measure program running time and time complexity.
- Algorithm analysis and design techniques.

C.4 Course Description

Algorithms and Data Structures is based on the implementation of photon mapping: an augmentation to a raytracer that supports global illumination with diffuse color bleeding, caustics, and participating media. NOTE:As of this writing, this course has been designed but not yet taught. This guide will undoubtedly require modifications after the first attempt to teach the course, which is scheduled for Fall, 2007.

C.5 Resources

Likely the best resource for implementing and teaching photon mapping is the guide provided by Henrik Wann Jensen: *A practical guide to global illumination using raytracing and photon mapping*, in ACM SIGGRAPH 2004 Course Notes (Los Angeles, CA, August 08 - 12, 2004), SIGGRAPH '04, ACM Press, New York, NY. Not only is the guide a wealth of information, but it also lists twenty other sources for reference on photon mapping, twelve on raytracing, four on data structures, and nearly sixty other references.

المنارة للاستشارات
www.manaraa.com

C.6 Lesson Guide

C.6.1 Suggested Course Policies

1. Suggested textbook: *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss, published by Addison Wesley Publishing Company.

2. Maximum grade for simply meeting guidelines be lower than 100%.

3. Allowance of problem discussion and minor debugging with other students.

4. Prohibition of code sharing, whether verbally or electronically.

C.6.2 Description of the Assignment

This is an important opportunity for the instructor to sell students on the idea of investing time into an assignment with exciting results. Selling the assignment might include the display of images the students will be able to create, description of the technique, and explanation of the impact of this technique in industry.

C.6.3 Provision of Raytracer

While providing starter code is not typically part of the $\tau\acute{\epsilon}\chi\nu\eta$ curriculum, photon mapping is an addition to basic raytracing. Students should have their own raytracers from CS2 and may use them. However, as most students will not have implemented refraction, it is best to provide them with the compiled code of a raytracer that supports reflection, refraction, vertical and horizontal planes, and spheres to allow the creation of the Cornell Box with a reflective and refractive sphere. The provided raytracer should be instrumental to allow the addition of photon mapping.

C.6.4 Phase 1

Lighting with a photon map composed of photons (in an array) randomly-placed on a surface. The benefit of randomly-placed photons is the ability to get visual results before the entire algorithm is implemented. Required knowledge: random numbers, beginnings of runtime complexity, nearest neighbor function, illumination algorithm.

1. Create a function to randomly generate photons with x, y, z values on an object in the scene (or multiple objects in the scene).

المنارة للاستشارات
www.manaraa.com

2. Create a photon structure to store information about a photon (e.g. location, color, incoming direction).

3. Create a function to generate n photons and store them in an array.

4. Create a nearest neighbor function that searches the array (very slowly). This may be a good time to begin discussion of runtime complexity.

5. Update the raytracer to include lighting from photon mapping. For added clarity, raytrace with no lights other than the random photons.

C.6.5 Phase 2

Lighting with a photon map composed of unreflected photons from a single point light stored in an array. The runtime will still be remarkably slow. Required knowledge: photon emission algorithm.

1. Create a function to randomly generate n directions from the point light. x, y, and z should be randomly generated. The azimuth (longitude) maybe be generated randomly from $[0, 2\pi]$ and the elevation (latitude) will have to be chosen with probability proportional to is circumference. i.e., $A = 2\pi \times random(0, 1)$, $e = arcsin(2random(0, 1) - 1)$.

2. Trace each photon from the light toward the generated direction and scale the power by the number of photons emitted.

3. Store each photon at the point of ray, object intersection.

C.6.6 Phase 3

Lighting with a photon map composed of unreflected photons with nearest neighbor found via a maxheap. Required knowledge: heaps and binary heap structure, complexity

1. Create a binary maxheap class for storing n nearest neighbor photons.

2. Use the maxheap to improve the efficiency of locating the nearest neighbors.

C.6.7 Phase 4

Lighting with a photon map composed of unreflected photons stored in an unbalanced kd-tree. Required knowledge: kd-tree, complexity

المنارة للاستشارات
www.manaraa.com

1. Create a kd-tree class to store the photons for efficient nearest neighbor searches.
2. Write the nearest neighbor search. This is a good time to discuss time and space complexity.

C.6.8 Phase 5

Lighting with a photon map composed of photons that have been reflected, transmitted, or absorbed using Russian roulette to statistically determine the fate of each photon. If 50% of photons are absorbed after the first intersection, another 50% of those reflected or transmitted should be absorbed after the second intersection. Required knowledge: use of reflection and refraction, color bleed.

1. Create a function to randomly determine whether a photon is reflected, transmitted, or absorbed using a randomly generated value (Russian roulette) and the object's diffuse and specular attributes.
2. Modify the color of the photons that are reflected by the color of the surface they interacted with. This color alteration should produce a color bleed effect.

C.6.9 Phase 6

Lighting with a photon map composed of photons that are traced based on a projection map. The projection map limits photons emitted to directions that will lead to an intersection with an object. Required knowledge: matrices

1. Create a projection map from the light source consisting of little cells with boolean values indicating whether emissions in that direction will lead to intersections.
2. Update the photon generation to generate only in directions that will lead to geometry intersections.

C.6.10 Phase 7

Lighting with a photon map that is a balanced kd-tree. Required knowledge: balancing algorithm.

1. Update the kd-tree to be balanced to improve efficiency.

C.6.11 Phase 8

Lighting with a photon map and a caustic photon map. Caustics are effects caused by light passing through a refractive object or reflecting from a specular object and focusing to a strong intensity that causes

www.manaraa.com

highlights a diffuse object. A large number of photons should be emitted toward refractive surfaces to generate good caustics. A separate caustic photon map should hold the resulting photons.

1. Create a second photon map to be used as the caustic photon map.
2. Create a second projection map that allows emission toward those geometries meant to generate caustics.
3. Incorporate the caustic photon map into the lighting algorithm.

C.6.12 Phase 9

More accurate lighting effects can be achieved by using an ellipsoid neighborhood for photons obtained by compressing the sphere neighborhood in the direction of the surface normal. This modification means that photons incorrectly used at edges and in corners will be minimized. Required knowledge: compression of sphere in the direction of the normal.

C.6.13 Phase 10

More accurate lighting effects can be achieved through use of a 2D Gaussian filtering. Filtering reduces blurriness and leaked photons by increasing the weight of photons that are close to the point of interest. Required knowledge: Gaussian filters

C.6.14 Phase 11

Lighting with multiple lights and varying light types. Required knowledge: methods of emitting photons from different light shapes.

1. Create functions to emit photons from any light source type desired.
2. Add multiple lights (stored in a list) to the scene.
3. Scale the weight of the photons appropriately to account for multiple lights.

C.6.15 Phase 12

Inclusion of participating media, such as fog. This topic will likely need to be an optional challenge for more advanced students and involves the creation of a volume map and use of ray marching and a volume radiance estimate.

المنارة للاستشارات
www.manaraa.com

Appendix D   Tools and Techniques for Software Development Guide

D.1   Credits

3 (2 hour lecture and 2 hour lab)

D.2   Prerequisites

A strong knowledge of programming and some knowledge of Object-Oriented design.

D.3   Course Goals

This course covers the following computer science skills and techniques:

- Understanding of Object-Oriented programming and design.
- Understanding of advanced OO techniques: inheritance, polymorphism, abstract classes, etc.
- Understanding of advanced programming techniques: event handling, exceptions, threads, network communications.
- Intermediate-level programming skills in Java.

D.4   Course Description

This course is structured around the creation of a GUI-based, networked chess game. To do something as large as a chess game, programmers must break the task into smaller phases (in line with the principles of Extreme Programming). This chess game can be broken down into as many as 20 phases, which can then be grouped into assignments as the instructor wishes. Descriptions of each phase, including examples solutions for the instructor to follow. These solutions are in Java and are merely for guidance and not meant to imply that there are not other, better ways to write chess playing programs.

D.5   Resources

In addition to the course textbook, Sun's online Java™Tutorials are quite helpful and have up-to-date information about all aspects of Java (*http://java.sun.com/docs/books/tutorial/*).

المنارة للاستشارات
www.manaraa.com

D.6 Lesson Guide

D.6.1 Suggested Course Policies

1. Suggested text: *Object-Oriented Software Development Using Java*, Second Edition, by Xiaping Jia, 2002.

2. Maximum grade for simply meeting guidelines be lower than 100%.

3. Allowance of problem discussion and minor debugging with other students.

4. Prohibition of code sharing, whether verbally or electronically.

5. Requirement of individual, brief (5-10 minute) presentations on relevant topics. e.g., Java style conventions (pp. 25, 103, 112, 649-651), Java 2 Platform (3.1 pp. 56-58), XP (1.4.3 pp. 15-16), UML (pp. 21-25), Javadoc (6.1.4 pp. 214-216, 647, 648), jar (pp. 71-72), Packages (4.5 pp. 134-138), Wrapper Classes (4.4.9 pp. 128-130), String vs. String Buffer (4.4.8 pp. 118, Java API), replacement for goto: break and continue with statement labels (4.3.7 pp. 99-100), interning strings (4.4.8 pp. 118-122), Double Buffering (Example 5.3 p. 194), etc.

D.6.2 Selling the Assignment

This is an important opportunity for the instructor to sell students on the idea of investing time into an assignment with exciting results. Selling the assignment might include the display of images the students will be able to create, description of the technique, and explanation of the impact of this technique in industry.

D.6.3 Checkers Rules

While the final project is chess, checkers provides a simple starting point that can be transitioned smoothly into chess.

1. Played on an 8 x 8 checkered board. (International checkers is played on a 10 x 10 board with different capturing/crowning rules.)

2. Each of the two players begins with 12 round, matching pieces with the plain sides facing up. The two sets are pieces are different colors (e.g. white and red).

3. The pieces are placed on the first three rows of the player's side on the dark squares.

المنارة للاستشارات
www.manaraa.com

4. Players take turns. *A player loses when he cannot make a valid play.*

5. If a piece reaches the end row opposite from its beginning side, it is crowned (typically, a second piece is flipped over and stacked on top of it.) Crowned pieces are called “kings.” Uncrowned are called “men.”

6. Valid plays:

   (a) Man: one diagonal “move” forward one square OR one or more diagonal “jumps” forward two squares over an opponent’s piece(s) that is diagonally one square away. Jumped pieces are removed.

   (b) King: one diagonal move in any direction OR one or more diagonal jumps in any direction(s) over an opponent’s piece(s) diagonally one square away. Jumped pieces are removed.

7. If a player can jump, he must jump and continue to jump until he cannot jump or is crowned.

D.6.4 Object-Oriented Software Development

1. Purpose: “The object-oriented software development methodology aims to significantly improve current software development practice” (*Object-Oriented Software Development Using Java* 2nd Ed., X. Jia, p.2).

2. Software: “the source code as well as all the associated documentation produced during the various activities in the software development process. The documentation of software may include requirements specifications, architecture and design documents, configuration data, installation and user manuals, and so on” (Ibid, p. 4).

3. Steps of OO development: 1) identify the classes, 2) identify the attributes and behaviors of the classes, 3) identify the relationships among the classes, 4) define the class interface, then 5) implement the classes.

4. A class: a “blueprint” of an object that defines each object’s instance variables and methods. A class can have “class” variables and methods as well. There are not copies of class methods and variables for each object but instead one copy for the entire class.

5. An object: an instance of a class; a variable whose type is that of a class. An object receives a copy of every instance variable and non-static method in the class. Although objects have access to the static

المنارة للاستشارات
www.manaraa.com

(class) variables and methods, there is only one copy of each static member for the class and all its objects.

6. Creating an object and storing a reference to it:

   (a) Declare a reference variable (Java has primitive data types and reference variables) of the type of the class you want the object to be an instance of. e.g. `Person p;`

   (b) Assign p to a new instance of the object by using the keyword `new` to call its constructor. e.g. `p = new Person ("Shirley");`

7. Calling non-static methods:

   (a) Create an object of the class with the method you wish to call.

   (b) Use the object name, followed by a dot (.) before the method name and its arguments. e.g. `p.toString();`

8. Calling static methods: Use the class name, followed by a dot (.) before the method name and its arguments. e.g. `Integer.parseInt("5");`

9. UML diagramming: class represented by a box with the class name at the top, data attributes under the name, and methods under the data attributes. More details may be found in Jia's textbook, pp. 21-23.

10. Identification of the nouns in checkers. e.g. game, piece, man, king, color, board, square, player, a move, a jump, a play, window, message bar, network communication.

11. Identification of the behaviors (verbs). e.g. play, move, jump, can play, can move, can jump, crown, send play, finish play, get play, set or clear piece, set message bar, draw.

12. Identification of relationships. e.g. king is a piece, man is a piece, board has squares, square has a piece, player has pieces, pieces have color, game has a board, game has two players, player uses a board.

13. Class structures will be explored more fully after the introduction of the programming language: Java. In order to get students up to speed more quickly, initial lectures on checkers rules and OO design can be intermixed with Java material.

المنارة للاستشارات
www.manaraa.com

D.6.5 Phase 1

Phase one is the creation of an empty, GUI-based checkerboard. Required Material: Introductory Java, Java graphics, inheritance, overriding methods, invoking parent methods.

1. Introduction to Java ™(background and structure). Note: this can be sprinkled into early lectures in order to get students on track sooner.
   (a) Object-oriented programming language developed by a research team led by James Gosling at Sun Microsystems (Ibid, p. 55).
   (b) Features: OO, distributed (designed for developing distributed applications), platform independent, secure (all programs run in their own "sand boxes").
   (c) Goals of Java's design: platform independence, security, and efficiency.
   (d) Java Virtual Machine: Java is executed in two stages: compilation to byte-code followed by execution of byte-code. Execution of byte-code is done by 1) interpretation, 2) Just-in-Time compilation, OR 3) direct execution via a Java chip (in PDAs, TV set-top boxes, and cellular phones).
   (e) Java documentation is located at http://java.sun.com/. At the time of this writing, Java Standard Ed. 6 was used http://java.sun.com/javase/6/docs/api/.
   (f) Programs in Java:
      i. ALL code in Java must be part of a class.
      ii. Methods in Java programs are invoked by the objects they belong to, unless they are `static` methods. Static methods can be called with the class's name and a dot before the method.
      iii. The first method executed in a Java application is the `main`. The `main` method must be `public` to allow Java to call it, it must be `static` to allow it to be called without an object, it must not return anything (be `void`), it must be named `main`, and it must take a parameter of a String array, traditionally named `String []args`.
      iv. Java uses booleans, which are not equivalent to integers as in C/C++. Therefore, `while (i)` is not valid. Instead use `while (i!=0)`.
   (g) Creating/Executing a Basic Java Program.
      i. Create a file with the same name as the class you intend to write. e.g. `Program.java`.

www.manaraa.com

```
public class Program {
   public static void main (String []args) {
      System.out.println ("Hello, World!");
   }
}
```

Algorithm .148: Simple program

ii. In the file, create a class. If the class name matches the program file name, it can be declared `public`. (If it is not declared public, it is “package visible”: visible to everything in that package (or directory).)

iii. Create a main method. The main method is the starting point of Java programs. When a Java program is executed (`java Program`), Program’s main method is starting point of code execution. The method must be public to allow external invocation, it must be static to allow invocation without the creation of an object, it never returns a value, and it accepts a String array holding any command-line arguments. To print a message, use the `java.lang.System.out` object’s `println` method. the “ln” means that the print will be followed by a new line. See Algorithm .148.

iv. Compile the program: `javac Program.java`

v. Execute the program: `java Program` The program should print `"Hello, World!"` to the screen.

(h) Java Memory Handling

i. All primitive variables – byte (8 bits), short (16 bits), int (32 bits), long (64 bits), float (32 bits), double (64 bits), boolean (at least a bit), char (16 bits in range 0-65,535) – are stored in non-dynamic memory.

ii. All objects are in dynamic, heap memory. Declaring an object variable merely creates memory for a “reference” for an object in non-dynamic memory. To create an object, use the keyword `new` and call the constructor. e.g. `new Board (8, 60);`

iii. Arrays of any data type are objects. Arrays must be created using `new` or array initializer lists. Array access is the same as in C/C++, but Java arrays also have data attributes specifying their sizes. e.g. `int size = intArray.length;`

iv. Primitive variables are ALWAYS passed by value (a copy).

المنارة للاستشارات
www.manaraa.com

v. Objects are ALWAYS passed by reference. To make a copy of an object, use `clone`.

vi. Java handles cleaning up dynamically-allocated memory via garbage collection. Therefore, programmers never need to free memory that has been allocated for objects.

(i) Java Style Conventions:

i. Packages are named for the reverse internet domain. e.g. `edu.clemson.mypackage`. A package is a named collection of classes grouped into a directory. A file is declared part of a package with the keyword `package` followed by the package name and a semicolon. Classes that are not part of a package are part of the "unnamed package."

ii. Class and interface names have capital letters for the first letter of every word in the name. e.g. `MyNewClass`.

iii. Method and field names have lowercase first letters, followed by capitalization of the first letter of every other word in the name (the so called, "camel case"). e.g. `isKingInCheck`. Method names should be verbs and variable names should be nouns.

iv. Local variables follow the same conventions as field names, but they are typically shorter. (e.g. `buf` for buffer, `bg` for background, etc.) If a variable is used for a very short time or is a loop control variable, one-letter names are appropriate:

```
byte b, char c, double d, Exception e,
float f, int i, int j, int k, String s
```

v. Parameters: if a parameter's sole purpose is setting a field, it is appropriate to name it the same thing as the field, differentiating them by `this`. e.g. `this.width = width;`

vi. Constants ("final variables") are all capital letters with underscores separating the words. e.g. `NUM_SQUARES`

(j) Java Graphics:

i. The graphic components for the checkers/chess game will be from the javax.swing package, which "provides a set of 'lightweight' (all-Java language) components that, to the maximum degree possible, work the same on all platforms" (Java API). The Checker/chess game will use JComponents and their child classes.

ii. Painting of colors or images will depend on the java.awt package, which "contains all of the classes for creating user interfaces and for painting graphics and images" (Java API).

المنارة للاستشارات
www.manaraa.com

iii. Event handling will rely on the java.awt.event package, which "provides interfaces and classes for dealing with different types of events fired by AWT components" (Java API).

2. Description of necessary graphics components: JFrame and JPanel. The Board will be a JPanel held by a JFrame.

   (a) A frame, implemented as an instance of the JFrame class, is a window that typically has decorations such as a border, a title, and buttons for closing and iconifying the window. Applications with a GUI typically use at least one frame.

   (b) A panel is a general-purpose container for lightweight components. Our example panel will be placed on the frame and will hold the checkerboard surface. Unlike JFrames, JPanels are double-buffered by default, and the pixel in the upper, left-hand corner (0,0) is below the border and therefore visible.

3. Identification of classes needed (the nouns): Board, GameSquare, Panel, Frame. The JFrame and JPanel classes are already written for us. The Board class will hold the colored squares and should fill the entire JPanel. Each GameSquare class object will be an individual square on the checkerboard and should know its location and how to draw itself.

4. Design the GameSquare class

   (a) Attributes: row, column, color, and width. Since all the squares will be the same size (and square), width can be a `static` class variable.

   (b) Behaviors: draw.

   (c) Relationships: Board has GameSquares.

5. Design the Board class

   (a) Attributes: width, a list of squares, and a count of the numbers of squares across (typically 8 for American checkers and 10 for international).

   (b) Behaviors: creation of squares, invoking draw methods for squares.

   (c) Relationships: Board has GameSquares, Board is a JPanel. Since the board will fill the entire space of the JPanel and should control how the JPanel is drawn, Board will extend the JPanel class. Although Board could inherit from JFrame (instead of JPanel) and perform all drawing

المنارة للاستشارات
www.manaraa.com

needed by overriding the JFrame paint method, JFrame does not perform double buffering and has offset problems not present in JPanel. JPanel uses double buffers, and 0, 0 is the first draw-able location in the upper left-hand corner. Thus, the JFrame will hold a Board which is a JPanel.

6. Discussion of the Color class.

   (a) The color class has public static Color objects of many typical colors.

   (b) If you want to specify a color, use a constructor to specify the rgb values.

7. Discussion of how a Board can be a JPanel: inheritance.

   (a) Inheritance is an extension relationship between two classes. The subclass extends (i.e. is a child of) the superclass.

   (b) Inheritance models the "is-a" relationship. e.g. If the class "Student" extends "Person," Student is a Person. Student is a specialization of the general Person class.

   (c) The sub class receives all of the public and `protected` behaviors and attributes.

   (d) The sub class can add its own behaviors and attributes, as well as "override" the behaviors (methods) of the super class. A method in a super class is overridden by the sub class if the subclass has a method with the identical signature and return type (with different functionality). e.g., if the Person class has a `toString()` method that returns the String "Person", Student may write a method `toString()` to instead return the String "Student."

   (e) Every class implicitly extends the superclass Object.

8. Create the GameSquare class. The attributes are a Color, a row, a column, and a static width. The constructor should initialize the instance variables, and a static method can set the width. Instance variables and class variables are usually private. See Algorithm .149. The behavior the GameSquare needs to implement is the ability to draw itself. The square is drawn as a rectangle filled with the specified color with equal sides that starts at the appropriate row and column, which are based on the width of each square. This drawing with be done by means of a Graphics2D object that will be passed to GameSquare's draw method. "This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java™platform" (Java API http://java.sun.com/javase/6/docs/api/). The Graphics class has a method

المنارة للاستشارات
www.manaraa.com

```
import java.awt.*; // for Color and Graphics2D

public class GameSquare {
   private Color bgcolor;
   private int row, col;
   private static int width;

   public GameSquare (Color bgcolor, int row, int col) {
      this.bgcolor = bgcolor;
      this.row = row;
      this.col = col;
   }
   public static void setWidth (int width) {
      GameSquare.width = width;
   }
```

Algorithm .149: Beginning of the GameSquare class

```
   public void draw (Graphics2D g) {
      g.setPaint(bgcolor);
      g.fillRect(col*width, row*width, width, width);
   }
}
```

Algorithm .150: Draw method for creating the square in the appropriate location

`fillRect` that creates a solid-colored rectangle starting at the specified location that is the given width and height. The color of the rectangle must be set first via the Graphics2D's `setPaint` method. See Algorithm .150.

9. Create the Board class.

   (a) Make the Board extend JPanel.

   (b) Create all the instance and class variables the Board class needs: width of each square, the number of squares across the board, and a two-dimensional array of the squares. As usual, the fields are private. The number of squares across will not change during execution and can therefore be a constant. Constants in Java are specified by the keyword `final`. Additionally, since the value is the same for any instance of the board, the number of squares can also be static.

   (c) Create a method to initialize the squares. This method will be invoked by the Board constructor and can therefore be private. It will handle creating the colors of the squares (a light and a dark),

المنارة للاستشارات
www.manaraa.com

creating each square, and storing its reference in an array. Since `GameSquare.setWidth` is static, it can be invoked using the class name.

To create each of the 64 squares, there must be two nested for loops that get their values from the size of the array using length. Since it is a two-dimensional array, `length` stores the number of rows. To determine the number of columns per row, use the `length` variable of any individual row. e.g. `squares[0].length`.

Since the board is checkered, the colors must alternate. Checkering may be achieved by determining whether the sum of the row number and column number is even or odd. The color of the square is light for even sums and dark for odd.

(d) Create the Board constructor to initialize its variables and create the JFrame object that will hold this Board. The JFrame with and height can be computed up front. See Algorithm .151.

(e) Set the JFrame's default close operation to exit. This setting means that when a user closes the JFrame, the application exists.

(f) Invoke the `buildGameSquares` method to create the GameSquare objects.

(g) Add this Board to the frame and make the frame visible.

(h) Overwrite JPanel's paint method to call each GameSquare's draw method in a loop using the passed in Graphics object. The draw method in the GameSquare class must have a `Graphics2D` object passed to it, but the paint method is passed a `Graphics` object. Fortunately, the passed-in object is truly a `Graphics2D` object and can be cast appropriately.

JPanel (and thus Board) inherits `paint` from `JComponent`. The paint method is invoked by Swing to draw components. As noted in the Java API, "applications should not invoke paint directly, but should instead use the repaint method to schedule the component for redrawing." Since the JPanel class already has a paint method that renders the JPanel over top of the JFrame, its paint method should be invoked first before the Board's modifications are done. To call a parent class's method in Java, use the keyword `super`.

(i) Create the main method to create the Board object. See Algorithm .151 and Figure 41.

D.6.6 Phase 2

Phase two is the creation of a GUI-based checkerboard set correctly with colored, filled circles. Required material: Graphics2D drawing tools

المنارة للاستشارات
www.manaraa.com

```
import javax.swing.*; // For the JFrame and JPanel classes

public class CheckerBoard extends JPanel {
   private int squareWidth = 60;
   private static final int NUM_ACROSS = 8;
   private GameSquare squares[][];

   private void buildGameSquares() {
      Color dark  = new Color (21, 106, 89);
      Color light = new Color (255, 255, 213);
      squares = new GameSquare [NUM_ACROSS][NUM_ACROSS];
      GameSquare.setWidth (squareWidth);
      for (int i = 0; i < squares.length; ++i) {
         for (int j = 0; j < squares[i].length; ++j) {
            if ((i+j)%2 == 0) {
               squares[i][j] = new GameSquare(light, i, j);
            } else {
               squares[i][j] = new GameSquare(dark, i, j);
            }
         }
      }
   }
   public CheckerBoard () {
      int width = squareWidth*NUM_ACROSS;
      int height = squareWidth*NUM_ACROSS;
      JFrame frame = new JFrame("Checkers");
      frame.setSize (width, height);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      buildGameSquares();
      frame.add(this);
      frame.setVisible(true);
   }
   public void paint(Graphics g) {
      super.paint(g);
      for (int i = 0; i < squares.length; ++i) {
         for (int j = 0; j < squares[i].length; ++j) {
            squares[i][j].draw((Graphics2D)g);
         }
      }
   }
   public static void main (String []args) {
      new CheckerBoard();
   }
}
```

Algorithm .151: CheckerBoard class

المنارة للاستشارات
www.manaraa.com

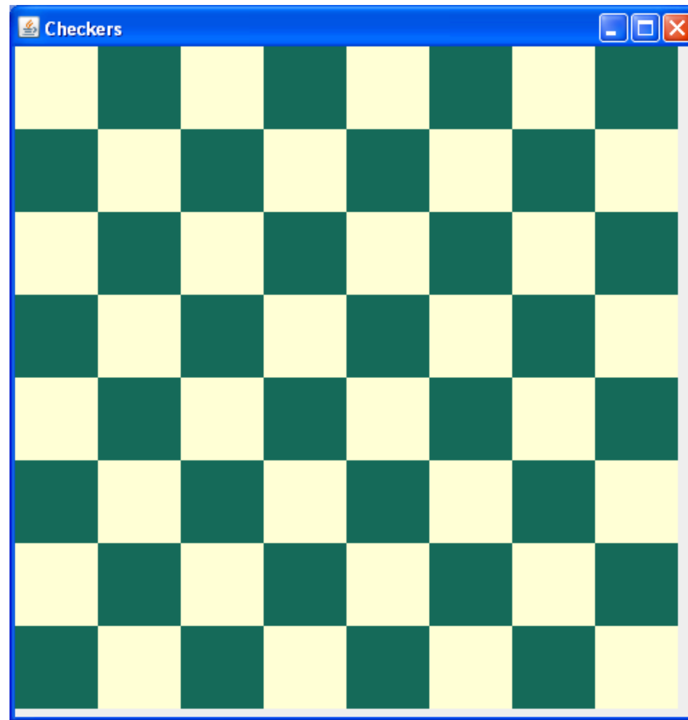


Figure 41: Checkerboard

1. Identification of new classes needed: Piece and Checkers. Checkers will be in charge of placing the pieces on the board. In the future, Checkers will also hold the JFrame containing the board.

2. Design of the Piece class:

   (a) Attributes: color, row, column, `static` width, `static` square width.

   (b) Behaviors: draw, setup and accessor methods.

   (c) Relationships: GameSquare has a Piece.

3. Design of the checkers class:

   (a) Attributes: Board, (width, number of squares per side).

   (b) Behaviors: initialization, setup pieces.

   (c) Relationships: Checkers has a Board.

4. Updated design of the GameSquare class:

   (a) New Attributes: Piece.

المنارة للاستشارات
www.manaraa.com

```
import java.awt.*;

public class Piece {
   private Color color;
   private int row, col;
   private static int squareWidth, width;

   public Piece (Color color) {
      this.color = color;
      row = -1;
      col = -1;
   }
   public Piece (Color color, int row, int col) {
      this.color = color;
      this.row = row;
      this.col = col;
   }
   public void setLocation (int row, int col) {
      this.row = row;
      this.col = col;
   }
   public static void setSquareWidth (int sqWidth) {
      squareWidth = sqWidth;
      width = (int)(squareWidth * 0.8f);
   }
```

Algorithm .152: Piece class instance variables and accessor/mutator methods

   (b) New Behaviors: Accessing, drawing, and updating Piece.

   (c) New Relationships: GameSquare has a Piece.

5. Updated design of the Board class:

   (a) New Behaviors: Placing a Piece at a specified location.

   (b) New Relationships: Checkers has a Board.

6. Create the Piece class.

   (a) Declare all needed attributes, constructors, and get/set methods. The diameter of the piece is 80% of the square's width. See Algorithm .152.

   (b) Create a draw method to generate the graphical representation of the piece. Call the Graphics object's fillOval method to draw a filled, colored circle centered on the piece's location with a

للاستشارات المنارة
www.manaraa.com

```
    public void draw (Graphics2D g) {
        int left = (int)(col*squareWidth + squareWidth*0.1f);
        int top  = (int)(row*squareWidth + squareWidth*0.1f);
        g.setPaint(color);
        g.fillOval (left, top, width, width);
    }
}
```

Algorithm .153: Piece's draw method

width 80% of the square. The diameter of the piece is already set to 80% in the `width` variable. Drawing of the circle must begin 10See Algorithm .153.

7. Update the GameSquare class to hold a reference to the Piece on that GameSquare, allow access to the Piece, draw it whenever the GameSquare object is drawn, and update the Piece width whenever the GameSquare width changes. See Algorithm .154.

8. Create a method in Board to allow a piece to be placed on a given GameSquare, as in Algorithm .155.

9. Update the Board constructor to accept as parameters the values of the square width and the number of squares across. Since checkers will now be responsible for creating the Board, checkers should be able to set such values. See Algorithm .156.

10. Define the checkers class to create the Board with squares of the appropriate width and number across and to place the pieces on the Board. The colors of the pieces are slightly darkened red and white. Later, when the pieces are made to have a three-dimensional appearance, they cannot be as bright as full red and white. See Algorithm .157.

    The `setOutPieces` method will place twelve dark and twelve light pieces on every other square of the board (i.e. the dark squares). The light pieces are on the top three rows, and the dark pieces are on the bottom three rows. The newly created pieces will be placed on the board at the specified row and column using the Board's `setPiece` method. After all the pieces are placed, the Board needs to be redrawn to reflect the added pieces. Since the `paint` method cannot be called directly, we instead call the `repaint()` method to schedule a call to `paint`. See Algorithm .158.

    Create a main in the Checkers class to create the Checkers object. See Algorithm .159 and Figure 42.

www.manaraa.com

```
// in GameSquare.java
import java.awt.*;

public class GameSquare {
   private Color bgcolor;
   private int row, col;
   private Piece piece;
   private static int width;

   public GameSquare (Color bgcolor, int row, int col) {
      this.bgcolor = bgcolor;
      this.row = row;
      this.col = col;
      piece = null;
   }
   public void setPiece (Piece piece) {
      this.piece = piece;
      if (piece != null) piece.setLocation (row, col);
   }
   public Piece getPiece () {
      return piece;
   }
   public void draw (Graphics2D g) {
      g.setPaint(bgcolor);
      g.fillRect (col*width, row*width, width, width);
      if (piece != null) {
         piece.draw (g);
      }
   }
   public static void setWidth (int width) {
      GameSquare.width = width;
      Piece.setSquareWidth (width);
   }
}
```

Algorithm .154: Square class

```
// in CheckerBoard.java
   public void setPiece (int row, int col, Piece piece) {
      squares[row][col].setPiece(piece);
   }
```

Algorithm .155: CheckerBoard Piece placement method

المنارة للاستشارات
www.manaraa.com

```
public class CheckerBoard extends JPanel {
   private int squareWidth;
   private int numAcross;
   private GameSquare squares[][];

   public CheckerBoard (int squareWidth, int numAcross) {
      this.squareWidth = squareWidth;
      this.numAcross = numAcross;
      int width = squareWidth*numAcross;
      int height = squareWidth*numAcross;

      JFrame frame = new JFrame("Checkers");
      frame.setSize (width, height);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      buildGameSquares();
      frame.add(this);
      frame.setVisible(true);
   }
```

Algorithm .156: Updated CheckerBoard constructor

```
import java.awt.Color;

public class Checkers {
   private CheckerBoard board;
   private int squareWidth=60;
   private int numAcross=8;

   public Checkers() {
      board = new CheckerBoard(squareWidth, numAcross);
      setOutPieces(new Color (230,230,230), new Color(210,0,0));
   }
```

Algorithm .157: Beginning of Checkers class

المنارة للاستشارات
www.manaraa.com

```
private void setOutPieces (Color light, Color dark) {
   for (int row=0; row < 3; ++row) {
      for (int col=(row+1)%2,int cnt=0;cnt <4; col+=2,++cnt){
         board.setPiece (i, j, new Piece (light));
      }
   }
   for (int row=5; row < 8; ++row) {
      for (int col=(row+1)%2,int cnt=0;cnt <4; col+=2,++cnt){
         board.setPiece (i, j, new Piece (dark));
      }
   }
   board.repaint();
}
```

Algorithm .158: Piece placement method

```
   public static void main (String []args) {
      Checkers checkers = new Checkers();
   }
}
```

Algorithm .159: Checkers class instantiation

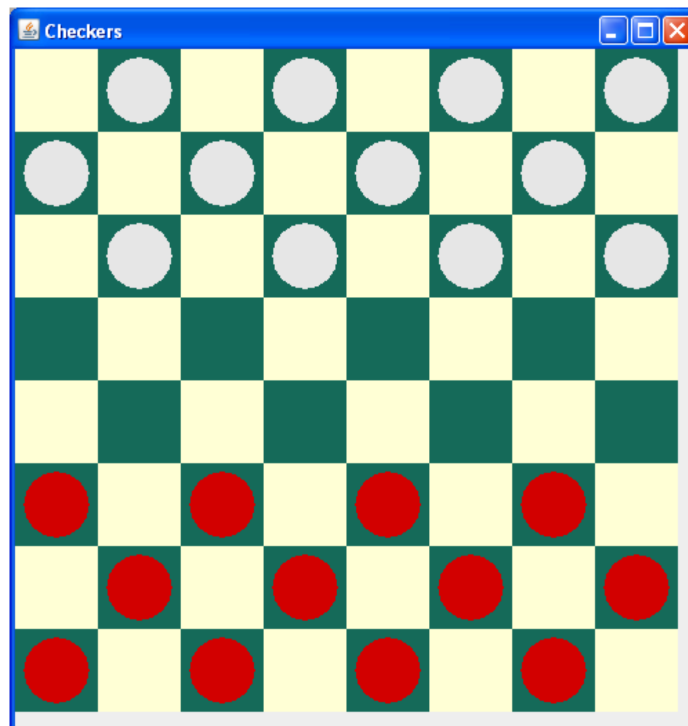


Figure 42: Checkerboard with immobile pieces

المنارة للاستشارات
www.manaraa.com

```
public void draw (Graphics2D g) {
   int left = (int)(col*squareWidth + squareWidth*0.1f);
   int top  = (int)(row*squareWidth + squareWidth*0.1f);
   int innerLeft = (int)(left + squareWidth * 0.1f);
   int innerTop = (int)(top + squareWidth * 0.1f);

   g.setPaint(color);
   g.fillOval (left, top, width, width);
```

Algorithm .160: Beginning of the Piece class draw method

```
   g.setPaint(Color.black);
   g.drawOval (innerLeft, innerTop, (int)(squareWidth*.6),
                                    (int)(squareWidth*.6));
}
```

Algorithm .161: End of draw method

D.6.7 Phase 3

Phase three is the creation of a GUI-based checkerboard set correctly with smooth, 3D-looking pieces. Required knowledge: Concept of anti-aliasing, gradient paint tool

1. Update the Piece class's draw method to draw a smaller, thicker ring over top of the filled circle.

   (a) Determine the starting point for the inner ring's left and top, as in Algorithm .160.

   (b) Change the color and draw an oval starting at the specified innerLeft and innerTop that is 60% the width of the square. See Algorithm .161 and Figure 43.

   (c) The checkers still look fake. Instead of a thin, black ring, make a slightly thicker ring with shading from dark to light, giving the checker the appearance of having an inset groove. To do this, we must make an object of the `GradientPaint` class that shades diagonally from a dark version of the checker's color to a light version of the checkers color. GradientPaint needs to know the starting location of the gradient, the ending location, and the two colors to use for the gradient. We already have the starting point: innerLeft and innerTop. The other point is similar. See Algorithm .162.

   (d) Once the filled circle is drawn, create the GradientPaint object and set it to be the current paint. The dark color and the light color for the gradient will be computed by Color's built-in `darker()` and `brighter()` methods. See Algorithm .163.

www.manaraa.com

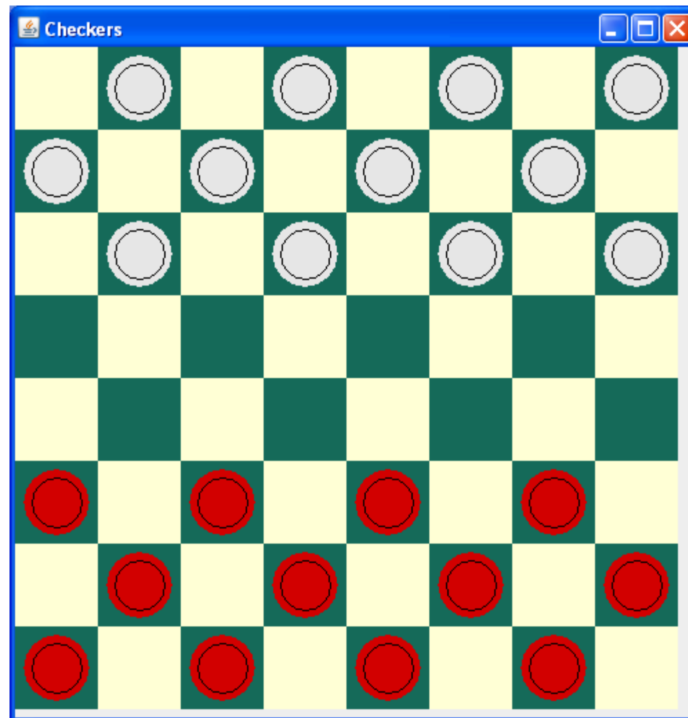


Figure 43: Checkerboard with inner circle

```
public void draw (Graphics2D g) {
   int left = (int)(col*squareWidth + squareWidth*0.1f);
   int top  = (int)(row*squareWidth + squareWidth*0.1f);
   int innerLeft = (int)(left + squareWidth * 0.1f);
   int innerTop = (int)(top + squareWidth * 0.1f);
   int innerRight =(int)((col+1)*squareWidth-squareWidth*0.3f);
   int innerBottom=(int)((row+1)*squareWidth-squareWidth*0.3f);

   g.setPaint(color);
   g.fillOval (left, top, width, width);
```

Algorithm .162: Beginning of updated draw method

```
   GradientPaint shade = new GradientPaint(innerLeft, innerTop,
   color.darker(), innerRight, innerBottom, color.brighter());
   g.setPaint(shade);
```

Algorithm .163: Gradient Paint

المنارة للاستشارات
www.manaraa.com

```
        g.setStroke(new BasicStroke(2.0f));
        g.drawOval (innerLeft, innerTop, (int)(squareWidth*.6),
                                         (int)(squareWidth*.6));
}
```

Algorithm .164: Creation of inset circle

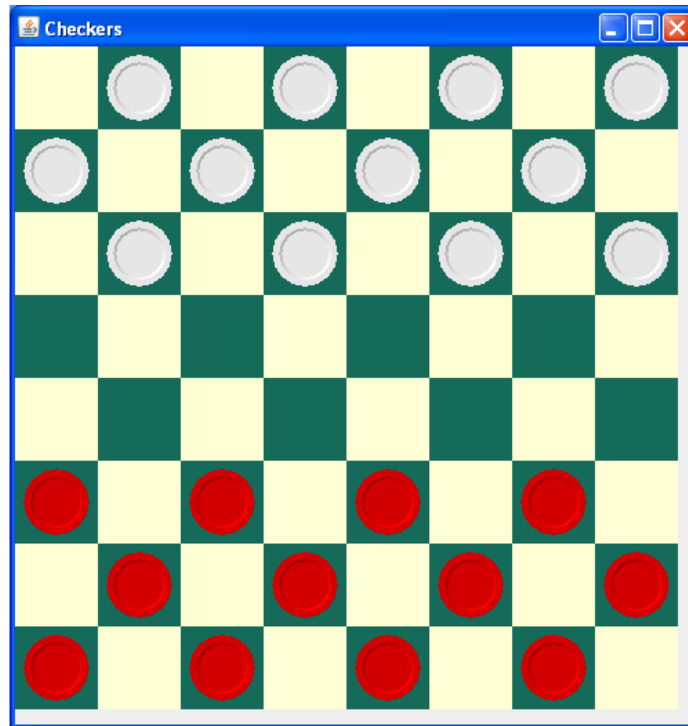


Figure 44: Checkerboard with 3D pieces

(e) Use the `setStroke` method to thicken the stroke and finally draw the ring. See Algorithm .164 and Figure 44.

(f) The Graphics2D object's `addRenderingHints` method lets users turn on anti-aliasing to smooth the pieces. The call can be made in the Board class's paint method. First, create the `RenderingHints` object to be used. See Algorithm .165 and Figure 45.

D.6.8 Phase 4

Phase four is the creation of a GUI-based checkerboard that allows pieces to be dragged to any square. Required knowledge: Mouse events.

المنارة للاستشارات
www.manaraa.com

```
// in CheckerBoard.java
public class CheckerBoard extends JPanel {
   private int squareWidth;
   private int numAcross;
   private GameSquare squares[][];

   public CheckerBoard (int squareWidth, int numAcross) {
      this.squareWidth = squareWidth;
      this.numAcross = numAcross;
      int width = squareWidth*numAcross;
      int height = squareWidth*numAcross;
      RenderingHints hints = new RenderingHints(
                              RenderingHints.KEY_ANTIALIASING,
                              RenderingHints.VALUE_ANTIALIAS_ON);
      JFrame frame = new JFrame("Checkers");
      frame.setSize (width, height);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      buildGameSquares();
      frame.add(this);
      frame.setVisible(true);
   }
```

Algorithm .165: Addition of anti-aliasing

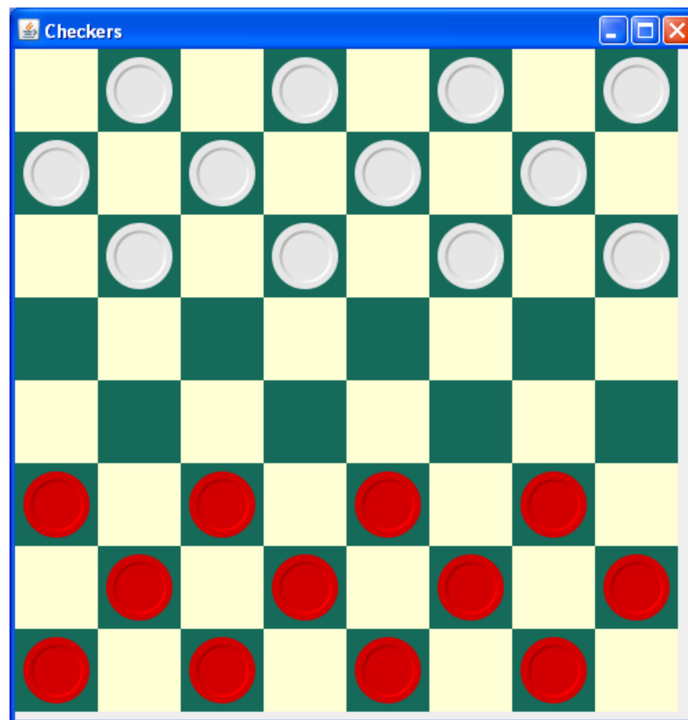


Figure 45: Checkerboard with 3D, anti-aliased pieces

المنارة للاستشارات
www.manaraa.com

```java
// In CheckerBoard.java
   public int numAcross() {
      return numAcross;
   }
```

Algorithm .166: Number across accessor method

1. Identification of new classes needed: a MouseAdapter. The MouseAdapter class already exists in the `java.awt.event`. We need to override its behaviors and will therefore create a child class of MouseAdapter.

2. MouseAdapter Child Class Design

   (a) Behaviors: mouse button press, drag, and release.

   (b) Relationship: child of MouseAdapter, (inner class of CheckerBoard).

3. Piece Design Update

   (a) New Behaviors: make play, draw at a given X,Y location.

4. Board Design Update

   (a) New Attributes: mouse x and y location, currently moving piece, and access to the number of squares across (for bounds checking).

   (b) New Behaviors: creation of mouse listeners, drawing of currently moving piece.

   (c) New Relationships: contains the MouseAdapter child class.

5. Update Board to allow access to the number of square across (for bounds checking), as in Algorithm .166.

6. Create a new Piece draw method that centers the piece at the given (mouse) location. See Algorithm .167.

7. Update Piece's `draw (Graphics2D)` method to invoke the new `draw (Graphics2D, int, int)` in order to reduce redundant code. See Algorithm .168.

8. Create a method in the Piece class for performing moves once a user has indicated (via dragging) where the piece is to go. In order to perform moves, the method must have the new location of the Piece and

المنارة للاستشارات
www.manaraa.com

```
public void draw (Graphics2D g, int x, int y) {
   int left = (int)(x - squareWidth*0.4f);
   int top  = (int)(y - squareWidth*0.4f);

   int innerLeft = (int)(left + squareWidth * 0.1f);
   int innerTop = (int)(top + squareWidth * 0.1f);
   int innerRight = (int)(x + squareWidth*2.0f);
   int innerBottom = (int)(y + squareWidth*2.0f);

   g.setPaint(color);
   g.fillOval (left, top, width, width);
   GradientPaint shade = new GradientPaint(innerLeft, innerTop,
   color.darker(), innerRight, innerBottom, color.brighter());
   g.setPaint(shade);
   g.setStroke(new BasicStroke(2.0f));
   g.drawOval (innerLeft, innerTop, (int)(squareWidth*.6),
                                    (int)(squareWidth*.6));
}
```

Algorithm .167: Piece draw method with center location specified

```
public void draw (Graphics2D g) {
   int x = col*squareWidth + squareWidth/2;
   int y = row*squareWidth + squareWidth/2;
   draw (g, x, y);
}
```

Algorithm .168: Simplified Piece draw method

المنارة للاستشارات
www.manaraa.com

```
public boolean makePlay(int endRow,int endCol,CheckerBoard b){
   if (endRow < 0 || endRow >= b.numAcross() ||
       endCol < 0 || endCol >= b.numAcross())
      return false;
   b.setPiece(endRow, endCol, this);
}
```

Algorithm .169: Piece play method

```
public class CheckerBoard extends JPanel {
   private int squareWidth;
   private int numAcross;
   private GameSquare squares[][];
   private int mouseX, mouseY;
   private Piece mover;
   private RenderingHints hints;
```

Algorithm .170: Addition of reference to moving piece

access to the Board in order to notify its newly-occupied square about the move and make sure the move is not outside the Board's boundaries. See Algorithm .169.

9. Update Board to have a variable for the Piece being dragged and the current mouse location. See Algorithm .170.

10. Update Board's paint method to draw the piece being moved (if any) at the current mouse coordinates. See Algorithm .171.

11. Create the MouseAdapter child class to handle mouse events. The MouseAdapter can be the Board (if it implements `MouseListener`), an anonymous class, or a nested class inside the Board class. See

```
public void paint(Graphics g) {
   super.paint(g);
   ((Graphics2D)g).addRenderingHints(hints);
   for (int i = 0; i < squares.length; ++i) {
      for (int j = 0; j < squares[i].length; ++j) {
         squares[i][j].draw((Graphics2D)g);
      }
   }
   if (mover!=null)mover.draw((Graphics2D)g, mouseX, mouseY);
}
```

Algorithm .171: Updated CheckerBoard's paint method

المنارة للاستشارات
www.manaraa.com

```
// In CheckerBoard.java in the CheckerBoard class
   class PlayListener extends MouseAdapter {
```

Algorithm .172: Extension of the MouseAdapter

```
      public void mousePressed(MouseEvent e) {
         if (mover == null) {
            int col = (int)(e.getX()/squareWidth);
            int row = (int)(e.getY()/squareWidth);
            mover = squares[row][col].getPiece();
            if (mover != null) {
               squares[row][col].setPiece(null);
            } else {
               mover = null;
            }
         }
         mouseX = e.getX();
         mouseY = e.getY();
         repaint();
      }
```

Algorithm .173: Mouse pressed event

Algorithm .172. The three events involved with moving a piece to a new location are

`public void mousePressed (MouseEvent e)`,

`public void mouseDragged (MouseEvent e)`, and

`public void mouseReleased (MouseEvent e)`. When the mouse is pressed, the piece in the square the mouse is over should be removed and become "attached" to the cursor. To attach the piece to the cursor, set Board's "mover" variable to be a reference to the piece being moved and track the current mouse coordinates. The piece will be drawn centered on those coordinates in Board's paint method. After the changes are made, the Board must be repainted to reflect the changes. See Algorithm .173. When the mouse is dragged, a mouse button is being depressed while the mouse moves. If there is a piece that is currently the "mover," each time the mouse is dragged, the piece should be redrawn at the new mouse coordinates. See Algorithm .174.

When the mouse button is released, if there was a piece being moved, it should be set down on the new square. If the new square is outside of the checkerboard, the piece should jump back to its old position. ("mover" is still storing its old position.) See Algorithm .175.

12. Update the Board class to create and add the MouseAdapter as the MouseListener and the MouseMo-

www.manaraa.com

```
      public void mouseDragged(MouseEvent e) {
         if (mover != null) {
            mouseX = e.getX();
            mouseY = e.getY();
            repaint();
         }
      }
```

Algorithm .174: Mouse dragged event

```
      public void mouseReleased (MouseEvent e) {
         if (mover != null) {
            int col = (int)(e.getX()/squareWidth);
            int row = (int)(e.getY()/squareWidth);
            if (mover.makePlay(row, col, CheckerBoard.this)) {
                    mover = null;
            } else {
               // play invalid; undo
               squares[mover.getRow()]
                      [mover.getCol()].setPiece(mover);
               mover = null;
            }
         }
         repaint();
      }
   } // end MouseAdapter child class
} // end CheckerBoard outer class
```

Algorithm .175: Mouse released method

المنارة للاستشارات
www.manaraa.com

```
public CheckerBoard (int squareWidth, int numAcross) {
  this.squareWidth = squareWidth;
  this.numAcross = numAcross;
  int width = squareWidth*numAcross;
  int height = squareWidth*numAcross;
  hints = new RenderingHints(RenderingHints.KEY_ANTIALIASING,
              RenderingHints.VALUE_ANTIALIAS_ON);
  JFrame frame = new JFrame("Checkers");
  frame.setSize (width, height);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  buildGameSquares();
  frame.add(this);
  frame.setVisible (true);

  PlayListener listener = new PlayListener();
  addMouseListener(listener);
  addMouseMotionListener (listener);
}
```

Algorithm .176: Addition of listener

tionListener. Mouse events involving motion (drag and move) must handled by MouseMotionListeners. Other mouse events (press, release, click (a combination of press and release), etc.) must be handled by MouseListeners. The Board JPanel must add action listeners to handle these events. Obviously, the listener will be the MouseAdapter child class that was just written (in our example, "PlayListener"). See Algorithm .176.

D.6.9 Phase 5

Phase five is the creation of a GUI-based checkerboard that allows pieces to be moved to valid move locations (diagonally forward). Turns do not matter yet.

1. Identification of new classes needed: Player. The Player class will keep track of the color and direction of its pieces. Piece no longer needs to track its color but only its Player. Later on, Player will be responsibly for taking turns, identifying when the game has been completed, sending plays over the network, etc.

2. Player Design

   (a) Attributes: color and direction

   (b) Behaviors: initialization and data access

المنارة للاستشارات
www.manaraa.com

```
import java.awt.Color;
public class Player {
   private boolean goingDown;
   private Color color;

   public Player (Color color, boolean goingDown) {
      this.color = color;
      this.goingDown = goingDown;
   }
    public Color getColor() {
      return color;
   }
   public boolean goingDown() {
      return goingDown;
   }
}
```

Algorithm .177: Player class

   (c) Relationships: Piece has a Player

3. Piece Design Update
   (a) New Attributes: Player replaces Color
   (b) New Behaviors: move validation
   (c) New Relationships: Piece has a Player
4. Checkers Design Update
   (a) New Behaviors: Creation of two Players
   (b) New Relationships: Checkers uses two Players
5. Board Design Update
   (a) New Behaviors: Allow access to Pieces on the Board.
6. Create Player class. The attributes needed are a color and a direction. The directions the pieces move on the board are up and down. Since there are only two values, Player's direction can be tracked by a boolean. The only behaviors Player currently needs to perform are initialization and instance variable access. See Algorithm .177.

المنارة للاستشارات
www.manaraa.com

```java
public class Piece {
   private int row, col;
   private Player player;
   private static int width, squareWidth;

   public Piece (Player player) {
      this.player = player;
      row = -1;
      col = -1;
   }
   public Piece (Player player, int row, int col) {
      this.player = player;
      this.row = row;
      this.col = col;
   }
```

Algorithm .178: Addition of a Player to the Piece class

```java
// in Piece.java
   public Color getColor () {
      return player.getColor();
   }
```

Algorithm .179: Updated color accessor method

7. Add a Player instance variable to Piece and remove Color. Additionally, update the constructors to accept a Player instead of a Color. See Algorithm .178.

   Add a `getColor()` method to Piece to simplify getting the color from Player, as in Algorithm .179.

8. Update the Piece class to use Player instead of Color for drawing itself. See Algorithm .180.

9. Update Piece to validate attempted moves. A move (not a jump) is valid if it is to an empty, valid square diagonally one square forward, where the forward is the direction dictated by the Piece's Player. Valid moves have a row change of 1 in the direction of play and a column change of 1 or -1. See Algorithm .181.

10. Update Piece's `makePlay` to call isMove before allowing a move. See Algorithm .182.

11. Update checkers to create two players and pass them appropriately to the pieces. See Algorithm .183.

12. Add a method to the Piece class to accept an attempted play, confirm its validity, and perform the move (or take it back).

المنارة للاستشارات
www.manaraa.com

```
public void draw (Graphics2D g, int x, int y) {
   int left = (int)(x - squareWidth*0.4f);
   int top  = (int)(y - squareWidth*0.4f);

   int innerLeft = (int)(left + squareWidth * 0.1f);
   int innerTop = (int)(top + squareWidth * 0.1f);
   int innerRight = (int)(x + squareWidth*2.0f);
   int innerBottom = (int)(y + squareWidth*2.0f);

   g.setPaint(getColor());
   g.fillOval (left, top, width, width);
   GradientPaint shade = new GradientPaint(innerLeft, innerTop,
                  getColor().darker(), innerRight, innerBottom,
                  getColor().brighter());
   g.setPaint(shade);
   g.setStroke(new BasicStroke(2.0f));
   g.drawOval (innerLeft, innerTop, (int)(squareWidth*.6),
                                    (int)(squareWidth*.6));
}
```

Algorithm .180: Updated Piece draw method

```
// in Piece.java
   private boolean isMove (int endRow, int endCol, CheckerBoard b) {
      if (b.getPiece(endRow, endCol) != null) {
         return false;
      } else {
         int dir = player.goingDown()? 1 : -1;
         return (row+1*dir == endRow && (col+1==endCol || col-1==endCol)
            );
      }
   }
```

Algorithm .181: Move validation

```
// in Piece.java
   public boolean makePlay (int endRow, int endCol, CheckerBoard b) {
      if (isMove (endRow, endCol, b)) {
         b.setPiece(endRow, endCol, this);
         return true;
      } else {
         return false;
      }
   }
```

Algorithm .182: Piece's make play move with validation

المنارة للاستشارات
www.manaraa.com

```
// in Checkers.java
   private void setOutPieces (Color light, Color dark) {
      int numLight = 12, numDark = 12;
      Player top = new Player (light, true);
      Player bottom = new Player (dark, false);
      for (int row=0; row < 3; ++row) {
         for(int col=(row+1)%2, int cnt=0;cnt <4;col+=2,++cnt){
            board.setPiece (i, j, new Piece (top));
         }
      }
      for (int row=5; row < 8; ++row) {
         for(int col=(row+1)%2,int cnt=0;cnt <4;col+=2,++cnt){
            board.setPiece (i, j, new Piece (bottom));
         }
      }
      board.repaint();
   }
```

Algorithm .183: Piece placement with associated players

13. Add a MouseAdapter object to the CheckerBoard as the MouseListener and MouseMotionListener.
14. Write the MouseAdapter's mousePressed method to remove the selected piece from the square, set the CheckerBoard moving piece as the piece from that location, and update the mouse coordinates (and repaint).
15. Write the mouseDragged event to update the mouse coordinates and repaint.
16. Write the mouseReleased method to calls the Piece's play method and clear the moved piece being stored in the CheckerBoard object.

D.6.10 Phase 6

Phase six is the creation of a GUI-based checkers game that allows pieces to be moved or single-jumped legally. Turns do not matter yet. No new knowledge is needed.

1. Identification of new classes needed: none.
2. Steps to allow single jumps:
   (a) In the Piece class, add handling for validating a jump. (i.e. Is the target location 2 diagonal spaces forward, over an opponent's piece?)

المنارة للاستشارات
www.manaraa.com

(b) In the Piece class, add handling for performing the jumps. (i.e. Place the piece in the new location, and remove the jumped piece.)

D.6.11 Phase 7

Phase seven is the creation of a GUI-based checkers game that allows pieces to be moved, singly-jumped, and crowns pieces that reach the last rows. Turns do not matter yet. Required knowledge: A way to draw stars on pieces.

1. Identification of new classes needed: King.
2. Identification of King's attributes: none but inherited attributes.
3. Identification of King's behaviors: initialization and drawing.
4. Identification of King's relationship: King is a Piece.
5. Steps to allow crowning:
   (a) In the King class, use the GeneralPath class to draw a star on the king pieces. An example of drawing a star is in the online Java documentation (http://java.sun.com/j2se/1.3/docs/guide/2d/spec/j2d-awt.fm4.html).
   (b) In the appropriate class, add handling for checking whether a Piece has reached the end of the board, and should be replaced by a King with its identical attributes.

D.6.12 Phase 8

Phase eight is the creation of a GUI-based checkers game that allows pieces to be moved, singly-jumped, crowned, and allows king plays. Turns do not matter yet. No new knowledge is needed.

1. Identification of new classes needed: none.
2. Steps to allow king plays: in the King class, override the play validations from the Piece class to allow plays in both directions.

المنارة للاستشارات
www.manaraa.com

D.6.13 Phase 9

Phase nine is the creation of a GUI-based checkers game that allows pieces and kings to be moved, jumped, crowned, and requires multiple jumps to be completed. Turns do not matter yet. Required knowledge: Mouse moved event.

1. Identification of new classes needed: none.
2. Steps to require multiple jump completion:
   (a) Create methods in Piece and King to determine if a given Piece/King can jump.
   (b) In CheckerBoard class, after a jump, check if the Piece object can jump again. If it can, do not allow the piece to be put down until it can no longer jump.
   (c) In the MouseInputAdapter class, add handling for mouseMoved events for when a piece cannot be put down until the play is complete.

D.6.14 Phase 10

Phase ten is the creation of a GUI-based checkers game that allows pieces and kings to be moved, jumped, crowned, and requires sides to take turns. The turns may be enforced using threads, as in an example in Chapter 10 of the suggested textbook. Required knowledge: Optionally threads.

1. Identification of new classes needed: none.
2. Steps to allow taking turns:
   (a) In the Player class,
      i. Add a boolean indicating the Player's turn, as well as corresponding get/set methods.
      ii. Add a variable to hold a reference to the other player.
      iii. When the Player's turn is complete, set the turn for the other player.
   (b) In the Piece/King class, add a method to determine whether a piece is selectable. (e.g. if it is this Piece's Player's turn, it can be selected).
   (c) In the CheckerBoard class, do not allow a piece to be selected unless the Piece is selectable.
   (d) In the Piece/King class, appropriately end the Player's turn. A Player's turn is done when any of the following occurs: a piece is moved, a piece is crowned, or a jumping piece can no longer jump.

المنارة للاستشارات
www.manaraa.com

D.6.15 Phase 11

Phase eleven is the creation of a GUI-based checkers game that allows pieces and kings to be moved, jumped, and crowned, requires turns, and displays current turns. Required knowledge: Layout managers, labels.

1. Identification of new classes needed: none.
2. Steps to allow displaying turns:
    (a) Add a message bar (likely a JLabel) to display whose turn it is. (Likely in the Checkers class, along with the initialization of the JFrame.)
    (b) Add methods to set the the message bar to whose turn it is.
    (c) In the Player class, add a String name (e.g. the player's color) for display.
    (d) When the Player's turn is begun, set the message bar to display his turn, via an object of the class holding the message bar.

D.6.16 Phase 12

Phase twelve is the creation of a GUI-based checkers game that allows pieces and kings to be moved, jumped, and crowned, requires turns, displays current turns, and requires jumps whenever they are available. No new knowledge is needed.

1. Identification of new classes needed: none.
2. Steps to allow forcing jumps when available:
    (a) In the Player class, add a method to determine whether any of the Player's pieces can jump. (All the pieces are located on the CheckerBoard.)
    (b) At the beginning of each turn, determine whether the Player can jump and store the result.
    (c) In the Piece class, do not allow a move to be completed if the Player can jump.

D.6.17 Phase 13

Phase thirteen is a fully-functional, GUI-based checkers game allowing only valid plays, requiring turns, and displaying the winner when the game is completed. Required knowledge: Game lost algorithm.

المنارة للاستشارات
www.manaraa.com

1. Identification of new classes needed: none.
2. Steps to allow displaying the winner:
   (a) In the Player class, add a method to determine whether any of the Player's pieces can move or jump. If a Player cannot perform a play, he has lost the game.
   (b) If no pieces can move or jump, display that this Player lost and the other won.

D.6.18 Phase 14

Phase fourteen is a fully-functional, GUI-based checkers game with double-buffered graphics. Required knowledge: Double buffering.

1. Identification of new classes needed: none.
2. Steps allow double buffering. In the CheckerBoard class,
   (a) Declare an Image instance variable to hold the offscreen drawing.
   (b) Initialize the new Image object to be the current size of the CheckerBoard.
   (c) In the paint method, perform all painting/drawing on the Image object's Graphics (`image.getGraphics()`).
   (d) In the paint method, draw the image onto the passed-in Graphics object. (e.g. `g.drawImage (image, 0, 0, this);`)

D.6.19 Phase 15

Phase fifteen is a fully-functional, double-buffered, GUI-based checkers game that allows resizing. Required knowledge: Component listeners.

1. Identification of new classes needed: a new ComponentAdapter.
2. Identification of the new ComponentAdapter's attributes: none.
3. Identification of the new ComponentAdapter's behaviors: override the componentResized event method.
4. Identification of the new ComponentAdapter's relationship: an extension of the ComponentAdapter class, inner class of CheckerBoard.

المنارة للاستشارات
www.manaraa.com

5. Steps to allow resizing:

   (a) Make sure the Piece classes and GameSquare class have static methods for setting their widths.

   (b) In componentResized method, get the smaller of the new dimensions. The dimensions can be obtained from the Component's provided `getSize()` method.
   (e.g. `Board.this.getSize()` or simply `getSize()`.)

   (c) Reset the size on the double-buffer-related Image object.

   (d) Set the width for the Piece and GameSquare classes.

D.6.20 Phase 16

Phase sixteen is a fully-functional, double-buffered, re-sizable, networked, GUI-based checkers game. Required knowledge: Exceptions, Sockets, Java IO.

1. Exceptions

   (a) Description

      i. Exceptions change the flow of control when something unexpected (such as an error) has occurred. Java exceptions are adapted from C++.

      ii. In the past (such as in C), error conditions were typically indicated by returned error codes. However, since programmers often forget to handle these problems, exceptions force some sort of handling or end the program. Thus, exceptions encourage programmers to take error conditions seriously.

   (b) Handling Exceptions

      i. An Exception is triggered by a "throw" statement. e.g. `throw new Exception();`

      ii. If a method can throw an exception (that is not a "RuntimeException"), it must be declared in the method head, e.g. `public void method () throws Exception {}`

      iii. If a method handles all possible (non-runtime) exceptions, it does not need the throws statement.

      iv. To handle an exception, you must "catch" it. If a method you call can trigger an exception, and you wish to handle it, you must put it in a "try-catch" block, e.g. Algorithm .184.

المنارة للاستشارات
www.manaraa.com

```java
try {
   methodWithPossibleException();
} catch (PossibleException e) {
   // handle exception. e.g.
   System.err.println (e);
}
```

Algorithm .184: Sample exception handling

v. RuntimeExceptions are exceptions typically thrown by the Java runtime library code. RuntimeExceptions are often considered "unrecoverable." Because of their (typically) unrecoverable status, you are not required to handle RuntimeExceptions.

vi. As is the case with all exceptions, unhandled RuntimeExceptions crash the program. You can catch RuntimeExceptions if you wish.

(c) In order to do networking and multi-threading, you must be able to handle exceptions.

(d) Additionally, in NetPlayer, if the input from the network is not in line with the specifications, a user-defined Exception should be thrown.

2. Network play

(a) Identification of new classes needed: a network player.

(b) Identification of NetPlayer's attributes: a port, a way to send to server, a way to receive from server, knowledge of whether to go first.

(c) Identification of NetPlayer's behaviors: client/server setup, override setTurn, and others.

(d) Identification of NetPlayer's relationship: NetPlayer is a Player.

(e) Steps to allow network play:

i. In the Checkers class,

A. Optionally accept a server name from the command line.

B. If a server name was accepted, this session is the acting client. Set the NetPlayer to move first with the dark-colored pieces. Otherwise, the local Player moves first and has the dark-colored pieces.

C. Set the pieces of the local player to be at the bottom of the screen and the NetPlayer to be at the top.

المنارة للاستشارات
www.manaraa.com

ii. In the Player class,

A. Create a method for notifying the other player of the coordinates of a move or part of a jump.

B. Create a method for accepting move/jump coordinates (from another player).

iii. In the NetPlayer class,

A. Create a constructor for setting up the NetPlayer as a client. This constructor must take the server name as a parameter.

B. Create a constructor for setting up the NetPlayer as a server.

C. Override the setTurn method to handle 1) accepting coordinates over the network, 2) translating them to be the right orientation for this board, and 3) performing them locally. This process is continued until the turn is complete: when the word "end" is received. If "end" is the first thing sent, the NetPlayer loses, and the local Player wins. Likewise, if the local player can't play, he loses, and the NetPlayer should transmit "end".

D. If the network input does not line up with expected input, throw your own Exception type.

E. Override "selectable" and "mustJump" to always return false. We will have only the local player check moves, and the NetPlayer's pieces are never movable locally.

D.6.21 Phase 17

Phase seventeen is a fully-functional, double-buffered, re-sizable, networked, GUI-based checkers game that uses threading to prevent freezing during network communication. Required knowledge: Threads.

(a) Threads

i. Concurrent programming is also known as multi-threaded programming. (p. 547)

ii. A thread is a single sequential flow of control within a program.

iii. Most conventional programming languages are single-threaded: handle only one task at any given moment.

iv. Multi-threaded or concurrent programs have multiple threads running simultaneously.

v. Two ways to create threads: extend Thread or implement Runnable.

المنارة للاستشارات
www.manaraa.com

vi. Life cycle (p. 554):

A. A thread is in the "new" state after creation before being started.

B. After the start() method is invoked, a thread is "alive." While a thread is alive, it maybe be "runnable" or "blocked." A thread is blocked if it is waiting (for a notify), waiting to join another thread, or sleeping.

C. Multi-threaded programs run the risk of "race conditions." To prevent these conditions, methods in Java can be declared "synchronized," in order to prevent more than one thread from executing at the same time.

vii. In order to allow regular functionality (e.g. resizing) while waiting for another player's play, the players should be threads. By being Threads, Player waits do not affect the rest of the system.

viii. Player (and therefore NetPlayer) will become a Thread. That is, Player inherits from Thread.

(b) Steps to allow threads:

i. In Player,

A. Have Player extend thread.

B. Override the synchronized `run` method to wait as long as the game is not over and it's not his turn. When it is the human player's turn, if the he can play, the thread waits again to be notified of a play by the human player.

C. When the turn is set for this player, `notify` this waiting thread to wake up and realize it is his turn.

ii. In Checkers, before setting turns, start both threads.

D.6.22 Phase 18

Phase eighteen is a fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game with all typical moves. Detecting checkmate and stalemate are not required yet. Required knowledge: loading images, chess moves.

1. Write `Piece` to be an abstract base class.

   (a) Have the image for the piece retrieved at initialization.

   (b) Create methods for getting and setting the piece location.

المنارة للاستشارات
www.manaraa.com

(c) Create a method checking if a piece is selectable.

(d) Create a method for checking if a move is valid.

(e) Create a method for performing a move.

(f) Create a method for drawing the piece.

2. Write `Knight` to extend and override Piece as needed to define Knight moves: two vertical steps and a horizontal step, or one vertical step and two horizontal steps terminating on a square not occupied by a piece of the same color.

3. Write `Bishop` to extend and override Piece as needed to define Bishop moves: any number of unobstructed, diagonal steps in one direction terminating on a spot not occupied by a piece of the same color.

4. Write `Queen` to extend and override Piece as needed to define Queen moves: any number of unobstructed steps in any single direction terminating on a spot not occupied by a piece of the same color.

5. Write `King` to extend and override Piece as needed to define King moves: One step in any direction terminating on a spot not occupied by a piece of the same color.

6. Write `Rook` to extend and override Piece as needed to define Rook moves: Any number of unobstructed steps in either the vertical or horizontal direction terminating on a spot not occupied by a piece of the same color.

7. Write `Pawn` to extend and override Piece as needed to define Pawn moves:

   (a) One step forward vertically to an empty spot.

   (b) One diagonal step forward to a spot holding an opponent's piece.

   (c) If the pawn has never been moved, two unobstructed steps vertically forward to an empty spot.

D.6.23 Phase 19

Phase nineteen is a fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game with special moves (en-passant, castling, and pawn promotion). Detecting checkmate and stalemate are not required yet.

المنارة للاستشارات
www.manaraa.com

1. Castling: consists of the king's moving horizontally two steps toward his rook, and the rook's moving to the spot the king passed through. Castling can be done if
   (a) The King has never been moved.
   (b) The involved Rook has never been moved.
   (c) There are no pieces between the King and the Rook.
   (d) The squares the King is on, will pass through, and ends on are not in check.
   (e) The King and the Rook are in the same row.
2. Rook: Castling accommodation.
3. Pawn:
   (a) If a pawn is horizontally next to an opponent's pawn that double moved *on the previous turn*, allow the "en-passant" capture diagonally forward above the double-moved pawn. The double-moved pawn is captured.
   (b) If a pawn reaches the last row of the board, promote the pawn to a queen. (Allowing the choice of promotion to a rook, bishop, knight, or queen is optional.)

D.6.24 Phase 20

Phase twenty is a fully-functional, double-buffered, resize-able, networked, multi-threaded, GUI-based chess game that notifies on checkmate and stalemate. Required knowledge: Checkmate and stalemate rules.

1. Update the `Piece` class:
   (a) Create a method for checking if this piece has any valid moves. NOTE: a move is not valid if it causes one's own king to be in check.
   (b) Create a method for undoing a move.
2. Update the `Player` class:
   (a) Create a reference to its king.
   (b) Create a method for determining if the king is "in check." The king is in check if any of the opponent's pieces has a valid move to the king's location.

للاستشارات

www.manaraa.com

(c) Create a way for determining if this player is "check-mated." A player is check-mated if it is already in check, and none of its pieces have valid moves that will leave king out of check.

(d) Create a way for determining if this player is "stale-mated." A player (and thus the game) is stale-mated if the king is NOT in check but none of this player's pieces have valid moves for taking the king out of check.

(e) Display a notification when a Player is check-mated. A Player is check-mated if he is already in check, and no valid move by any of his Pieces will take him out of check.

(f) Display a notification when a Player is stale-mated (considered a tie). A Player is stale-mated if he is NOT in check, but all valid moves put him in check.

(g) Identification of stalemates resulting from threefold repetition and the fifty move rule is optional.

المنارة للاستشارات
www.manaraa.com

Bibliography

[1] David Arnow and Oleg Barshay. On-line programming examinations using web to teach. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 21–24, New York, NY, USA, 1999. ACM Press.

[2] Owen Astrachan and Susan H. Rodger. Animation, visualization, and interaction in cs 1 assignments. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 317–321, New York, NY, USA, 1998. ACM Press.

[3] Doug Baldwin. Teaching introductory computer science as the science of algorithms. In *SIGCSE '90: Proceedings of the twenty-first SIGCSE technical symposium on Computer science education*, pages 58–62, New York, NY, USA, 1990. ACM Press.

[4] Catherine C. Bareiss. A semester project for cs1. In *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 310–314, New York, NY, USA, 1996. ACM Press.

[5] H.S. Barrows and R. M. Tambly. *Problem-Based Learning: An Approach to Medical Education*. Springer Publishing Company, New York, 1980.

[6] Mordechai Ben-Ari. Constructivism in computer science education. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 257–261, New York, NY, USA, 1998. ACM Press.

[7] C. Bereiter and M. Scardamalia. *Intentional learning as a goal of instruction*. Erlbaum Associates, Hillsdale, NJ, USA, 1989.

[8] Kim B. Bruce. Controversy on how to teach cs 1: a discussion on the sigcse-members mailing list. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[9] Kevin R. Burger. Teaching two-dimensional array concepts in java with image processing examples. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 205–209, New York, NY, USA, 2003. ACM Press.

[10] Lawrence Cavedon, James Harland, and Lin Padgham. Problem based learning with technological support in an ai subject: description and evaluation. In *ACSE '97: Proceedings of the 2nd Australasian conference on Computer science education*, pages 191–200, New York, NY, USA, 1996. ACM Press.

[11] Mel Ò Cinnèide and Richard Tynan. A problem-based approach to teaching design patterns. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 80–82, New York, NY, USA, 2004. ACM Press.

[12] Steve Cunningham. Graphical problem solving and visual communication in the beginning computer graphics course. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 181–185, New York, NY, USA, 2002. ACM Press.

المنارة للاستشارات
www.manaraa.com

[13] Steve Cunningham and Angela B. Shiflet. Computer graphics in undergraduate computational science education. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 372–375, New York, NY, USA, 2003. ACM Press.

[14] Timothy Davis, Robert Geist, Sarah Matzko, and James Westall. Course development under $\tau\acute{\epsilon}\chi\nu\eta$. In *Eurographics '04: Proceedings of Eurographics 2004*, pages 23–27, New York, NY, USA, 2004. ACM Press.

[15] Timothy Davis, Robert Geist, Sarah Matzko, and James Westall. $\tau\acute{\epsilon}\chi\nu\eta$: a first step. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 125–129, New York, NY, USA, 2004. ACM Press.

[16] Timothy Davis, Robert Geist, Sarah Matzko, and James Westall. $\tau\acute{\epsilon}\chi\nu\eta$: trial phase for the new curriculum. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 415–419, New York, NY, USA, 2007. ACM Press.

[17] Timothy A. Davis. Graphics-based learning in first-year computer science. In *Eurographics '06: Proceedings of Eurographics 2006*, New York, NY, USA, 2006. ACM Press.

[18] Timothy A. Davis and Edward W. Davis. Exploiting frame coherence with the temporal depth buffer in a distributed computing environment. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 29–38, New York, NY, USA, 1999. ACM Press.

[19] Rick Decker and Stuart Hirshfield. The top 10 reasons why object-oriented programming can't be taught in cs 1. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pages 51–55, New York, NY, USA, 1994. ACM Press.

[20] Adair Dingle and Carol Zander. Assessing the ripple effect of cs1 language choice. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pages 85–93, , USA, 2000. Consortium for Computing Sciences in Colleges.

[21] Barbara Duch, Susan Gron, and Deborah Allen. *The power of problem-based learning*. Stylus Publishing, LLC, Sterling, VA, 2001.

[22] Andrew T. Duchowski and Timothy A. Davis. Teaching algorithms and data structures through graphics. In *Eurographics '07: Proceedings of Eurographics 2007*, New York, NY, USA, 2007. ACM Press.

[23] Harriet J. Fell and Viera K. Proulx. Exploring martian planetary images: C++ exercises for cs1. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 30–34, New York, NY, USA, 1997. ACM Press.

[24] Ronald A. Fisher. *The Design of Experiments, 8th edition*. Hafner Publishing Company, New York, USA, 1966.

[25] Ahmad Ghafarian. Teaching design effectively in the introductory programming courses. In *CCSC '00: Proceedings of the fourteenth annual consortium on Small Colleges Southeastern conference*, pages 201–208, , USA, 2000. Consortium for Computing Sciences in Colleges.

[26] Ahmad Ghafarian. Incorporating a semester-long project into the cs 2 course. *J. Comput. Small Coll.*, 17(2):183–190, 2001.

[27] Andrew S. Glassner. *An introduction to ray tracing*. Academic Press Ltd., London, UK, 1989.

[28] Tony Greening, Judy Kay, Jeffrey H. Kingston, and Kathryn Crawford. Results of a pbl trial in first-year computer science. In *ACSE '97: Proceedings of the 2nd Australasian conference on Computer science education*, pages 201–206, New York, NY, USA, 1996. ACM Press.

المنارة للاستشارات
www.manaraa.com

[29] Brian Hanks, Charlie McDowell, David Draper, and Milovan Krnjajic. Program quality with pair programming in cs1. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 176–180, New York, NY, USA, 2004. ACM Press.

[30] Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 131–134, New York, NY, USA, 1997. ACM Press.

[31] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 71–78, New York, NY, USA, 1992. ACM Press.

[32] Chenglie Hu. Rethinking of teaching objects-first. *Education and Information Technologies*, 9(3):209–218, 2004.

[33] John Hunt and Sarah Matzko. Retooling a curriculum. *Accepted to J. Comput. Small Coll.*, 2007.

[34] Kenny Hunt. Using image processing to teach cs1 and cs2. *SIGCSE Bull.*, 35(4):86–89, 2003.

[35] Ricardo Jimènez-Peris, Sami Khuri, and Marta Patiño-Martìnez. Adding breadth to cs1 and cs2 courses through visual and interactive programming projects. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 252–256, New York, NY, USA, 1999. ACM Press.

[36] D. Johnson and R. Johnson. *Learning Together and Alone*. Allyn and Bacon, Needham Heights, MA, 5 edition, 1999.

[37] Immanuel Kant. *Critique of Pure Reason*. St. Martin's Press, New York, USA, 1965.

[38] Neha Katira, Laurie Williams, Eric Wiebe, Carol Miller, Suzanne Balik, and Ed Gehringer. On understanding compatibility of student pair programmers. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 7–11, New York, NY, USA, 2004. ACM Press.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, PTR, Upper Saddle River, NJ, USA, 2 edition, 1988.

[40] Andrew Koenig. C traps and pitfalls. Computing Science Technical Report 123, AT&T Bell Laboratories, Murray Hill, NJ, July 1 1986.

[41] Michael Kölling, Bett Koch, and John Rosenberg. Requirements for a first year object-oriented teaching language. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 173–177, New York, NY, USA, 1995. ACM Press.

[42] John L. Kundert-Gibbs. *Maya: Secrets of the Pros*. SYBEX Inc., Alameda, CA, USA, 2002.

[43] Gilbert W. Laware and Andrew J. Walters. Real world problems bringing life to course content. In *CITC5 '04: Proceedings of the 5th conference on Information technology education*, pages 6–12, New York, NY, USA, 2004. ACM Press.

[44] Pete Lee and Chris Phillips. Programming versus design (poster): teaching first year students. In *ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education*, page 289, New York, NY, USA, 1998. ACM Press.

[45] Paul M. Leonardi. The mythos of engineering culture: A study of communicative performances and interaction. Master's thesis, Boulder, CO, USA, 2003.

المنارة للاستشارات
www.manaraa.com

[46] Raymond Lister, Anders Berglund, Tony Clear, Joe Bergin, Kathy Garvin-Doxas, Brian Hanks, Lew Hitchner, Andrew Luxton-Reilly, Kate Sanders, Carsten Schulte, and Jacqueline L. Whalley. Research perspectives on the objects-early debate. In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 146–165, New York, NY, USA, 2006. ACM Press.

[47] Margaret Martinez. Designing intentional learning environments. In *SIGDOC '97: Proceedings of the 15th annual international conference on Computer documentation*, pages 173–180, New York, NY, USA, 1997. ACM Press.

[48] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, and Rosemary Monahan. Reveal: a tool to reverse engineer class diagrams. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 13–21, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[49] Sarah Matzko and Timothy Davis. Pair design in undergraduate labs. *J. Comput. Small Coll.*, 22(2):123–130, 2006.

[50] Sarah Matzko and Timothy Davis. Using graphics research to teach freshman computer science. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Educators program*, page 9, New York, NY, USA, 2006. ACM Press.

[51] Sarah Matzko and Timothy A. Davis. Teaching cs1 with graphics and c. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 168–172, New York, NY, USA, 2006. ACM Press.

[52] Alasdair McAndrew and Anne Venables. A "secondary" look at digital image processing. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 337–341, New York, NY, USA, 2005. ACM Press.

[53] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald. The impact of pair programming on student performance, perception and persistence. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 602–607, Washington, DC, USA, 2003. IEEE Computer Society.

[54] Emilia Mendes, Lubna Basil Al-Fakhri, and Andrew Luxton-Reilly. Investigating pair-programming in a 2nd-year software development and design computer science course. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 296–300, New York, NY, USA, 2005. ACM Press.

[55] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 75–79, New York, NY, USA, 2004. ACM Press.

[56] F. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical Report RR-639, Yale University, Dept. of Comp. Sci., July 1988.

[57] M. A. Pèrez-Quiñones, Steven Edwards, Claude Anderson, Doug Baldwin, James Caristi, and Paul J. Wagner. Transitioning to an objects-early three-course introductory sequence: issues and experiences. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 499–500, New York, NY, USA, 2004. ACM Press.

[58] Jean Piaget. *The development of thought: equilibration of cognitive structures (translated by A. Rosin)*. Viking Press, New York, USA, 1977.

المنارة للاستشارات
www.manaraa.com

[59] Margaret M. Reek. A top-down approach to teaching programming. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 6–9, New York, NY, USA, 1995. ACM Press.

[60] Erik Reinhard, Michael Ashikhmin, Bruce Gooch, and Peter Shirley. Color transfer between images. *IEEE Comput. Graph. Appl.*, 21(5):34–41, 2001.

[61] James Robergè. Creating programming projects with visual impact. In *SIGCSE '92: Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 230–234, New York, NY, USA, 1992. ACM Press.

[62] Eric Roberts. The dream of a common language: the search for simplicity and stability in computer science education. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 115–119, New York, NY, USA, 2004. ACM Press.

[63] Eric S. Roberts. Using c in cs1: evaluating the stanford experience. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 117–121, New York, NY, USA, 1993. ACM Press.

[64] Nathan Rountree, Janet Rountree, Anthony Robins, and Robert Hannah. Interacting factors that predict success and failure in a cs1 course. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 101–104, New York, NY, USA, 2004. ACM Press.

[65] Jean-Jacques Rousseau. *Emile, or On Education*. Paris, France, 1762.

[66] Keith Rule. *3D graphics file formats: a programmer's reference*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[67] G. Michael Schneider. A model for a three course introductory sequence. *SIGCSE Bull.*, 36(2):40–43, 2004.

[68] E. Seymour and N. Hewitt. *Talking about leaving: Why Undergraduates Leave the Sciences*. Westview Press, Boulder, CO, USA, 1997.

[69] Rahman Tashakkori. Encouraging undergraduate research: a digital image processing approach. *J. Comput. Small Coll.*, 20(3):173–180, 2005.

[70] Joseph A. Turner and Joseph L. Zachary. Using course-long programming projects in cs2. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 43–47, New York, NY, USA, 1999. ACM Press.

[71] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard Univ. Press, Cambridge, MA, USA, 1978.

[72] S. Walker and B. Fraser. Development and validation of an instrument for assessing distance education learning environments in higher education. *Learning Environments Research*, 8:289–308, 2005.

[73] Kent White. A comprehensive cmps ii semester project. *SIGCSE Bull.*, 35(2):70–73, 2003.

[74] Richard Wicentowski and Tia Newhall. Using image processing projects to teach cs1 topics. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 287–291, New York, NY, USA, 2005. ACM Press.

[75] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Softw.*, 17(4):19–25, 2000.

المنارة للاستشارات
www.manaraa.com

[76] Laurie Williams and Richard L. Upchurch. In support of student pair-programming. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 327–331, New York, NY, USA, 2001. ACM Press.

[77] Rosalee Wolfe. New possibilities in the introductory graphics course for computer science majors. *SIGGRAPH Comput. Graph.*, 33(2):35–39, 1999.

[78] Craig Zilles. Spimbot: an engaging, problem-based approach to teaching assembly language programming. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 106–110, New York, NY, USA, 2005. ACM Press.

المنارة للاستشارات

www.manaraa.com