2005-07-08

Accelerated Ray Traced Animations Exploiting Temporal Coherence

Darwin Tarry Baines
Brigham Young University - Provo

ACCELERATED RAY TRACED ANIMATIONS EXPLOITING TEMPORAL COHERENCE

by

Darwin T. Baines

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Darwin T. Baines

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Robert P. Burton, Chair
Date	Thomas W. Sederberg
Date	Yiu-Kai Dennis Ng

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Darwin T. Baines in its final form and have found that (1) its format, citation, and bibliographical style are consistent and acceptable and fulfill university and department style require-ments; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for sub-mission to the university library.

Date	Robert P. Burton Chair, Graduate Committee

Accepted for the Department

Date	David W. Embley Graduate Coordinator

Accepted for the College

Date	G. Rex Bryce, Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

ACCELERATED RAY TRACED ANIMATIONS EXPLOITING TEMPORAL COHERENCE

Darwin T. Baines

Department of Computer Science

Master of Science

Ray tracing is a well-know technique for producing realistic graphics. However, the time necessary to generate images is unacceptably long. When producing the many frames that are necessary for animations, the time is magnified. Many methods have been proposed to reduce the calculations necessary in ray tracing. Much of the effort has attempted to reduce the number of rays cast or to reduce the number of intersection calculations. Both of these techniques exploit spatial coherence. These acceleration techniques are expanded not only to exploit spatial coherence but also to exploit temporal coherence in order to reduce calculations by treating animation information as a whole as opposed to isolating calculations to each individual frame. Techniques for exploiting temporal coherence are explored along with associated temporal bounding methods. By first ray tracing a

temporally expanded scene, we are able to avoid traversal calculations in associated frames where object intersection is limited. This reduces the rendering times of the associated frames.

ACKNOWLEDGMENTS

Dr. Robert P. Burton provided me every freedom, opportunity, and encouragement in completing this thesis. The nD research group assisted me in overcoming roadblocks and provided assistance with mathematical background and proofs. Dr. Thomas W. Sederberg and Krzysztof Klimaszewski assisted me in understanding and re-implementing adaptive grids. David Eberly responded helpfully to emails about current methods used in computer graphics. Andrew Glassner responded to inquiries related to temporal bounding. My wife Amber gave me encouragement and support essential to completion of this work.

Table of Content

List of Tables

List of Figures

Chapter 1: Ray Tracing

1.1. Introduction

Ray tracing [Whi80] is a technique for producing images of virtual scenes that contain complex light interaction such as reflection and refraction. Unfortunately the time needed to render these images is substantial. Because the intersection of each ray with each object in the scene must be calculated, calculations times can be very high for complex scenes. This is amplified further by intersection calculations for secondary rays.

Rendering animations adds further to the complexity of performing these calculations. An animation is the assembly of static images that, when viewed sequentially, produces a coherent image that appears to be dynamic.

Several approaches have been taken to reduce rendering times in ray tracing. This work investigates means of exploiting temporal coherence to reduce further the rendering times of individual animation frames.

1.2 Tracing Rays

One of the primary focuses of computer graphics is the production of photo-realistic images. Photorealism usually is achieved by creating a scene and by simulating the interaction of light and objects in the scene. Ideally, any simulation will model the transport intensity of light from each point in the scene to all others points in the scene. This is expressed in the *rendering equation*[Kaj86]. Unfortunately, with current processing limitation, simulating even simple scenes in this manner is impractical. Therefore, rather than attempt to simulate a scene perfectly, computer graphics techniques often concentrate on certain aspects of lighting simulation that are relatively important to creating a realistic looking scene. This is true for ray tracing. Although ray tracing is able to model phenomena such as shadows, refraction, specular reflection, and diffuse and specular illumination[Whi80], it does not include light interactions such as diffuse reflection, diffraction, diffuse transmission (e.g. a lampshade that spreads the light), or indirect specular reflection or refraction (e.g. a mirror reflecting light onto another object, or a magnifying glass focusing light). Traditional ray tracing has been extended to address these light interactions using non-traditional ray tracing techniques [Fuj88][GWS04].

As its name suggests, ray tracing follows the path of light rays as they travel in a virtual scene. Rather than following rays from the light to the eye, a reverse path follows the ray from the eye to the light source. This reduces the number of rays that are calculated by excluding all rays that do not intersect the viewplane. Rays that initially emanate from the eye and pass through the viewplane are called primary rays or eye rays. These rays are tested against the objects in the scene to see if there is any intersection with them and, if so, which intersection is the closest. Lighting of the object then is calculated for the intersection point. Secondary rays also are sent out for the purpose of including such things as shadows, reflection, and refraction in the lighting calculations. This process continues recursively until a depth tolerance is reached, no intersection is found, or a ray intersects an object that is neither reflective nor refractive. An example of this is presented in Figure 1.1.

From this a ray tree can be created where any path can be followed from the eye to its final intersection point, as shown in Figure 1.2.

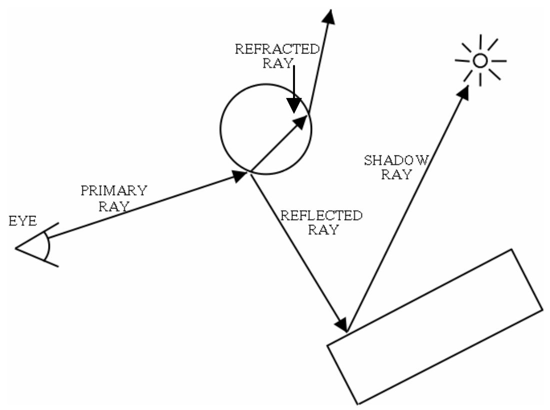


Figure 1.1 Ray path through a scene.

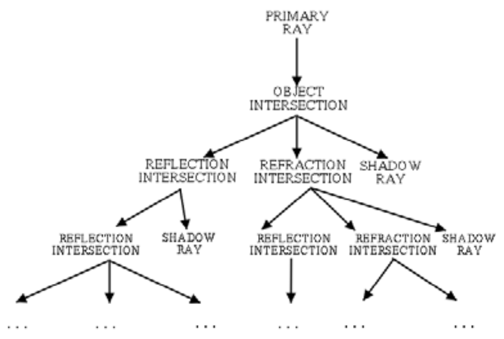


Figure 1.2 Ray tree.

1.3 Illumination Model

As discussed in the previous section, the lighting of a pixel is based on the ray that originates at the eye, passes through that pixel, and strikes the closest object along that ray's path. From that intersection, lighting can be calculated based on diffuse and specular attributes of the object, and also from recursive calls to reflected and refracted rays. The lighting equation can be written as

$$I = I_a + I_d + I_s + I_r + I_t \tag{1.1}$$

where I is the intensity of the pixel, I_d is the intensity of the diffuse component, I_s is the intensity of the specular component, I_r is the contribution of reflection, and I_t is the contribution of transmission or refraction. Because the ray tracing algorithm fails to simulate the illumination model perfectly, there is an ambient component (I_a) which is a simple addition of light to compensate for this deficiency.

An expanded version of the illumination model can be written as

$$I_\lambda = \underbrace{k_a O_{d\lambda}}_{ambient} + \sum_{1\le i\le m} S_i f_{att_i} \left(\underbrace{k_d O_{d\lambda} (\overline{N} \bullet \overline{L}_i)}_{diffuse} + \underbrace{k_s O_{s\lambda} (\overline{R} \bullet \overline{V})^n}_{specular} \right) + \underbrace{k_s I_{r\lambda}}_{reflected} + \underbrace{k_t I_{t\lambda}}_{refracted} \qquad (1.2)$$

where k is the coefficient for the object material that determines that portions contribution to the lighting. O is the reflected color of the objects material. Although we include the ambient light consistently with the other light contributions, it isn't important that it be modeled this way. This is because it isn't modeling any natural light interaction. f_{att_i} is the attenuation factor which is related to the distance of the intersection point from the light source. When the light is farther away from the intersection point, less light reaches the intersection point and eventually the eye. S_i is a binary function which has a value of 0 if the shadow ray (the direct path from the intersection point to the light source) is occluded and 1 if it is not. λ represents separate color channels.

1.4 Distributed Ray Tracing

Because it is unreasonable to assume that a single primary ray can model the light represented in one pixel, approaches such as supersampling and ray distribution have been introduced. Supersampling involves dividing pixels into regions and sending/shooting rays through those sub pixel regions. This can help reduce aliasing, but not eliminate it.

A more effective approach is stochastic sampling[Coo86]. This technique eliminates aliasing by distributing the rays nonuniformly, similar to the method that the human eye uses to avoid aliasing. Outside of the fovea where cones are less prominent (and few samples are taken), the cones are distributed according to a Poisson disk distribution. This means that the cones are distributed similarly to a random distribution, except that there is a high probability that they are no closer than a certain threshold. A similar effect can be achieved by jittering each ray in a subpixel area.

Subpixel sampling need not apply only to primary rays. When reflection ray samples are distributed, gloss (blurred reflection) can be achieved. By distributing refracted rays, translucency is achieved. By distributing shadow rays, penumbras are created. Distributed rays also can be used when simulating a camera lens to produced depth of field. Finally, when rays are distributed in time, motion blur is achieved.

Chapter 2: Acceleration Techniques

2.1 Intersection Acceleration

Because each cast ray potentially tests for intersections with every object in the scene, ray tracing can take an intolerably large amount of time to complete the rendering task. Because ray-object intersection dominates the time required for rendering ray traced images, most of the attempts to accelerate ray tracing has been focused on reducing the time spent determining ray-object intersection. . By reducing the number of ray-object intersection tests, generally available/used techniques discussed later in this chapter reduce the ray-object intersection time

One approach that has been taken to reduce rendering times is the exploitation of spatial coherence. Spatial coherence in a scene occurs because objects and groups of objects are contained within a relatively small space when compared to the space of the entire scene or the space traversed by rays. Because of this, rays that travel through an area need to test for intersections only with objects that are located in that area. Even when objects are not clustered together but are randomly distributed throughout the scene, because the individual objects are contained in a relatively small space compared to the entire scene, a small local path can limit the number of object intersection calculations.

2.2 Bounding Box Hierarchy

Most spatial coherence techniques have attempted to accomplish their objective by partitioning the scene and by associating objects with the partition in which the object resides. One of the first techniques to be associated with ray tracing is the bounding volume hierarchy[RW80]. The process involves adding objects to a bounding volume hierarchy where the resulting surface area minimizes the bounding volume's surface area. Although some propose using bounding volumes that are parallelepipeds oriented to minimize the surface area[RW80], it has become common practice to use axis-aligned parallelepipeds[GS87]. When a ray traverses a scene, it first tests the outermost bounding volume. Should there be an intersection, objects (including other bounding volumes) found inside are then tested for intersections. This is illustrated in figure 2.1. In optimal situations, applying this technique can reduce the number of intersections tested for each ray from n to log n.

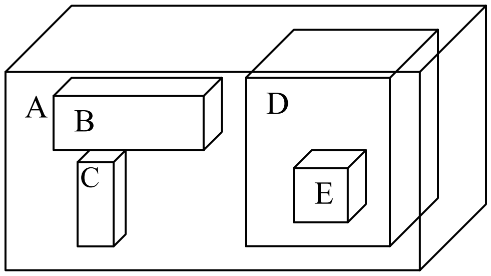


Figure 2.1 A bounding volume hierarchy. In this case, box *A* has children *B*, *C*, and *D*. Box *D* has child *E*.

Goldsmith and Salmon introduce a technique for automatically creating the bounding volume hierarchy[GS87]. Although the hierarchy created is suboptimal, it is generated in $n \log n$ time. Also, should objects be inserted into the hierarchy in a random order, the result is a hierarchy that is near optimal. Any optimal, automatically generated hierarchy takes at least n^2 time to generate. An optimal grid requires comparisons of different hierarchy configuration based on a global search whereas [GS87] performs a local evaluation. When an object is to be added to the hierarchy, there are three possible options for inserting an objects at a location: 1) create a new bounding volume which includes the object and the bounding volume tested against as shown in figure 2.2a, 2) add the object as a child of the bounding volume as shown in figure 2.2b, or 3) recursively test, inserting the object into the children volumes of the bounding volume to determine which surface area is increased the least.

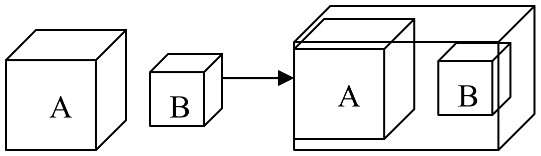


Figure 2.2a A new box is created to include box A and B.

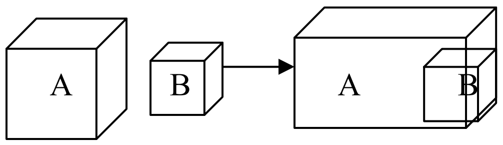


Figure 2.2b Box B becomes a child of Box A. If necessary box A's bounds are extended.

2.3 Grid Traversal

While the hierarchical bounding box is extremely scene-dependent, other technique attempt to partition scenes independent of the scene. One of these techniques is the application of uniform grids. Although the grids can be dependent on the size of the entire scene and number of objects in the scene, the actual division of the scene has no dependence on the placement of the objects in the scene. The technique divides the entire scene into grid areas, where traditionally the number of division in each dimension is equal [FTI86; SB87; Dev89; JW89; CDP95]. In common practice, the number of grids is set to equal the number of objects in the scene. This results in $\sqrt[3]{n}$ divisions in each dimension where n is the number of objects in the scene.

The algorithm follows a ray path and traverses those grids through which the ray passes, as seen in Figure 2.3. As the ray passes through a particular grid, the objects that intersect that grid are tested for intersections. Should there be an intersection in that grid, the closest intersection in that grid is found and is used as the ray intersection point. Much of the speed-up results from the fact that when an intersection is found in a voxel, subsequent grids are not traversed because none of those grids can produce a closer intersection. A quick grid traversal algorithm comes courtesy of the scan-line algorithm that is well known in computer graphics. The technique is extended to a third dimension and is commonly known as the 3DDDA algorithm[FTI86].

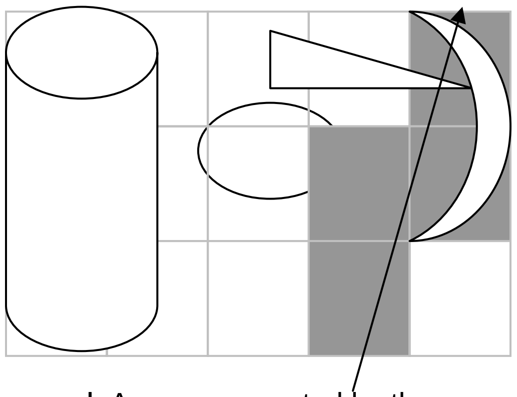


Figure 2.3 Grid Traversal. A ray represented by the arrow in this scene enters two voxels without detecting any intersections. In the third voxel entered, there is an intersection with an object (The crescent-shaped object) found in the voxel, but the intersection lies outside of the voxel. Therefore, the ray enters a fourth voxel and finds the closest intersection that lies in the fourth voxel.

2.4 Hybrid Acceleration Techniques

Some techniques attempt to achieve a compromise between scene dependent structures and scene independent structures. A straightforward technique that attempts to achieve this compromise is the Jevans and Wyvill technique that introduces subvoxel grids[JW89]. This technique begins by creating a grid of regular voxels in the same way as uniform grids are generated. Once grids are generated, each voxel is checked for overpopulation. Should a voxel be overpopulated, it is divided recursively and replaced by a sub-grid.

A similar technique is the use of octrees[Gla84][Kap85]. Rather than dividing the scene into grids with a varying number of partitions, the scene is divided using binary partitions, which divides a region into eight octants. Like the subvoxel grids, it recursively divides the octants until there are fewer objects in an octant than a predefined threshold.

Devillers[Dev89] proposes creating empty regions by creating maximized axis-aligned areas void of any objects or containing very few objects. Initially a standard uniform grid is constructed. Then *macro-regions* are found, consisting of sparsely populated areas. When a ray traversing the grid encounters a macro-region, objects in the region are tested. If no object intersected in the region, the ray continues out of the region and into the grid position of the exit point of the macro-region. From this point on, the ray continues traversing the grid in the normal fashion, thus simplifying calculations in the simple (underpopulated) portions of the scene. Because macro-regions may overlap, problems may arise when object edges do not lie along principal axes. In such cases an overabundance of macro-regions is created to accommodate a majority of areas with low density. An example is shown in Figure 2.4.

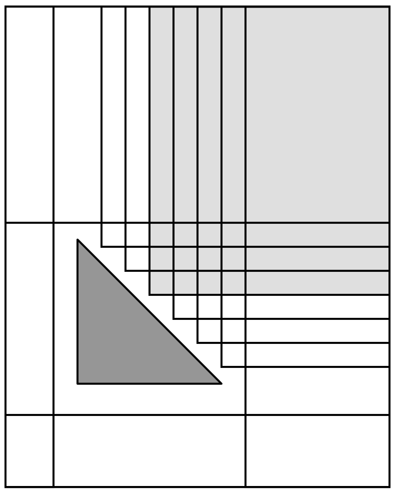


Figure 2.4 Because macro-regions may overlap, when an object or group of objects occupies an area whose boundary is not close to being axis-aligned, an excess of macro-regions may be created. In this illustration, the object's (or group of objects') area is represented by the shaded surface. Macro-regions are represented by rectangular edges. To avoid confusion or ambiguity, one of the macro-regions is shaded lightly.

Finally, Klimaszewski and Sederberg [KS97], apply two of the most common acceleration techniques in grid traversal and bounding volume hierarchies to create adaptive grids. The algorithm involves voxelizing bounding parallelepipeds that were organized using the Goldsmith/Salmon algorithm[GS87]. Prior to inserting objects into the bounding hierarchy, a thorough search organizes close objects into single bounding boxes. Child bounding boxes are tested to remove any child boxes that occupy a large percentage of the parent's area. This facilitates adaptation to the non-uniform organization of a scene while taking advantage of the simplicity of scene subdivision. Because the research and results reported here are based heavily upon this technique, the details of the algorithm are elaborated in chapter 5.

A similar algorithm developed independent of adaptive grids is introduced by Cazals et al. [CDP95]. Like [KS97], they cluster objects in a scene and voxelize the cluster. Large objects are not included in the clustering. Clusters then are inserted into the grid in a recursive manner where smaller clusters are inserted into larger clusters that completely surround them.

2.5 Intersection Reduction

As a result of object coherences, neighboring rays are likely to intersect similar objects. Therefore, neighboring pixels in an image are likely to have identical or similar intensities. This attribute has been described as image coherence, area coherence, or pixel coherence. One of the simplest techniques taking advantage of image coherence uses adaptive undersampling. An obvious disadvantage to undersampling is the loss of information that falls in between samples. This happened in images where skinny objects are lost from the image, or the ends of sharply pointed objects fall between samples and are cut off.

The most basic recursive undersampling implementation is outlined in [AMS91]. In this implementation, a sample is taken at a given interval in both x and y pixel coordinates. Should the four neighboring pixels have similar colors, the intermediate pixels are interpolated. Otherwise, either a finer sample is taken and the algorithm is repeated recursively or all the intermediate pixels are calculated when the sample interval is below a predefined threshold. An example is shown in figure 2.5.

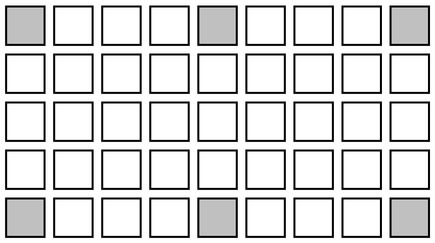


Figure 2.5a An undersampling example. The boxes represent pixels. Pixels colored gray are considered sampled pixels. In this case there are 2 sample areas shown.

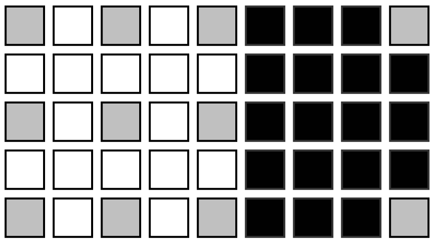


Figure 2.5b One level of recursion. The left area is found incoherent and finer samples are taken. The right area is found coherent and intermediate pixels are interpolated (colored black).

[AMS91] further develops an undersampling technique by introducing error checking to avoid the cutoff of sharp edges. Unfortunately the memory requirements are high relative to any picture quality gains.

Klimazewski [Kli94] proposes a simpler version of this technique which eliminates the recursion from the undersampling algorithm. Area sampling calculates pixel colors at certain samples. When there is lack of coherence, all intermediate pixels are calculated.

Chapter 3: Computer Generated Animation

3.1 Traditional Animation

Many of the techniques used to produce computer animation derive from traditional animation. Many of the steps required to produce such an animation carry over unchanged. Although both use techniques such as storyboarding, soundtracking, and modeling, those techniques are outside the scope of the research reported here. For more information on traditional animation see [HM76].

3.2 Modeling and Positioning

A popular method of creating virtual models is to create surfaces using simple geometric shapes. Limiting surface models to simple triangles can simulate complex non-rigid surfaces effectively -- particularly when shading is included to simulate non-flat surfaces. Because of its simplicity, ease of manipulation, and rendering speed, complex surfaces often are tessellated into triangles or quadrilaterals.

Once a simple model of an object or a portion of an object is available, transformations are use to place the portion of the object in a location in the virtual scene (or with respect to other portions of the object). These transformations are effected by using scaling (***S***), rotation (***R***) and finally translations (***T***) which can position objects arbitrarily in the scene. With the triangles or quadrilaterals being represented by a series of points (***P***), a single transformation can be represented as

$$TRSP = P' \tag{3.1}$$

where P' represents the points after the transform.

This can be expanded as follows:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} r_x^2 + \cos\theta(1 - r_x^2) & r_x r_y(1-\cos\theta) - r_z \sin\theta & r_z r_x(1-\cos\theta) + r_y \sin\theta & 0 \\ r_x r_y(1-\cos\theta) + r_z \sin\theta & r_y^2 + \cos\theta(1 - r_y^2) & r_y r_z(1-\cos\theta) - r_x \sin\theta & 0 \\ r_z r_x(1-\cos\theta) - r_y \sin\theta & r_y r_z(1-\cos\theta) + r_x \sin\theta & r_z^2 + \cos\theta(1 - r_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{bmatrix} \tag{3.2}$$

where $\bar{t}$ represents the translations, $\bar{r}$ represents the axis about which to rotate, θ represents the rotation angle, $\bar{s}$ represents the scaling factor, $\bar{p}$ represents the initial points, and $\bar{p}'$ represents the transformed points.

Furthermore, a hierarchy of transformations of several levels can be used to place simple polygons in the correct scene location. Commonly, one or more levels of transformations place the polygon with respect to the object of which it is part. Then further transformations place the object in relation to the entire scene.

3.3 Keyframing

Once a mechanism for placing objects in the scene is in place, a method of describing movement in the scene over time is required. Similar to traditional animation, a keyframing technique can be used by modelers to define objects at certain intervals and to use other means of defining where the objects are at intermediate times. To get intermediate positions, a calculated path is interpolated between keyframe positions. Even though the term keyframe is used in computer animation, unlike traditional animation the actual positioning at a certain time is arbitrary and does not need to fall in line with the timing of the frames, nor do these defined moments have to be consistent from object to object. They can be defined independent of each object.

A variety of techniques can be used to interpolate intermediate positions, but a common technique is to use cubic splines. Cubic splines offer a simple parametric means of providing acceptable continuity between intermediate sections joined by a keyframe moment while satisfying the variation diminishing property which can plague polynomial interpolation.

Two major variations of cubic splines used for modeling and temporal positioning have certain desirable properties discussed below.

The most popular spline is the Bézier[Bez72] curve. A cubic Bézier curve can be defined by four control point. Two control points define the start and end points of the curve. The other two control points are used to control the exiting and entering tangents at those control points.

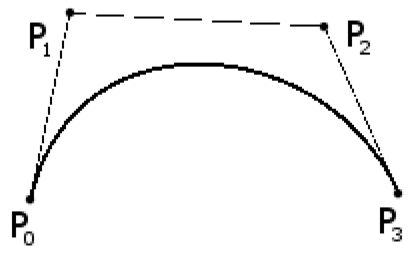


Figure 3.1 A Bézier curve. Points P_0 and P_3 are used in determining the start and end points, respectively. Point P_0 and P_3 are used to calculate the tangents at those endpoints.

The Bézier curve can maintain a degree of continuity by making sure that any following curve segments use the reverse vector created by the end point and its tangent to define the next control point of the following segment. This can be controlled easily in any modeling environment beneath the user interface so the modeler does not have to consider continuity when creating the curve.

The major benefits of using Bézier curves fall more on the computational side than in the ease of use by modelers. The Bézier curve is subdivided easily and satisfies the convex hull property.

The other major path defining spline used in modeling is the Catmull-Rom spline[CR74] which originally is documented in [Over68]. Some modelers wish simply to provide a set of points to be interpolated by the resulting path. Catmull-Rom splines are an effective solution when the modeler is not concerned with such things as tangents or derivatives, but only with moving an object from point "a" to point "b" to point "c".

Catmull-Rom splines pass through all of the control points. The tangent at which the spline passes through a control point is determined by a vector created by the previous and following control points. This requires an initial point and a final point that lie outside the path. If the start point is positioned at the same place as the initial control point, then the starting point tangent points from the starting point to the next control point.

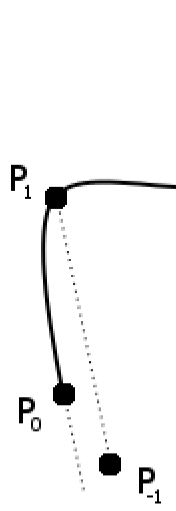


Figure 3.2 A Catmull-Rom Spline. Points P_0 through P_4 are interpolated control points. Points P_{-1} and P_5 are used to establish the initial and ending tangents. Dotted lines are used to try to establish the relationship of those initial and final tangents.

A variation of the Catmull-Rom spline allows for more control over the path of the spline. The Kochanek-Bartels [KB84] splines are known also as TCB-splines since they offer parameters for manipulating the tension, continuity, and bias at any control point. All parameters range between -1 and 1. When all values for tension, continuity, and bias are zero, the spline degenerates into a Catmull-Rom spline.

A tension value greater than zero tightens the rigidness with which the curve follows the control points. With a value less than zero, the curve is loosened. Continuity greater than zero relates the incoming tangent to the following control point and the outgoing tangent to the preceding control point. Negative continuity reverses the relationship relating the incoming with the preceding tangent and the outgoing with the following tangent. The bias parameter maintains the continuity of the outgoing and incoming tangents, but adjusts its weight more on the preceding or following control point—positive continuity weights the preceding point more heavily and a negative number the following point more heavily.

Although these parameters may give the modelers more functionality, they also come with more risk. These parameters—particularly the tension and continuity—can eliminate continuity at the control points.

3.4 Rotation Interpolation

Although translation and scaling are represented easily in Cartesian coordinates, as is visible in equation 3.2 rotation cannot be presented easily. Complexities are introduced when attempting to interpolate between defined rotations. Because rotation is not a linear process, attempting to use a linear interpolation scheme (Bézier curves can be described as a sequence of linear interpolations) to interpolate rotation can result in non-fluid or unnatural movement.

This can be overcome by using quaternions[Ham53] to represent an arbitrary rotation in three dimensions. Quaternions are hypercomplex numbers that have one real part and three imaginary parts. Quaternions represent an extension of the two-dimensional representation of rotation using complex numbers as seen in Figure 3.3.

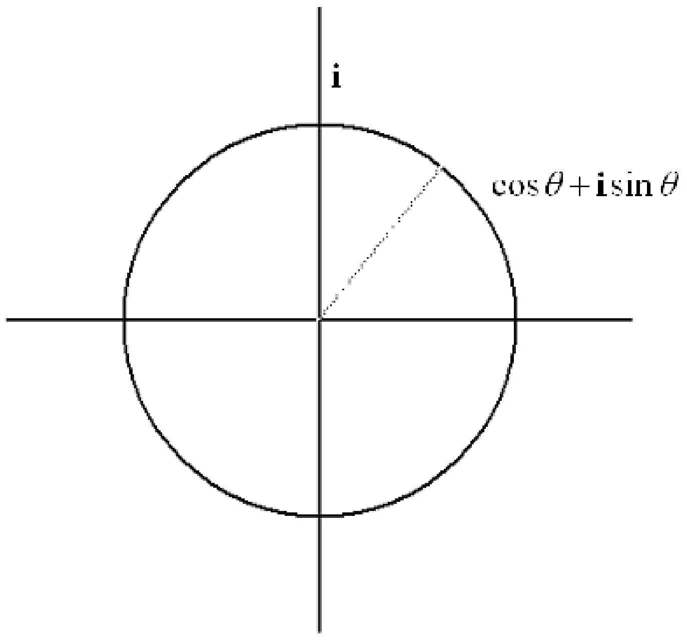


Figure 3.3 2D rotation represented by complex numbers.

The necessity of four parts when extending rotations to three dimensions can been seen by representing a quaternion on a unit sphere. Rather than using a point to represent the placement on the two-dimensional sphere surface, a simple two-dimensional object, such as an arrow or a compass spindle can be used as shown in Figure 3.4. That spindle can take any orientation at a given point. This shows that there is more than a unique rotation for a given point on the unit sphere. To keep the spaces consistent, quaternions often are represented for rotation by points on a unit hypersphere.

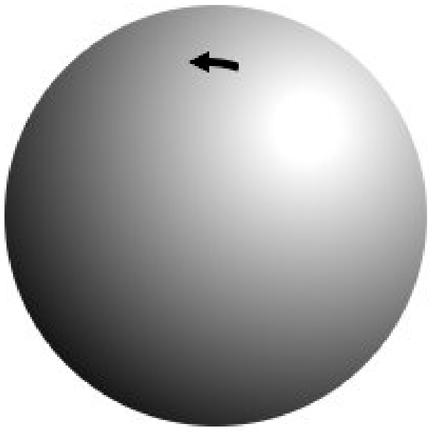


Figure 3.4 3D rotation represented by on a unit sphere.

Rotations can be converted easily from Cartesian coordinates to quaternions. Rotation about vector $[x\ y\ z]^T$ by angle θ corresponds to

$$\cos\frac{\theta}{2} + x\sin\frac{\theta}{2}i + y\sin\frac{\theta}{2}j + z\sin\frac{\theta}{2}k \tag{3.3}$$

where *i*, *j*, and *k* are the imaginary parts. Interpolation in quaternion space can be done using a spherical linear interpolation (slerp)[Sho85]. This creates a straight path from one point on the hypersphere to another. A slerp can be represented as:

$$Slerp(q_0, q_1, t) = q_0\left(q_0^{-1}q_1\right)^t \tag{3.4}$$

Similar to how Bézier curves use linear interpolation in Cartesian space, slerps can be used in quaternion space to effect a similar curve where the continuity is established to produce smooth rotation interpolation.

3.5 Camera Positioning and Motion

Camera positioning can be treated much the same way that object positioning is treated. Translation is used to position the camera point, and rotation is used to orient the look-at and the up vectors. Scaling can be used theoretically to manipulate the field of view, but we are aware of no such implementation.

Chapter 4: Adapting Adaptive Grids and Undersampling to Animation

4.1 Bounding Movement

The most challenging task in this work has been the need to bound all movement. Even though others have addressed temporal bounding, none have addressed it for standard transformations are concerned.

Most of the reported work that addresses temporal bounding includes disclaimers such as “One must be careful to insure that each bound completely encloses the object for the entire time interval,”[Gla88] but none has presented a means of doing so. Rather than address traditional animation positioning, the reported work uses a technique friendly to bounding temporally. Even though temporal bounding of traditional positioning techniques seems useful in dealing with such features such as motion blur, any attempts to find a solution have been left unresolved.

Our first attempt at solving this problem uses Newton's methods where, given the derivatives for the paths, the zero crossing could be located using the iterative method [SB02]. Unfortunately, the time taken to iterate through this process is so large, that even allowing the process to finish is unreasonable.

Our next attempt at bounding movement over time involved a tradeoff between bound tightness and quick bound generation. This process involved using interval arithmetic to find a loose but quick bound on objects as they move over time. To accomplish this, each part of the transformation was bounded and the intervals were combined using interval multiplication. Multiplication then was used to combine all the transformations. The more multiplications that take place, the more the bound can loosen relative to the optimal bound. Bounding each portion of an object's movement involved separate techniques for bounding translation and scale and a different technique for bounding rotation.

Because the control points for translation and scaling are stored in Cartesian coordinate style (x, y, z), a simple bound of the curve that established the movement over time will do. One way of realizing this simple bound involves establishing a convex hull around the curve. For curves that satisfy the convex hull property, only the control points are needed.

For cubic curves that do not satisfy the convex hull property, there is a simple way to convert the curves to cubic Bézier curves. Equation 4.1 illustrates the equality between a cubic Bézier curve and a Kochanek-Bartels spline.

$$TM_bP_b = TM_{kb}P_{kb} \qquad (4.1)$$

In this equation, *T* represents the time parameter vector defined in equation 4.2.

$$\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \qquad (4.2)$$

The Bézier matrix M_b is represented in equation 4.3 as:

$$\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.3}$$

The Kochanek-Bartels spline M_{kb} is defined by equation 4.4 as:

$$\begin{bmatrix} -c_{(0,1)} & 2-c_{(0,0)}+c_{(0,1)}-c_{(1,1)} & -2+c_{(0,0)}-c_{(1,0)}+c_{(1,1)} & c_{(1,0)} \\ 2c_{(0,1)} & -3+2c_{(0,0)}-2c_{(0,1)}+c_{(1,1)} & 3-2c_{(0,0)}+c_{(1,0)}-c_{(1,1)} & -c_{(1,0)} \\ -c_{(0,1)} & -c_{(0,0)}+c_{(0,1)} & c_{(0,0)} & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.4}$$

where the *c* values represent the tension, continuity, and bias parameters associated with the incoming and outgoing tangents at the starting and ending points. The first subscript associates it with either the starting tangent (0) or the ending tangent (1) and the second subscript, the difference (subtraction) between the current control point, and the next (0) or previous (1) control-point. To be more specific, the outgoing tangent T_{0out} at a starting point P_i can be defined as:

$$T_{0out} = c_{(0,0)}\left(P_{i+1} - P_i\right) + c_{(0,1)}\left(P_i - P_{i-1}\right) \tag{4.5}$$

And the incoming tangent T_{1in} at the ending point P_{i+1} can be defined as:

$$T_{1in} = c_{(1,0)}\left(P_{i+2} - P_{i+1}\right) + c_{(1,1)}\left(P_{i+1} - P_i\right) \tag{4.6}$$

From this we can define *c* in terms of tension τ, continuity γ, and bias β as:

$$c_{(0,0)} = \frac{(1-\tau)(1-\gamma)(1-\beta)}{2} \qquad c_{(0,1)} = \frac{(1-\tau)(1+\gamma)(1+\beta)}{2} \tag{4.7a,b}$$

$$c_{(1,0)} = \frac{(1-\tau)(1+\gamma)(1-\beta)}{2} \qquad c_{(1,1)} = \frac{(1-\tau)(1-\gamma)(1+\beta)}{2} \tag{4.7c,d}$$

Having defined the necessary components for equation 4.1, the equation can be rewritten to solve for the Bézier control points as:

$$P_b = M_b^{-1} M_{kb} P_{kb} \tag{4.8}$$

Where the inverse Bézier matrix M_b^{-1} is:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{3} & 1 \\ 0 & \frac{1}{3} & \frac{2}{3} & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.9}$$

Equation 4.8 reduces further to

$$P_b = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{3}c_{(0,1)} & 1+\frac{c_{(0,1)}-c_{(0,0)}}{3} & \frac{1}{3}c_{(0,0)} & 0 \\ 0 & \frac{1}{3}c_{(1,1)} & 1+\frac{c_{(1,0)}-c_{(1,1)}}{3} & -\frac{1}{3}c_{(1,0)} \\ 0 & 0 & 1 & 0 \end{bmatrix} P_{kb} \tag{4.10}$$

This can be solved logically by realizing that the starting points and the ending points are control points (the first and fourth control points for Bézier curves and the second and third control point for Kochanek-Bartels splines). The second and third Bézier control points also are related to the tangents of the starting and end points and can be expressed as:

$$T_{0out} = 3(P_{1b} - P_{0b}) \tag{4.11}$$

$$T_{1in} = -3(P_{2b} - P_{3b}) \tag{4.12}$$

Solving for P_{2b} and P_{3b} and substituting the Bézier tangents with the Kochanek-Bartels tangent definitions results in:

$$P_{1b} = \frac{1}{3}\left[c_{(0,0)}(P_{2kb} - P_{1kb}) + c_{(0,1)}(P_{1kb} - P_{0kb})\right] + P_{1kb} \tag{4.13}$$

$$P_{2b} = -\frac{1}{3}\left[c_{(1,0)}(P_{3kb} - P_{2kb}) + c_{(1,1)}(P_{2kb} - P_{1kb})\right] + P_{2kb} \tag{4.13}$$

which correspond to rows two and three of the matrix in equation 4.10.

Once splines are converted to cubic Bézier curves, control points can be used to establish bounds on the curve. Because Bézier curves are easily subdivided, bounds can be tightened further because the increase in control points more closely fit the curve.

Rotation is more difficult. Because rotation interpolation is not established in Cartesian coordinates, bounding its movement is not trivial. Keyframe interpolation points often are mapped into quaternions space. Once they are mapped in quaternion space, control points are calculated to establish the curve interpolation in that space. Spherical Bézier (sbez) curves are established similar to their Cartesian counterparts. They simply use a series of slerps to establish any point on the curve.

Some fundamental properties of quaternions are used to establish sbez control points. (For more information, see [Eber00]). The derivative of a slerp (equation 3.4) can be written as

$$slerp'(q_0, q_1, t) = q_0\left(q_0^{-1}q_1\right)^t \log\left(q_0^{-1}q_1\right) \tag{4.14}$$

This means that the tangent for spherical lines can be established at *t*=0 as:

$$slerp'(q_0, q_1, 0) = q_0 \log\left(q_0^{-1} q_1\right) \tag{4.15}$$

Besides this, the incoming and outgoing tangents can be calculated for the Kochanek-Bartels spline (our originally inputted keyframes).

$$T_{0out} = c_{(0,1)} \log\left(q_{1kb}^{-1} q_{2kb}\right) + c_{(0,0)} \log\left(q_{0kb}^{-1} q_1\right) \tag{4.15}$$

$$T_{1in} = c_{(1,1)} \log\left(q_{2kb}^{-1} q_{3kb}\right) + c_{(1,0)} \log\left(q_{1kb}^{-1} q_2\right) \tag{4.16}$$

Along with this, Bézier control quaternions can be established knowing the tangents are related to a Bézier in quaternion space.

$$T_{0out} = 3q_{0b} \log\left(q_{0b}^{-1} q_{1b}\right) \tag{4.17}$$

$$T_{1in} = 3q_{3b} \log\left(q_{2b}^{-1} q_{3b}\right) \tag{4.18}$$

The missing Bézier control quaternions can be calculated using the tangent equations from both the Kochanek-Bartels base tangents and the spherical Bézier tangents.

$$q_{1b} = q_{0b} \exp\left(q_{0b}^{-1} \frac{T_{0out}}{3}\right) \tag{4.19}$$

$$q_{2b} = \left(\exp\left(q_{3b}^{-1} \frac{T_{1in}}{3}\right) q_{3b}^{-1}\right)^{-1} \tag{4.20}$$

Using the Bézier control quaternions will bound any movement to the inside of the convex hull on the unit hypersphere. Unfortunately this bound does not create a bound for the movement in Cartesian coordinates. Since we are not dealing with a flat surface in the case of a hypersphere, an internal point may elevate above the edge of the convex hull and hence has the potential of producing internal extrema as demonstrated in figure 4.1. So although these Bézier control points are useful in determining the bounds, further work remains to be done and is described below.

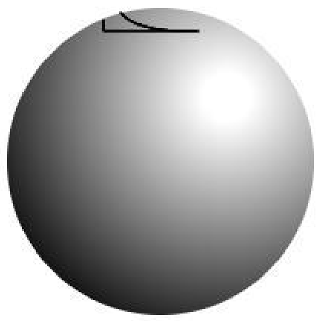


Figure 4.1 An example of how the convex hull of a Bézier on a sphere may not represent all extrema.

A useful approach is to find extrema along the spherical line created between control quaternions as shown if figure 4.2.

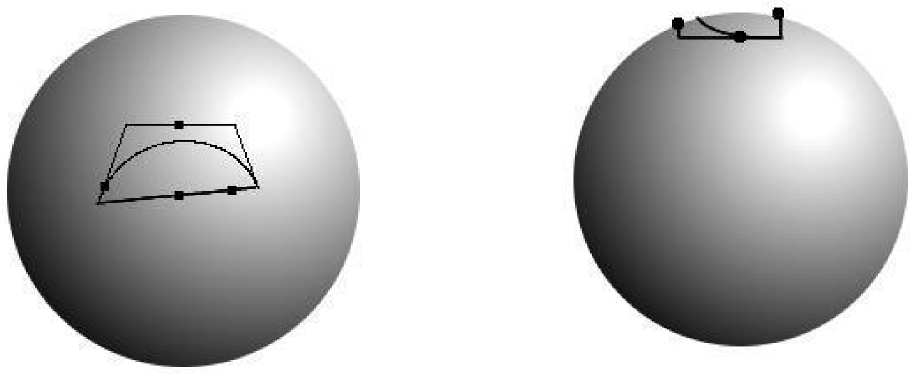


Figure 4.2 Square dots represent the extrema found along the convex hull of a spherical Bézier curve. The two views represent the same object viewed from different angles.

This can be calculated using the derivative of a slerp. A slerp may be represented in terms of the angle θ between the two quaternions.

$$slerp(q_0, q_1, t) = \frac{q_0 \sin((1-t)\theta) + q_1 \sin t\theta}{\sin\theta} \tag{4.21}$$

Its derivative is given by

$$slerp'(q_0, q_1, t) = \frac{\theta(q_1 \cos t\theta - q_0 \cos((1-t)\theta))}{\sin\theta} \tag{4.22}$$

Equating the derivative to zero and solving for *t* gives us the extrema along the slerps. We use the property

$$\cos(a-b) = \cos a \cos b + \sin a \sin b\,. \tag{4.23}$$

Using equation (4.23) we find that

$$q_1 \cos t\theta - q_0 \cos\theta \cos t\theta - q_0 \sin\theta \sin t\theta = 0 \tag{4.24}$$

Then variable *t* is isolated on one side of the equation and equation 4.25 is obtained.

$$t = \frac{\tan^{-1}\left(q_0^{-1} q_1 \csc\theta - \cot\theta\right)}{\theta} \tag{4.25}$$

Once we have any extrema along the convex hull, we can use slerps recursively to connect extrema found along the original slerps. This provides us with any internal extrema as shown in figure 4.3.

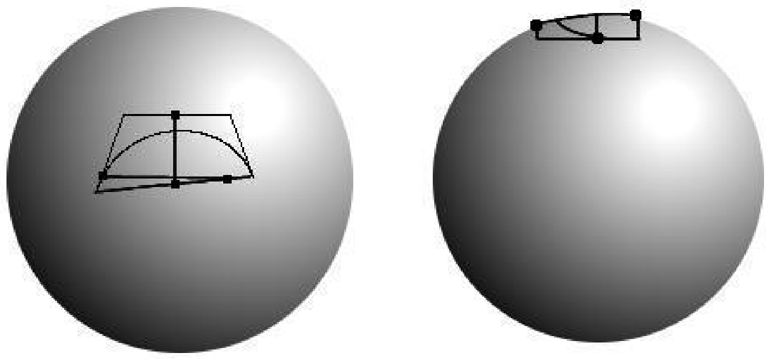


Figure 4.3 Square dots represent the extrema found along the convex hull of a spherical Bézier curve. The two lines through the convex hull represent slerps traveling along an extremum.

Although not representable on a three-dimensional sphere, another level of recursion may be necessary to find all internal extrema.

Once all potential extrema are located, control points and extrema are used to place bounds on x, y, z and θ. These bounds then are used to establish bounds for the rotation matrix found in equation 3.2. Note that $\cos\theta$ and $\sin\theta$ are bound by the end points and at any intermediate crossing by 1 at cos 0, by -1 at $\cos\pi$, by 1 at $\sin\frac{\pi}{2}$, and by -1 at $\sin\frac{3\pi}{2}$.

Now that the bounds for the individual translation, rotation, and scale matrices are available, their bounds can be multiplied together using interval algebra. Once all the transformation matrices are multiplied together, they are multiplied by the bound of the initial object giving us a loose bound on the object as it moves over time.

4.2 Exploiting Object Temporal Bounds

To exploit temporal coherence, the temporal bounds are treated initially as simple objects in the scene and therefore can be placed in a hierarchy for ray traversal.

Once the hierarchy is established, rays are cast through the scene to establish any possible intersection with the temporal bounds. A set of possible ray-object intersections is kept for each ray. Should the number of possible ray-object intersections exceed a predetermined threshold, the set is discarded and no further temporal bound tests are performed. Once the sets have been established for each ray, standard ray tracing is performed for each frame. For those rays that had sets below the cardinality threshold, no hierarchy traversal is used on the primary ray. In this case each object in the set is tested for the closest intersection. When a ray's set cardinality exceeds the threshold, a regular hierarchy traversal is used.

Because the initial temporal bounds has limits on the number of intersection tests that it will perform, and should be a quick and easy means of finding initial intersections, a simple voxel grid is used for placing objects in the scene, reducing the amount of memory used and time necessary to generate the traversal structures. In order to compute reflection, refraction, and shadow ray intersections, normal hierarchy traversal is used.

To accommodate camera movement, and therefore ray movement, the camera is considered static. Any camera transformation is performed inversely on the objects in the scene. This simplifies the bound and intersections calculations. Of course, any camera movement has a large negative impact on the frame-to-frame and inter-frame pixel coherence of an animation.

Camera movement in virtual scenes may be more likely than in traditional filmed scenes due to the absence of physical limitations associated with moving a physical camera. However, diverging too much from tradition may counter the virtual animation's attempt to mimic reality. A leading producer of virtual animated films has stated that virtual animators are careful to limit camera movement [BP03].

4.3 Extending Undersampling Temporally

Extending undersampling temporally is a simple task of extending the initial frame sample interval on top of the initial inter-pixel sample interval. This treats the frame dimension as three-dimensional so that pixels have neighbors in the x, y, and frame directions. This means that pixels that fall in the x, y, and frame intervals are sampled to look for any coherent regions and to eliminate the calculations necessary for rendering those coherent regions.

In this case, area sampling proposed in [Kli94] offers a basis where colors are calculated at x and y intervals and further extended to sample inter-frame at the same interval as shown in Figure 4.4

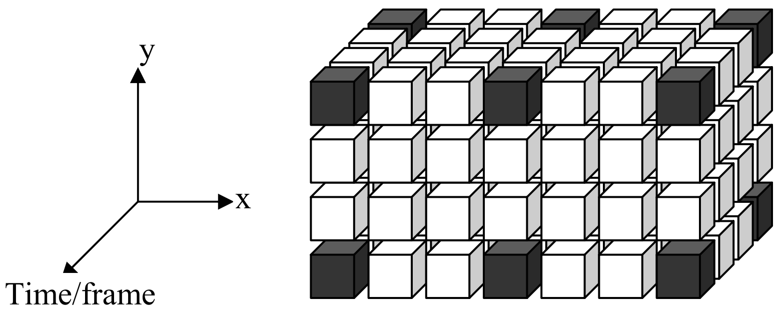


Figure 4.4 Dark cubes represent sampled pixels. Light cubes represent intermediate pixels.

Because region interpolation is now dependent on 8 sampled pixels instead of 4 being coherent, and if we assume a random pixel color, there is half the chance that an area will be determined to be coherent. Fortunately, pixel color is not random but is related closely to neighboring pixels in not only x and y, but also between frames.

When extending the undersampling technique, the ratio of sampled pixels to total pixels is reduced from $4:(i+1)^2$ to $8:(i+1)^3$ where i is the interval of samples. Any interval greater than one will have a reduced ratio that increases as i increases. Thus when there is substantial three-dimensional pixel coherence, temporal undersampling will interpolate more pixels than traditional two-dimensional undersampling.

4.4 Combining Adaptive Grids with Undersampling

Although the temporal undersampling technique described in section 4.3 can be combined with the technique described in section 4.2 for temporal adaptive grids, there is a more complimentary technique that combines the concepts of undersampling and temporal adaptive grids.

The temporal adaptive grid technique attempts to perform an initial scan for object coherence which can simplify the final calculations in raytracing. One approach is to take an initial sample where there is some interval between the x and y pixels. Having calculated an initial sample, in regions of coherence, the sampled object is used to determine the initial intersection. In regions where there is no such coherence, standard raytracing techniques are used. This reduces the number of initial temporal bound intersection calculations which in turn reduces the initial calculations. It also reduces the number of object intersections that must be stored, reducing the memory usage.

Chapter 5: Results

5.1 Specifications

All tests were performed on a Dell Dimension 8400 with an Intel Pentium® 4 Processor 640 with hyper-threading technology with a clock rate of 3.2 GHz. The tests did not attempt to take advantage of the threading technology, utilizing only one thread. The system included one gigabyte of RAM. The operating system used was a Red Hat Fedora Core 3 Linux distribution with the 2.6.9 kernel. The code was written in C++ but with limited use of C++ functionality. The code was compiled with the 3.4.3 gnu compiler (gcc).

Table 5.1 lists test scenes and reports several of their properties.

Scene	***Number of objects***	***Number of frames***	***Resolution***	***Camera movement***
Cubes	Spheres: 512 Cones: 768	24	256 by 256	Translation for initial positioning
Common	Polygons:11622	500	512 by 512	Translation, rotation for initial positioning
Museum	Polygons:10143 Cones:8 Animated Triangles: 64	300	800 by 600	Translation and scaling
Kitchen	Triangles:110561	800	800 by 600	Translation and rotation

Table 5.1 Test scenes and their properties.

The Cubes scene is an extension of the Akimoto Cube [AMS91] where one cube was insufficient for this testing, so several cubes were grouped together and animated. In this scene the camera is static, but the objects revolve around the center of their mass. The Common scene has 3 familiar static objects—the Utah teapot, a Beethoven bust, and a cow. While the objects are static, the camera is translated across the scene and the animation takes place. The Museum and Kitchen scenes are publicly available scenes from [LAM03].

We wrote the ray tracer from scatch, but used the AFF file format whose parser is available at [LAM03]. We produced a common library of ray tracing techniques and used this library to implement the different variations of speedup techniques explored below. Bounding volumes and voxel grids were first explored, follow by subvoxel grids. Building on top of these, adaptive grids were implemented and extended to take advantage of temporal coherence. Finally, undersampling techniques were explored.

5.2 Adaptive Grid Exploration

After completing the basic raytracer, an attempt was made to replicate the work reported in adaptive grids[KS97]. After some time and without success in replicating the results, the original author was contacted. We successfully located an implementation which captures his technique.

The implementation originally did not replicate the results because [GS87] performs a greedy local search for an adequate place to insert objects into the hierarchy. Although [KS97] attempted to emphasize the "minimum surface area" insertion criteria, the search technique for the minimum surface area was not distinguished. Because adaptive grids did not specify the search technique, it was assumed that it used the logarithmic localized search found in [GS87]. The other possibility would be to use the optimal n^2 exhaustive search. Upon review of adaptive grids source code and comparison of results, the exhaustive search establishes a hierarchy that is traversed much more efficiently than by using the localized logarithmic search, significantly reducing the computation time required for a frame. With the merging process reducing the number of objects prior to the hierarchy generation, the hierarchy generation computation differences are reduced. Upon acquiring this information, the implementation results were consistent with adaptive grids.

The merging process is an area of concern for adaptive grids. One of the assumptions that drives adaptive grids is that objects are dispersed statistically throughout the scene in such a way that objects merge quickly, reducing the number of objects that must be searched as the merging process proceeds. Because the merging process is (worst case) order n^3, the potential for a merging bottleneck increases. This may be the case particularly as scene complexity grows as the number of objects is increased and the optimality of the object dispersion is reduced. To speed up this process, a scene is sorted initially by voxelizing the scene and then by merging objects local to each voxel. After some iterations of this process, the merging process is performed on the entire scene. In the spirit of [CDP95], the entire scene is checked initially for large objects. Should their surface area be large enough with respect to the entire scene, the objects are inserted into the root bounding box.

In response to concerns raised in [KWC97], we implemented both adaptive grids and subvoxel grids [JW89]. A sub-voxel creation threshold of 12 was used for the subvoxel grids. For all adaptive grids comparisons, no subvoxels were created. A merging threshold of 2 was used, and a parent-child ratio of 0.1 was used for merging with its parent. That same ratio was used to place large objects in the root node.

Results are presented in table 5.2

Scene	***Subvoxel grid***	***Adaptive grids***
Cubes		
Hierarchy building time average	0	0
Hierarchy building time standard deviation	0	0
Ray tracing time average	9.805833 seconds	10.71333333 seconds
Ray tracing time standard deviation	0.111508	0.164413759
Maximum memory usage	4984 KB	4416 KB
Common		
Hierarchy building time average	0.006333 seconds	0.16422 seconds
Hierarchy building time standard deviation	0.005028	0.005181
Ray tracing time average	16.03384 seconds	22.62176
Ray tracing time standard deviation	2.402787	4.002093
Maximum memory usage	11364	11076

Table 5.2 Computation time and memory comparison of Subvoxel grids and Adaptive grids.

In these cases, both the subvoxel technique and the adaptive grids use an insignificant amount of time generating their scene hierarchy. In these cases, the subvoxel grids were faster, but that adaptive grids used less memory. These results are consistent with results found in [HS99]

The subvoxels hierarchy generation technique has similar but opposite downfalls to adaptive grids; should the scene have area of high object cluster, the time and memory required to build the scene hierarchy is increased greatly. Adaptive grids builds better grids when the objects are clustered closer together because objects are merged into the same hierarchy. The merging time and grid traversal time are increased when object are not close enough to merge. Subvoxel grids must subdivide voxels until all voxels have fewer objects in them than the predefined threshold. This can produce a large number of voxels, thus increasing the memory usage and the grid traversal time. This information could be used beforehand in determining which technique should be used.

5.3 Temporally Adaptive Grids

In order for temporal adaptive grids to be successful, the statistical layout of the temporal scene must be similar to the layout of a standard scene. The surface area of the object's bounding box is particularly important because that determines the likelihood of the bounding box being hit. Any large increase in the size of the temporal bounding box compared to the bound box of a static object will increase the false positives where the temporal bounding box is intersected by a ray but the object is not intersected.

Table 5.3 compares the ratio of temporal bounding boxes of the object as it moves through several frames to the bounding boxes corresponding to the object in a single frame.

Scene	*Ratio of surface areas*	*Standard Deviation*
Cubes	0.648681	0.158358
Common	0.071445	0.128451
Museum	0.008721	0.029677
Kitchen	0.003976	0.027166

Table 5.3 Comparison of temporal bounding box surface area to their equivalent bounding box for the object in a single frame.

The Cubes scene is the only scene where the camera is stationary. The Common scene has a high ratio value (low surface area difference) compared to the Museum and Kitchen scenes, because there is no camera rotation and the objects in the scene are static. Unfortunately it is still a low ratio value when compared with a static camera (as in the Cubes scene). The camera movement even without rotation produces large temporal bounds significant enough to remove any suggestion of tightness of bounds around the object and its movement. The Museum scene and the Kitchen scene were built deliberately to limit the coherence it the scene, so their bounding ratios were expected to be lower than an average scene. The worse the bounding ratios, the less opportunity there is for speedup.

Table 5.4 compares rendering times and memory usage of temporally adaptive grids. Temporally adaptive grids were tested with a frame interval of 4 and a ray caching threshold of 30.

Scene	Temporal Bounding Time	Temporal Voxelization Time	Temporal Initial intersection test	Positioning Time	Hiearchy generation time	Ray tracing Time	Total Time	Cached Rays	Traced Rays
Cubes	a:0.0583 s:0.0075	0	a:0.4883 s:0.0430	a:0.0104 s:0.00204	0	a:10.292 s: 0.2063	10.439	a:65536 s:0	0
Common	a:0.0586 s:0.0061	a:23.305 s:21.427	a:3.0366 s:1.6956	a: 0.0948 s: 0.0948	a:0.3200 s:0.0103	a:21.856 s:3.4423	28.871	a: 57218 s: 65589	a:155474 s:122050
Museum	a:0.0505 s: 0.0110	a:0.2402 s:0.3869	a:3.0366 s:1.6956	a:0.0383 s: 0.0077	a:0.1728 s:0.0350	a: 87.907 s: 0.8308	88.950	a:137726 s:162090	a:342273 s:162090
Kitchen	a:0.94209 s:0.13694	a: 356.97 s: 651.55	a:19.323 s:35.942	a: 0.6801 s: 0.08083	a:12.573 s: 6.1444	a:865.39 s:633.86	972.95	a:42470.75 s:95774.14	a:437529 s:95774

Table 5.4 Time needed to render scenes using temporally adaptive grids. *a* represents the mean while *s* is the standard deviation. The Total Time is calculated by dividing the Temporal relating objects and dividing them by the number of frames that they apply to (in this case 4) and adding them to the other times to calculate a frame.

Scene	*Temporal Adaptive grids*	*Adaptive grids*
Cubes		
Ray tracing time average	10.292 seconds	10.713 seconds
Total time	10.439 seconds	10.713 seconds
Common		
Ray tracing time average	21.856 seconds	22.621 seconds
Total time	28.871 seconds	22.791 seconds
Museum		
Ray tracing time average	87.907 seconds	85.242 seconds
Total time	88.950 seconds	85.37 seconds
Kitchen		
Ray tracing time average	865.39 seconds	656.79 seconds
Total time	972.95 seconds	660.52 seconds

Table 5.5 Comparison of temporal adaptive grids and adaptive grids rendering times.

Even with the poor bounds, the rendering times are similar to the adaptive grids times, with minor improvements in the general ray tracing time on some scenes as shown in table 5.5. Because a larger bound intersects more voxels, more time is needed to insert the objects in the voxels. Should these initial temporal bounding boxes be improved, there is potential for more time improvement in 1) hierarchy generation time because the initial boxes intersect few voxels, 2) the initial temporal tracing time because there are fewer intersection tests in each voxel traversed and 3) The final rays traced because there should be fewer ray traversals when fewer false positive temporal box intersections push the count above the threshold for grid traversal.

The poor temporal bounds also create memory issues. The more voxels that a bounding box intersects, the more memory is required to store pointers to the object represented by the bounding box. Figure 5.6 show memory usage in temporally adaptive grids and adaptive grids. While adaptive grids memory was relatively constant, temporally adaptive grids usage varied.

Maximum memory usage	***Temporally adaptive Grids***	***Adaptive grids***
Cubes	6332KB	4416KB
Common	131MB	11 MB
Museum	92MB	18 MB
Kitchen	2683MB	103MB

Table 5.6 Memory comparison of adaptive grids and temporally adaptive grids.

It should be noted that the Kitchen scene's memory use exceeded the physical memory available (1024 MB) and also came close to filling the virtual memory available. Although the CPU time was used in all our timing, this appears to have an effect on the timing result of the Kitchen scene in Table 5.4. This explains why other scene times were similar to adaptive grid but the Kitchen scene is noticeably higher.

5.4 Undersampling

The second area explored is the use of undersampling in ray tracing. First a comparison of intra-frame to inter-frame sampling is explored. Finally an extension of temporally adaptive grids is explored. For these experiments, a sample interval of 4 is used. The error is calculated similarly to [Klim94] and [AMS91]. The error threshold was set to 3 which is equivalent to 1 for each color channel.

Table 5.7 compares two-dimensional sampling with the three-dimensional sampling.

Scene	*Two-dimensional rendering time (average)*	*Two-dimensional interpolated pixels (average)*	*Two-dimensional memory usage*	*Three-dimensional rendering time (average)*	*Three-dimensional interpolated pixels (average)*	*Three-dimensional memory usage*
Cubes	9.1320	35644	5252KB	8.9800	30293	7012KB
Common	13.680	196245	11284KB	17.720	145404	48968KB
Museum	33.051	338247	18880KB	72.310	376358	31156KB

Table 5.7 A comparison of two-dimensional sampling and three-dimensional sampling where the average rendering time per frame in seconds and average number of interpolated pixels per frame are shown along with the maximum memory usage.

Sampling in three dimensions sometimes produces more interpolated pixels than in two-dimensional sampling. However, the reverse is also true. Although the museum scene had more pixels interpolated on average, the closing frames had a large drop in the number of pixels interpolated. This also produced poor rendering times where some of the final frames where close to three times as slow as previous frames. So although the average number of interpolated pixels is higher than the two-dimensional sample, its times are close to standard adaptive grids because of the slowdown of these last few frames.
There is a correlation between the camera movement and the speedups attained.

Interpolation Error	***Two-dimensional sampling***	***Three-dimensional sampling***
Cubes	0.043938	0.0068721
Common	0.076328	0.065085
Museum	0.21104	0.42786

Table 5.8 A comparison of two-dimensional sampling error and three-dimensional sampling error where it is the average difference in a color channel value that can vary between 0 and 255.

The three-dimensional sampling has as good as if not better error rates than its two

-dimensional equivalent as seen in table 5.8. Both have acceptable error rates as neither differs on average more that one pixel channel intensity which generally is indiscernible to the human eye.

Finally table 5.9 shows the results from rendering done with temporally adaptive grids with temporal undersampling.

Scene	***Rendering Time***	***Number of Rays Interpolated***	***4D intial intersect time***
Cubes	9.740	33006.22	0.02
Common	32.45	0	0.13
Museum	105.04	16255.4	0.95

Table 5. 9 Results of temporally adaptive grids with temporal undersampling.

The temporally adaptive grids with temporal undersampling have a quicker initial intersection test than temporally adaptive grids because fewer tests are performed. However, fewer temporally coherent regions are found. The Common scene found no coherent regions. The Museum scene also has large sections of frames where no coherent regions are found. This means no speedups of the kinds that are associated with standard temporally adaptive grids. Due to the large lack of coherence in the temporal samples, fewer rays where cached to skip traversal. This meant that little or no speedup was achieved and further time costs were incurred creating the hierarchies and initial traversal.

5.5 Conclusion

We have designed and implemented a bounding technique that quickly produces bounds around object hierarchies as they are transformed over time. While the bounds are not optimal, our work represents an important step toward producing near optimal temporal bounds.

Even with the poor bounds, we were able to produce speedups on the order of 4% by taking advantage of temporal coherence in cases when camera movement is limited; improvements are greatest when there is minimal camera movement in the animation.
Should the efficiency of the temporal bounds be improved, we anticipate that these speedups will improve further.

Undersampling can also be effective in reducing rendering times when sampling inter-frame without increasing the interpolated pixel value error. This is especially true when object and camera movement is minimized.

Generally, camera movement greatly reduces the temporal coherence in a scene. It was our original intent to produce techniques that allow for camera movement so as to not limit the application of the techniques. We have concluded that scenes with camera movement lack the temporal coherence sufficient to warrant their inclusion since little improvement is possible using these techniques. However, absent of camera movement, techniques that exploit temporal coherence can reduce the rendering time of animations. Knowing beforehand the extent of movement of objects and cameras facilitates *a priori* determination of the relative appropriateness of temporal acceleration techniques vs. standard acceleration techniques.

5.6 Future Work

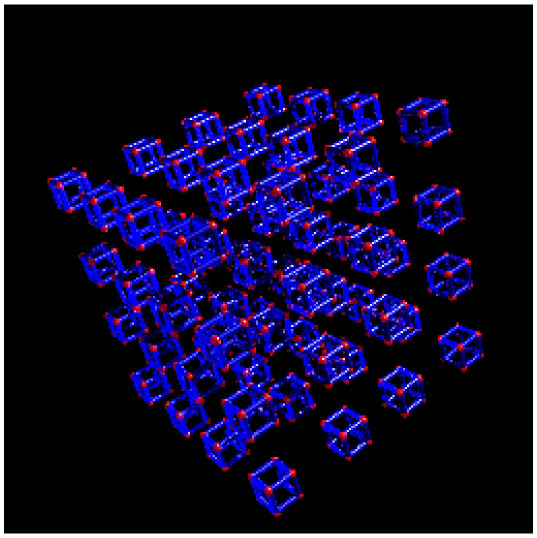
The majority of future work will involve improvements to the bounding of object movement. While Newton's method is too slow and interval algebra is insufficiently accurate, a combination offers potential. Guidance may be offered by [Nor05] where interval algebra is used as introduced in this work, but is extended to add checks to prevent error from increasing unacceptably. Additionally, more calculations done prior to establishing the interval may reduce the opportunity for error.

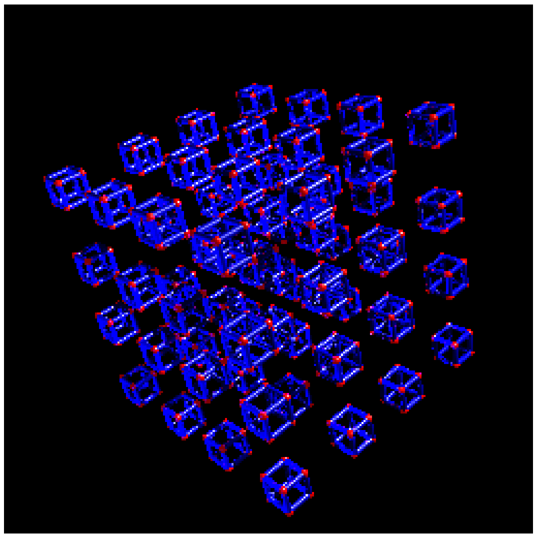
Because transformation of the camera and objects in the scene is directly related to temporal coherence, establishing a transformation metric may be beneficial in deciding if it is appropriate to use a method that takes advantage of temporal coherence. Prior to a scene being rendered, the transformation metric could be calculated and if below a certain threshold, then the temporal coherence is exploited. Scene that do not show sufficient promise relative to the metric could be rendered with standard techniques.

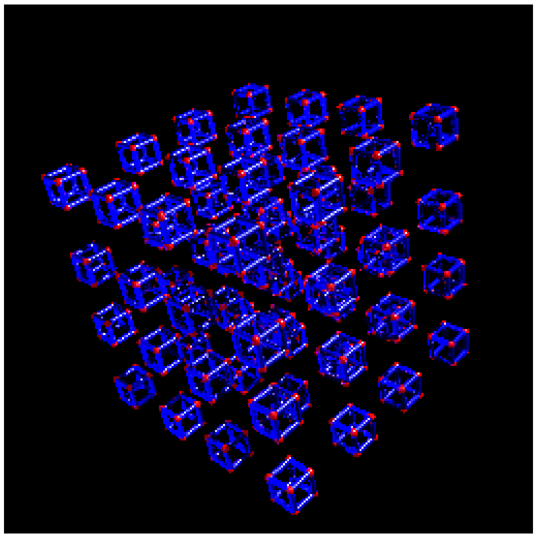
Another approach not implemented in this work would bound ray movement in the scene and perform a temporal ray bound-object bound intersection to establish initial temporal intersections. Because temporal adaptive grids require a ray to be static, when there is camera movement each object is moved inversely to the camera movement to simulate the camera movement. This increase the number of transformations performed on each object. If the camera is free to move and the rays are not static, a bound could be placed on the ray and intersected with the scene. This would reduce the transformations on the objects, and most likely would improve the object hierarchy. Unfortunately it is less elegant to implement. Further, the ray movement has the potential to create a very large bound.

It is unknown if optimally temporally bounded objects have the statistically beneficial properties for object hierarchy generation that regular scenes have. A study of the scene properties of objects as they move may be beneficial.

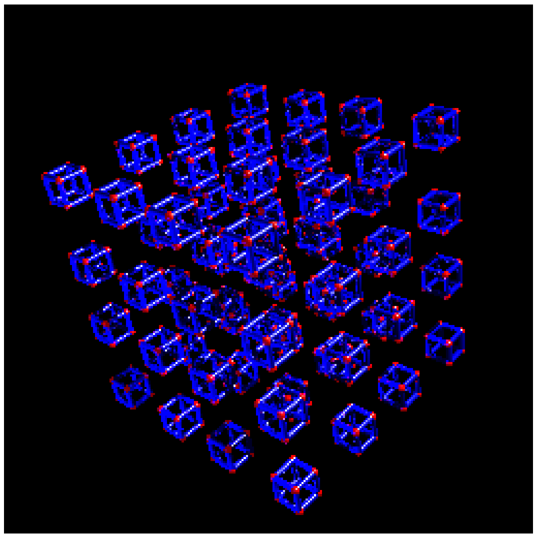
Appendix – Sample Frames from Animations.


Cube frame 0

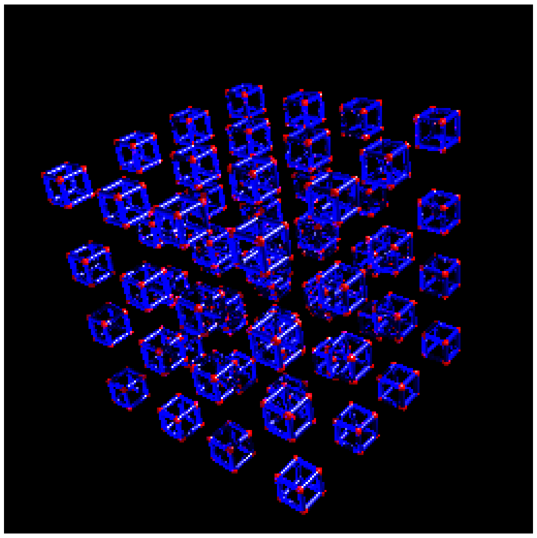

Cube frame 5



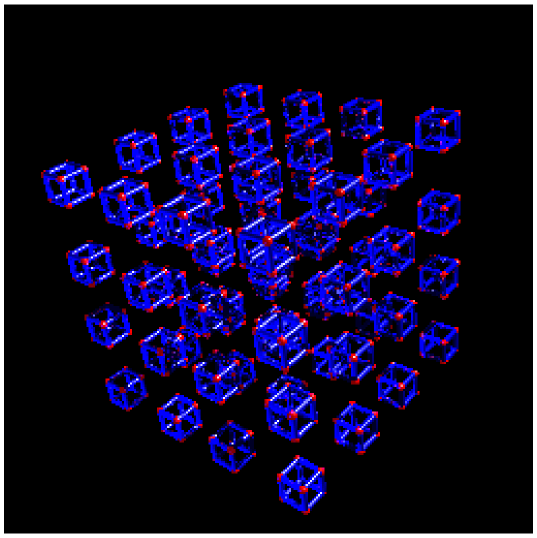
Cube frame 10



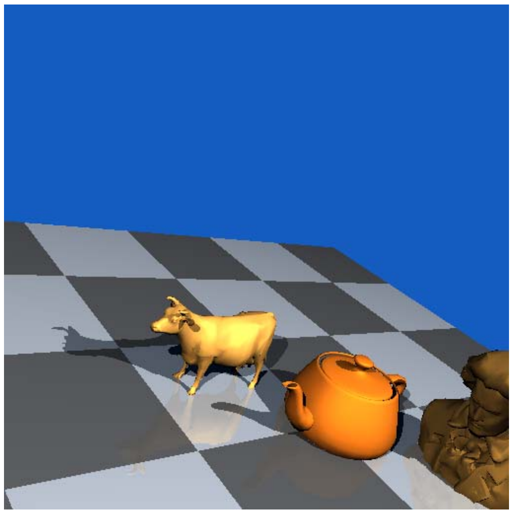
Cube frame 15

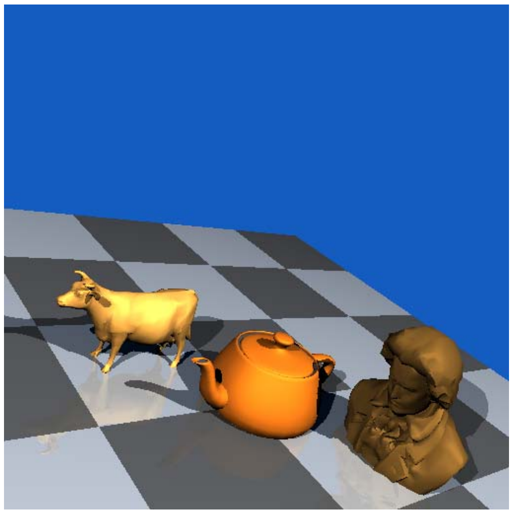


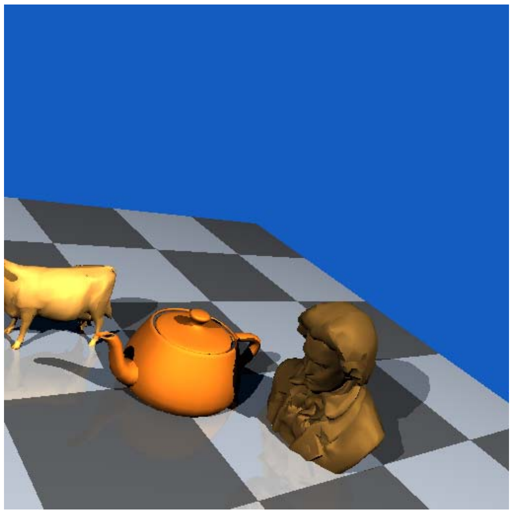
Cube frame 20



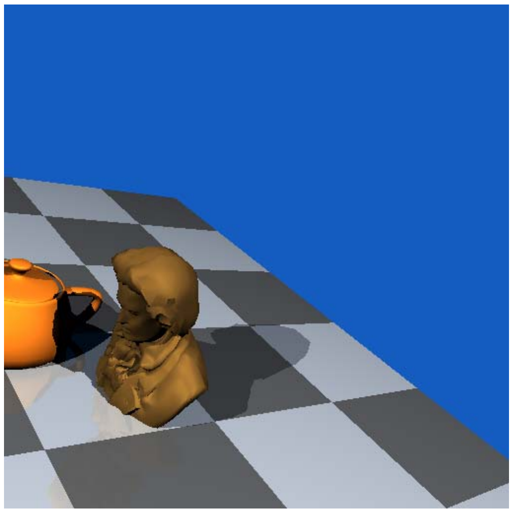
Cube frame 23

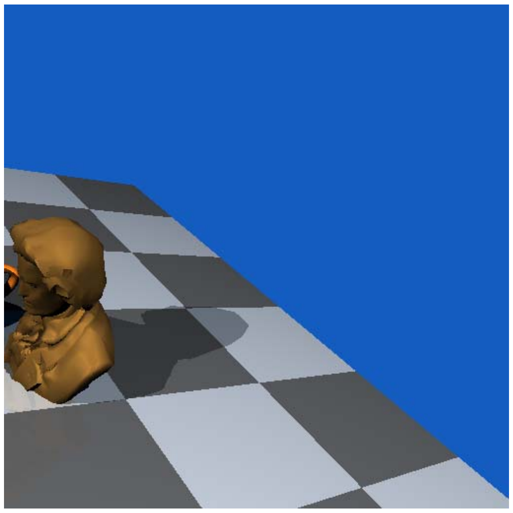

Common frame 0

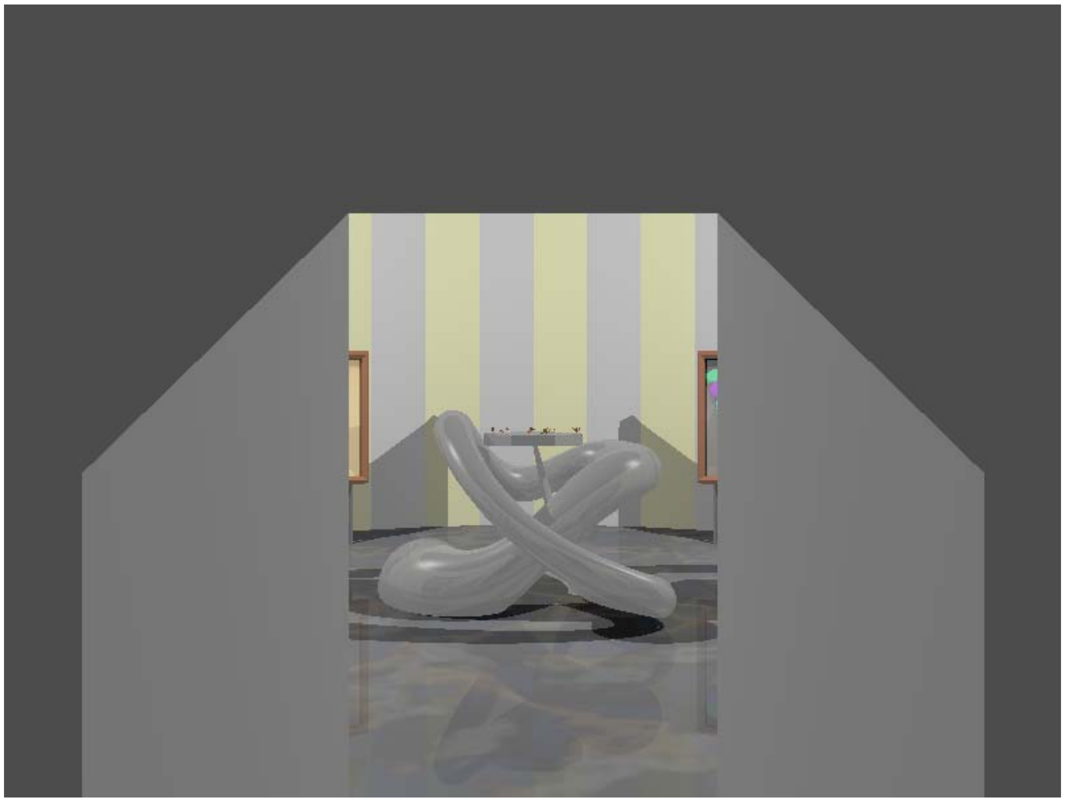

Common frame 100

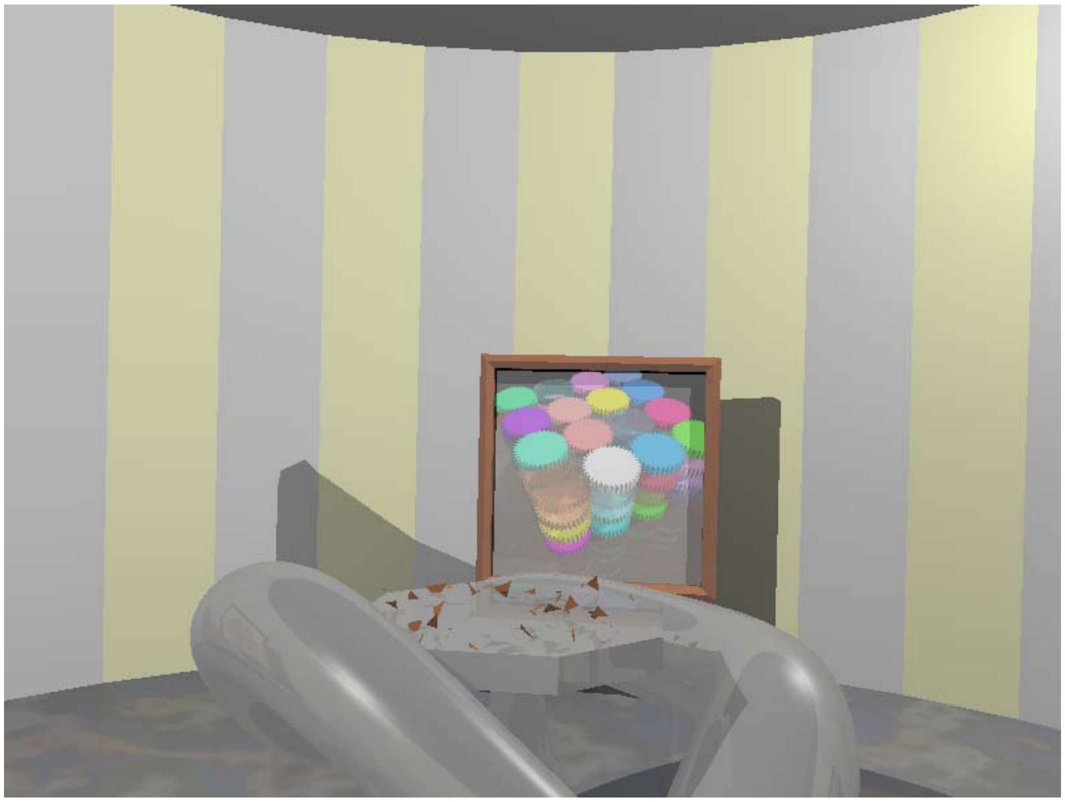

Common frame 200

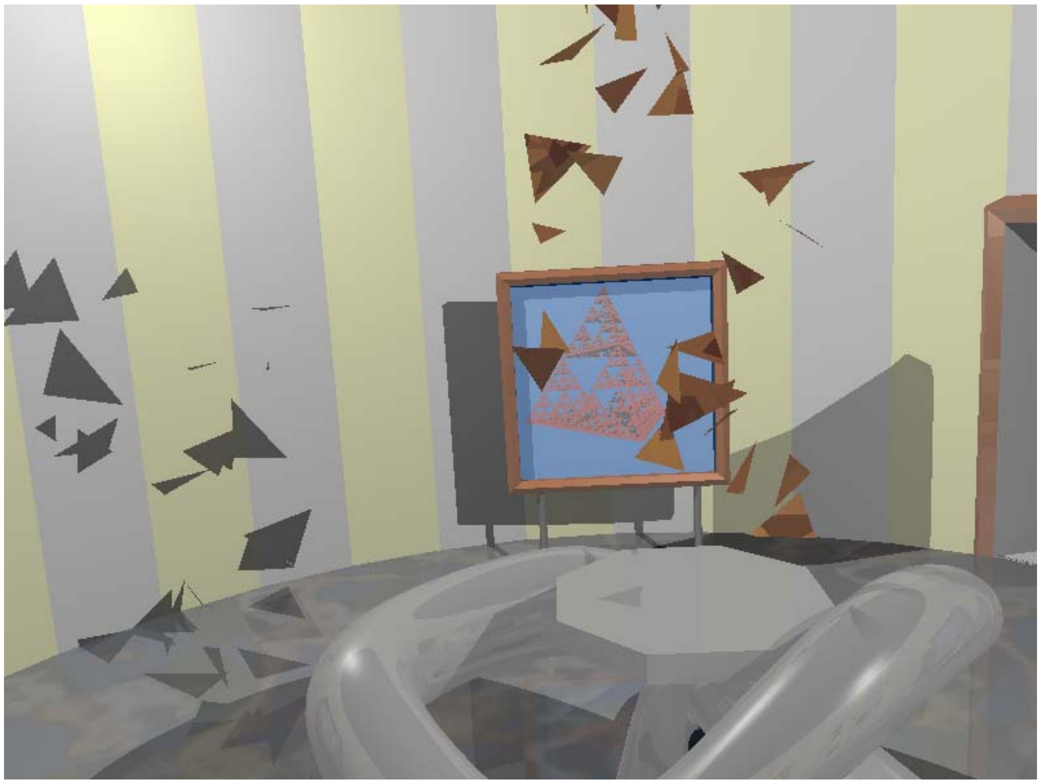
Common frame 300


Common frame 400

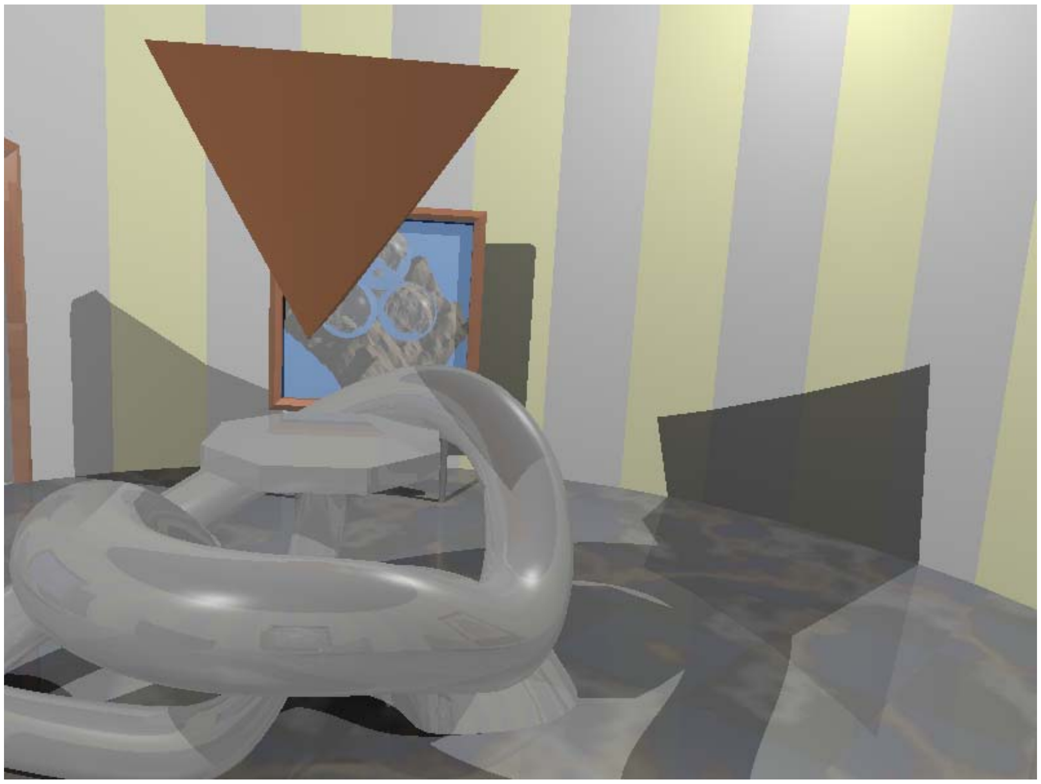

Common frame 499


Museum frame 0

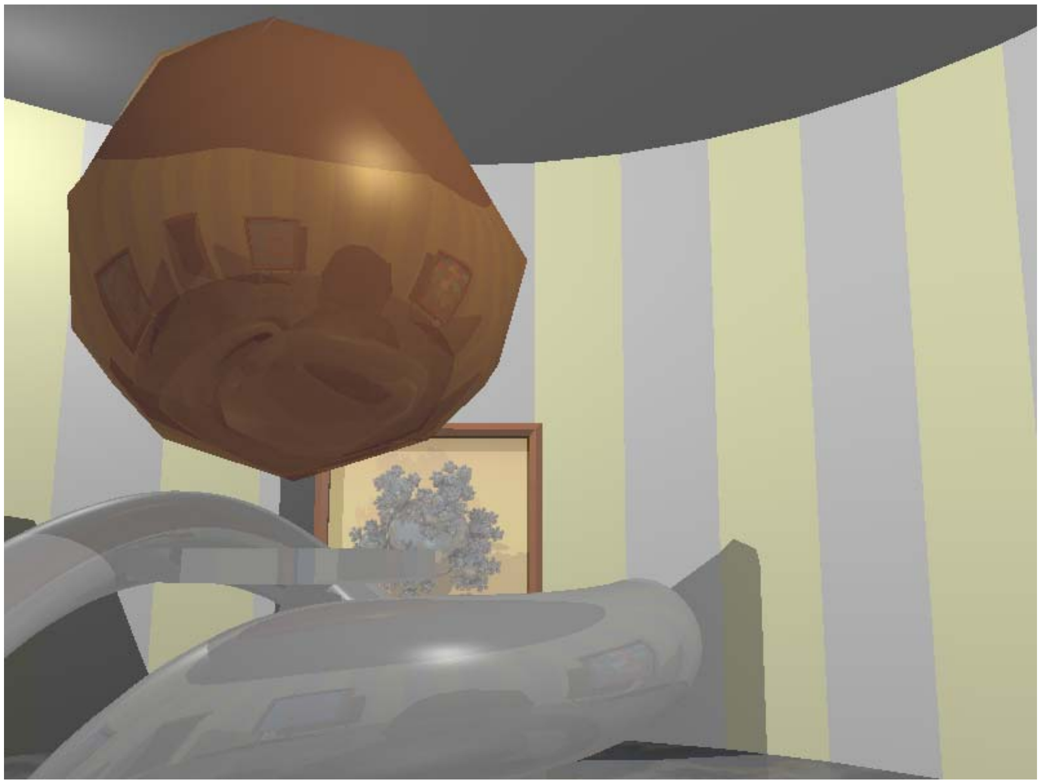

Museum frame 50



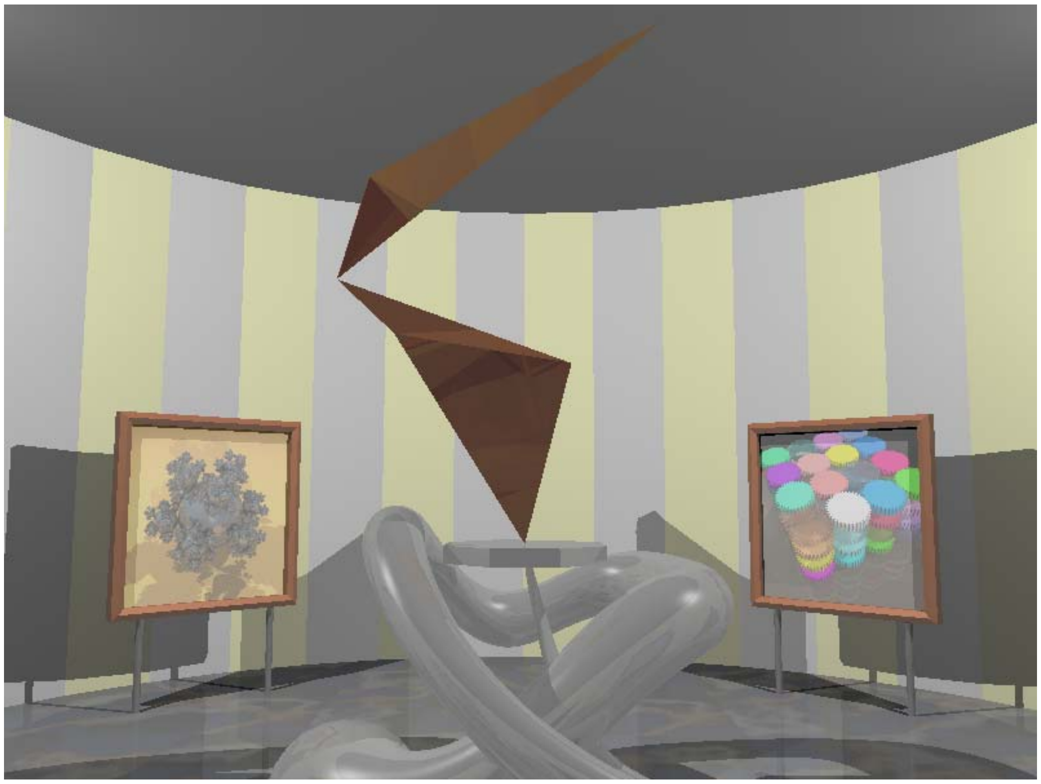
Museum frame 100



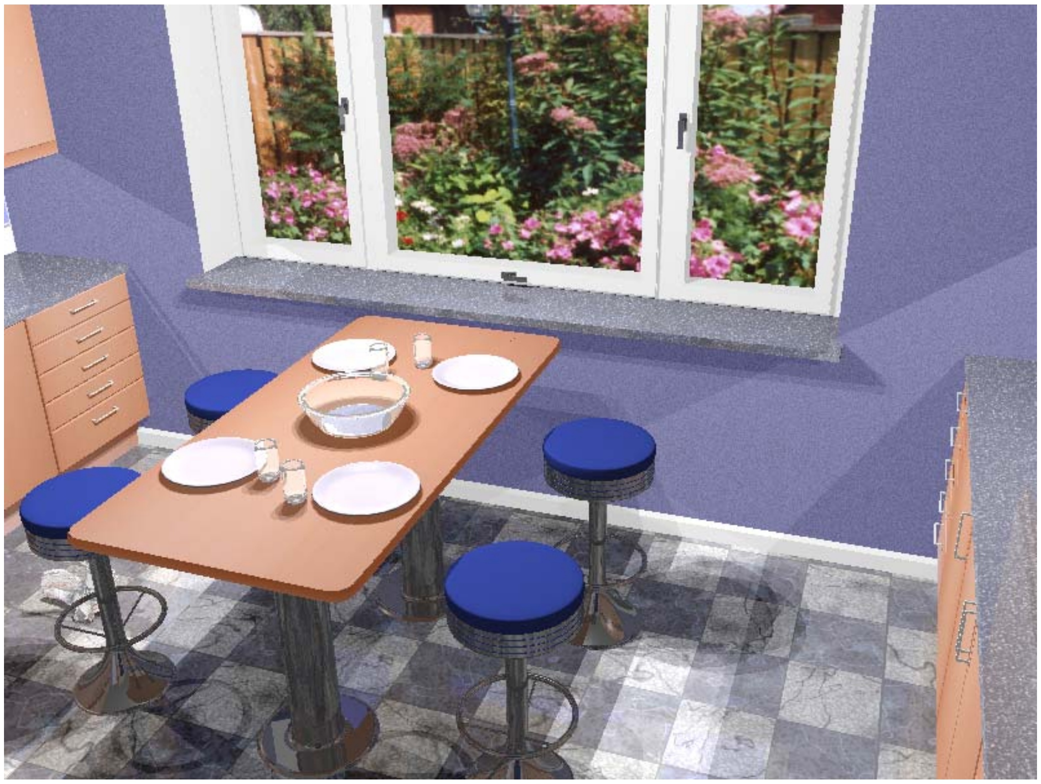
Museum frame 150



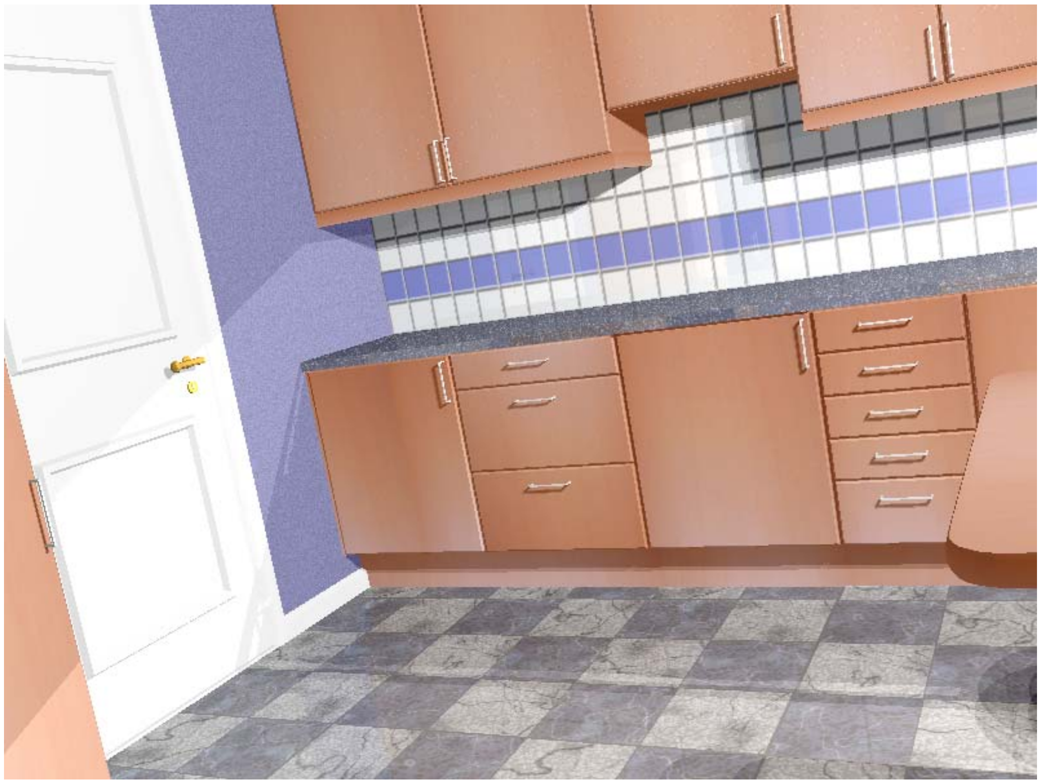
Museum frame 200



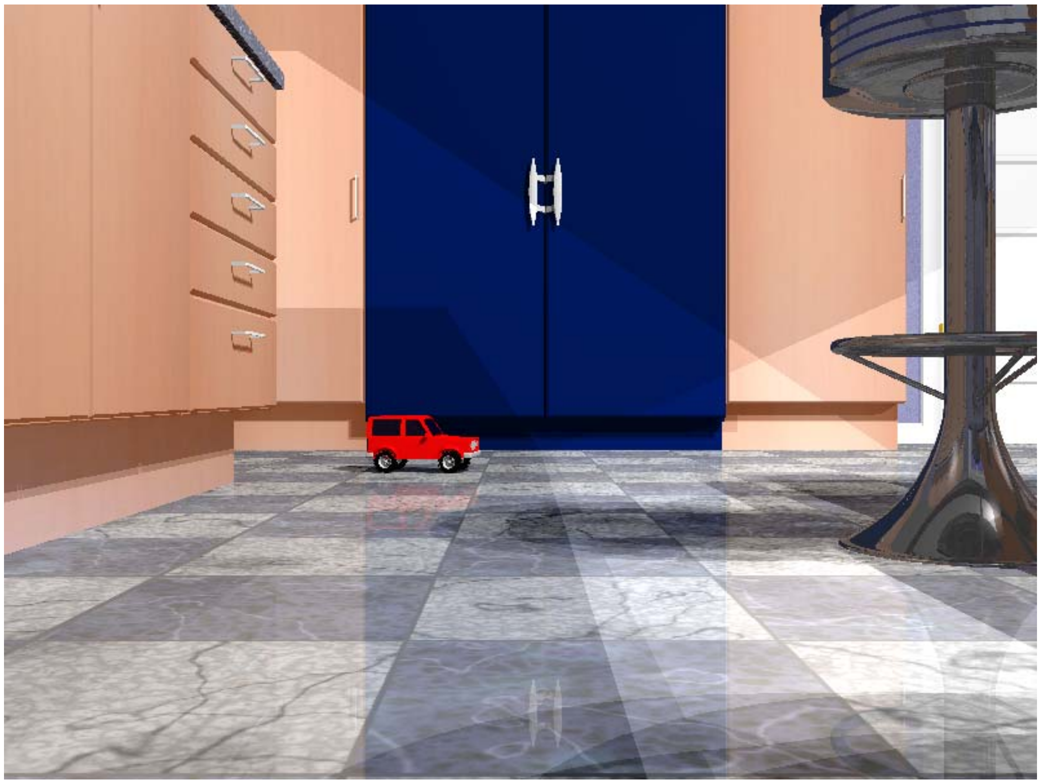
Museum frame 250

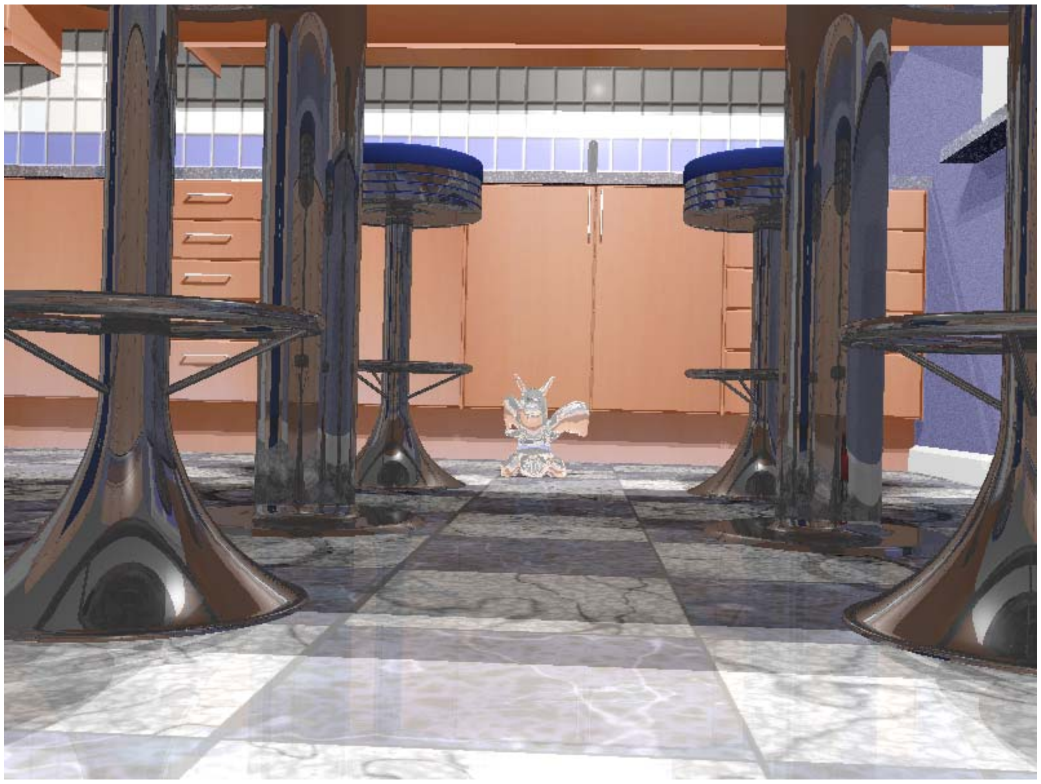


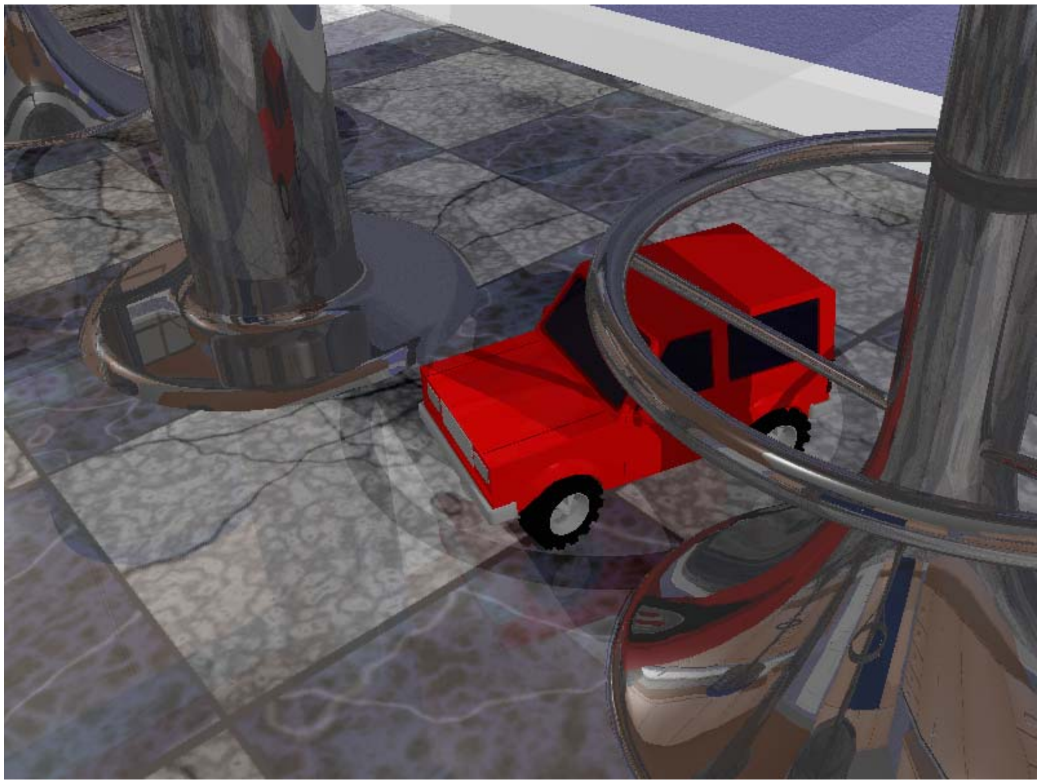
Kitchen frame 0

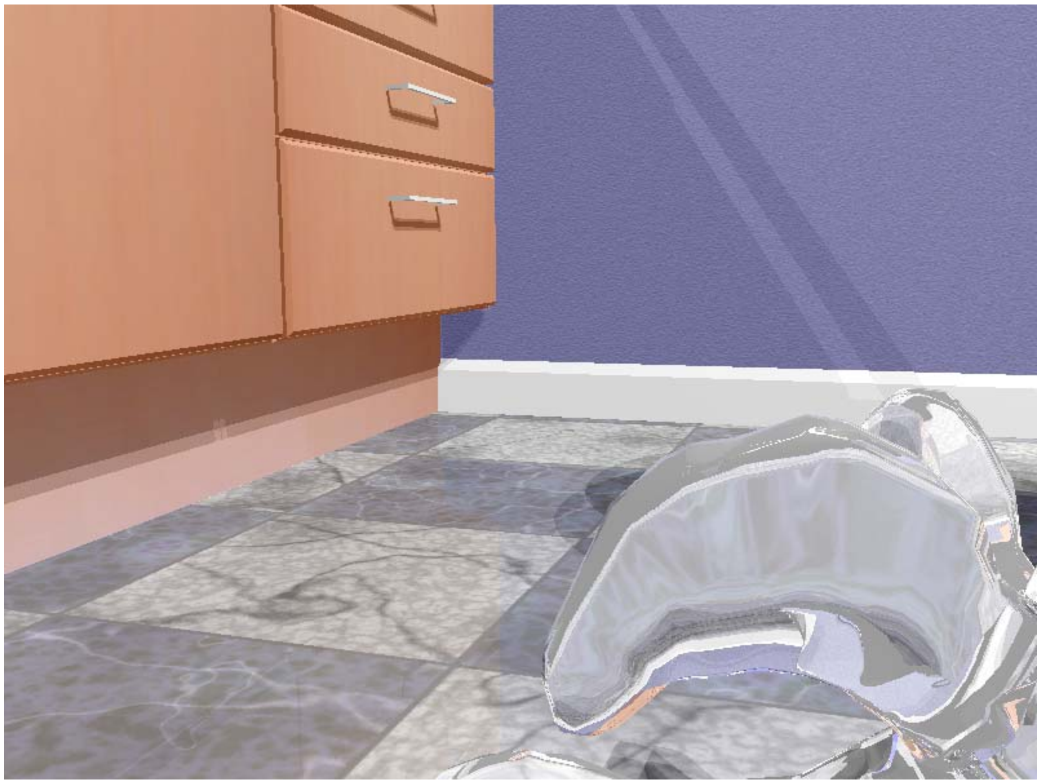


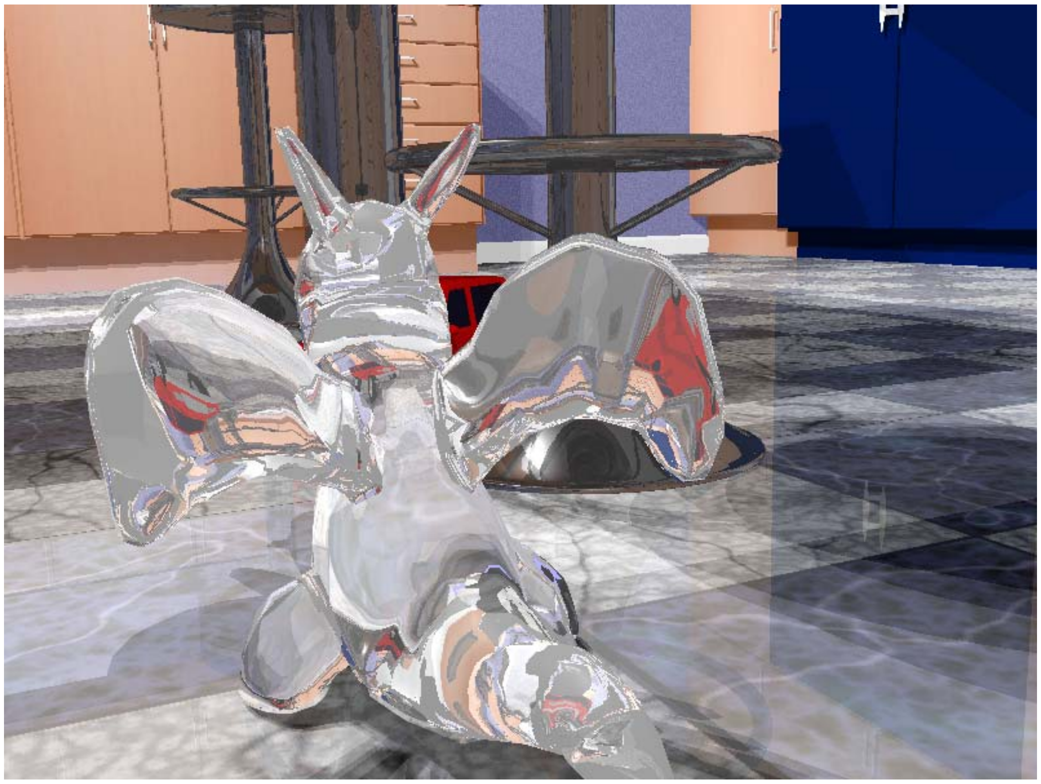
Kitchen frame 100


Kitchen frame 200

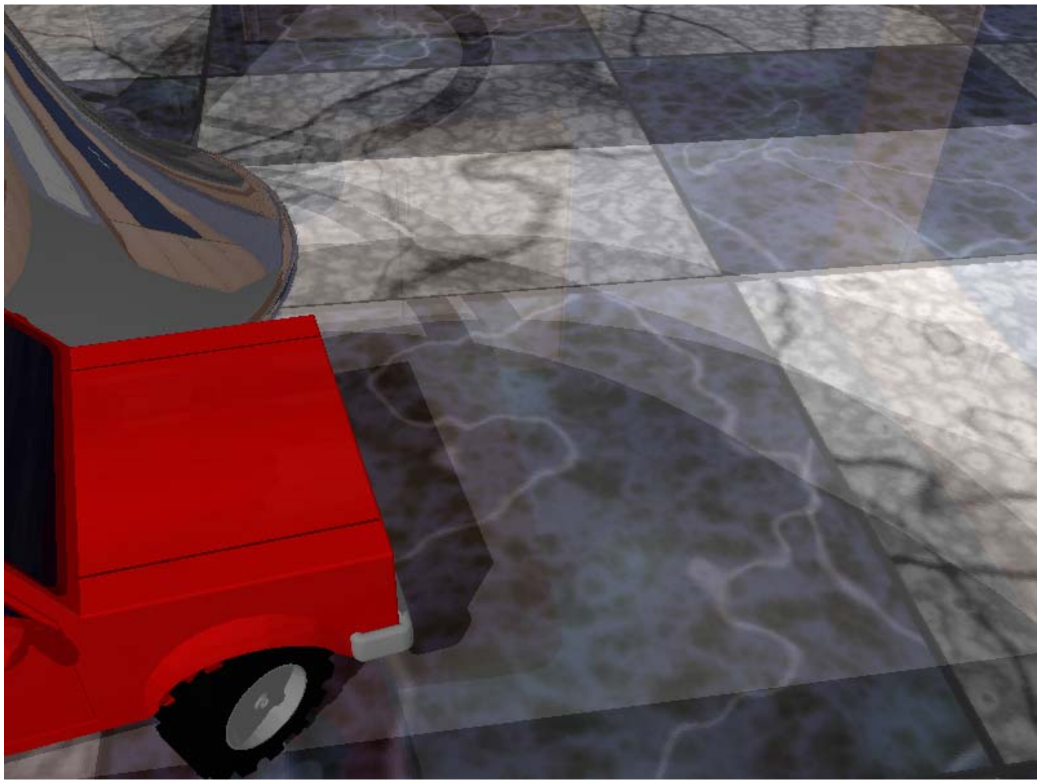

Kitchen frame 300


Kitchen frame 400


Kitchen frame 500



Kitchen frame 600



Kitchen frame 700

Kitchen frame 799

[AMS91] Taka-aki Akimoto, Kenji Mase, and Yasuhito Sueanga, “Pixel-Selected Ray Tracing,” IEEE Computer Graphics and Applications, 21(40):14-22, July 1991. Similar paper found in Eurographics ’89, pages 39-50, including author Akihiko Hashimoto.

[Bez72] Bézier, Pierre, “Numerical Control – Mathematics and Applications,” Wiley, London, England, 1972.

[BP03] Buena Vista Home Entertainment and Pixar Animation Studios, “Finding Nemo (Collector's Edition)” Filmmakers’ Visual Commentary Including Deleted Scenes And Recording Session, Chapter38 (minutes 54-56), November, 2003.

[CDP95] Cazals, Frederic, George Drettakis, and Claude Puench, “Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes,” EUROGRAPHICS ’95, pages 371-382, 1995.

[Coo86] Cook, Robert L., “Stochastic Sampling in Computer Graphics,” ACM Transactions on Graphics, 5(1):51-72, January 1986.

[CR74] Catmull, Edwin and Raphael Rom, “A Class of Local Interpolating Splines,” Computer Aided Geometric Design; edited by Barnhill, Robert E. and Riesenfeld, Richard F., Academic Press; New York, pages 317 – 326, 1974.

[Dev89] Devillers, Olivier, “The Macro-regions: an efficient space subdivision structure for Ray-tracing", EUROGRAPHICS ’89, pages 27-38, 1989.

[Eber00] Eberly, David, “3D Game Engine Design,” Morgan Kaufmann, September 2000.

[FTI86] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata, “ARTS: Accelerated Ray-Tracing System,” IEEE Computer Graphics and Applications, 6(4):16-26, April 1986.

[Fuj88] Fujimoto, Akira, “Turbo Beam Tracing - A Physically Accurate Lighting Simulation Environment,” Knowledge Based Image Computing Systems, May, 1988.

[Gla84] Glassner, Andrew S., “Space Subdivision for Fast Ray Tracing,” IEEE Computer Graphics and Applications, 4(10):15-22, October 1984.

[Gla88] Glassner, Andrew S., “Spacetime Ray Tracing for Animation,” IEEE Computer Graphics and Applications, 8(2):60-70, March 1988.

[GS87] Goldsmith, Jeffrey, and John Salmon, “Automatic Creation of Object Hierarchies for Ray Tracing,” IEEE Computer Graphics and Applications, 7(5):14-20, May 1987.

[GWS04] Guenther, Johannes, Ingo Wald, and Philipp Slusallek, “Realtime Caustics using Distributed Photon Mapping,” Rendering Techniques 2004, Eurographics Symposium on Rendering, June 21-23, 2004.

[Ham53] Hamilton, William Rowan, “Lectures on Quaternions: Containing a Systematic Statement of a New Mathematical Method,” Dublin: Hodges and Smith, 1853.

[HM76] Halas, John and Roger Manvell., “The Technique of Film Animation,” Focal Press, 1976.

[HS99] Havran, Vlastimil and Filip Sixta, “Comparison of Hierarchical Grids,” http://www.acm.org/tog/resources/RTNews/html/rtnv12n1.html, Ray Tracing News, 12(1), June 25, 1999.

[JW89] Jevans, David, and Brian Wyvill, “Adaptive Voxel Subdivision for Ray Tracing,” Proceedings of Graphics Interface ’89, pages 164-172, 1989.

[Kaj86] Kajia, James T., “The Rendering Equation,” Proceedings of SIGGRAPH ’86, Computer Graphics, 20(4):143-150, August 1986.

[Kap85] Kaplan, Michael R., “Space Tracing a Constant Time Ray Tracer: State of the Art in Image Synthesis,” SIGGRAPH ’85 Course Notes, 18(3):149-158, July 1985.

[KB84] Kochanek, Doris H. U. and Richard H. Bartels, “Interpolating splines with local tension , continuity and bias control,” Computer Graphics, 18(3):33–41, July 1984.

[Kli94] Klimaszewski, Krzysztof, S., “Faster Ray Tracing Using Adaptive Grids and Area Sampling,” doctoral dissertation, Brigham Young University, Provo, Utah, Dept. of Civil and Environmental Engineering, 1994.

[KS97] Klimaszewski, Krzysztof, S. and Thomas W. Sederberg, “Faster Ray Tracing Using Adaptive Grids,” IEEE Computer Graphics and Applications, 17(1):42-51, January 1997.

[KWC97] Klimaszewski, Krzysztof, S., Andrew Woo, Frederic Cazals and Eric Haines, “Additional Notes on Nested Grids,” http://www.acm.org/tog/resources/RTNews/html/rtnv10n3.html, Ray Tracing News, 10(3), December 2, 1997.

[LAM03] Lext, Jonas, Ulf Assarsson, and Tomas Möller, “BART: A Benchmark for Animated Ray Tracing.” http://www.ce.chalmers.se/BART/, Department of Computer Engineering, Chalmers University of Technology, Sweden. accessed Apr 1 2005, last updated June 2, 2003

[Nor05] North, Nicholas, “A Robust Algorithm for Curve/Surface Intersection,” 19th Annual Spring Research Conference 2005, College of Physical and Mathematical Sciences, Brigham Young University, CS5(c), March 19, 2005.

[Ove68] Overhauser, Albert W., “Analytic Definition of Curves and Surfaces by Parabolic Blending,” Technical Report No. SL68-40, Ford Motor Company Scientific Laboratory, May 8. 1968

[Pho75] Phong, Bui-Tuong, “Illumination for Computer Generated Pictures,” Communications of the ACM, 18(6):311-317, June 1975.

[RW80] Rubin, Steven M., and Turner Whitted, “A 3-Dimensional Representation for Fast Rendering of Complex Scenes,” Proceedings of SIGGRAPH ’80, Computer Graphics, 14(3):110-116, July 1980.

[SB02] Stoer, Josef and R. Bulirsch, “Introduction to Numerical Analysis,” Springer-Verlag, 2002.

[Sho85] Shoemake, Ken, “Animating Rotation with Quaternion Curves,” Proceedings of SIGGRAPH ’85, Computer Graphics, 19(3):245-254, July 1985.

[War69] Warnock, John E., “A Hidden-Surface Algorithm for Computer-Generated Half-Tone Pictures,” Technical Report TR 4-15, NTIS AD-733 671, Computer Science Department, University of Utah, Salt Lake City, June 1969.

[Whi80] Whitted, Turner, “An Improved Illumination Model for Shaded Display,” Communications of the ACM, 23(6):343-349, June 1980.