1 Introduction

Animation is the creation of the illusion of motion by displaying images in rapid succession—typically 25 or 30 frames per second. The naïve way to produce an animation sequence is to render and record each frame separately, using optimization techniques applicable to still image rendering, e.g. [11, 13, 21, 26]. However, when rendering computer graphics images within an animation sequence, it is possible—and desirable—to utilize additional optimization methods, unique to animation.

Desirable—because there are numerous frames in a typical animation sequence, and therefore the rendering of the entire sequence may take a very long time (e.g. weeks for a film of a few minutes). Even if we can afford to wait several hours for a single image, generally we wouldn’t be willing to wait that long if the image will later be displayed for only 1/25 of a second. Furthermore, since most computer graphics today are rendered within animation sequences, each improvement in animation rendering will bring about considerable savings in computer resources.

Possible—because the difference between consecutive frames in an animation sequence is relatively small. If we inspect a typical image in a sequence, we will find that large areas in it did not change at all relative to the previous image, or changed very little (e.g. slight movement or deformation and slight change of color). The significant differences between frames will generally be concentrated in a few restricted regions, whose total area is small compared to the entire image.

This property—the similarity between consecutive images in an animation sequence—is called temporal coherence. In fact, the similarity is not just between the images themselves, but also between the scene model states at consecutive frame times: the object forms and locations, surface properties, light sources and viewpoint all change gradually, and are almost identical at close points in time. This kind of similarity is also called temporal coherence. To distinguish between the two types of coherence, the former—similarity between the images themselves—is called image-space temporal coherence, while the latter is commonly known as object-space temporal coherence. Both kinds of coherence may be utilized to save computations when rendering images as part of an animation sequence.

Temporal coherence has been used in various aspects of animation rendering: visible surface determination, ray-object intersection, ray tracing and radiosity. The major works in each field are surveyed in the following sections.

2 Visible Surface Determination

Visible surface determination (also known as hidden surface removal) may be the oldest use to which temporal coherence has been applied. As pointed out in Foley et al.’s textbook [10, Chapter 15.2, pp. 657, 664], since the model surfaces and the viewpoint move little between consecutive frames,
the visible surface portions can be expected not to change much either. It is suggested in [10] that this coherence can be used to optimize visible surface determination in animation sequences; however, no specific algorithm is given there.

2.1 Convex Scenes

The first known temporal coherence algorithm was presented by Hubschman and Zucker in [19]. The algorithm works only for a very restricted class of models: convex, non-intersecting, finite polyhedra. Furthermore, the scene is static throughout the animation sequence; the only movement allowed is that of the viewpoint.

From these assumptions, it follows that the visible parts of a polyhedron A are contained in a boundary composed of A’s silhouette edges (edges where a forward-facing and a backward-facing polygon meet), and of the projections of occluding polyhedra’s silhouette edges onto A, relative to the viewpoint. It also follows that A can become (or cease to be) partly occluded by polyhedron B only when the viewpoint crosses a support plane—a plane tangent to both A and B.

The algorithm begins by finding all support planes as a preprocessing stage. Then, for each frame in the animation sequence, the partly occluded objects can be found by considering which support planes were crossed by the viewpoint since the previous frame; the silhouette edges are found by traversing the polyhedra’s graph representations, starting with the previous frame’s silhouette edges.

The visible surfaces are found in time $O(EP)$ per frame, where $P$ is the number of polyhedra in the scene and $E$ is the number of edges. Unfortunately, the algorithm is highly dependent upon the convex polyhedra assumption, and cannot be easily modified for more general models.

2.2 General Polyhedral Scenes

Tost [28] described a hidden surface removal algorithm, applicable to polyhedral scene models. No object deformations are allowed; object and viewer movements should be piecewise-linear, and known in advance.

Under these restrictions, the algorithm finds the points in time when the occlusion relationships between the objects change, or when object faces turn from forward-facing to backward-facing or vice versa. At each such point in time, the occlusion relationship is updated incrementally; between these points, hidden surface removal is greatly simplified, because the occlusions do not change.

The speedup attained by Tost’s technique ranges from 1.2 to 333. It should be noted that this fantastic speedup is just for the visibility calculation between the occlusion change times (not for the entire rendering process of the animation sequence), and that it is only achieved for scenes containing a small number of simple objects which move very slowly.

2.3 Hierarchical Z-Buffer Visibility

The main purpose of Greene et al.’s hierarchical Z-buffer visibility algorithm [16] is to determine the visible surfaces of large, complex scenes in time proportional to the number of visible surfaces, rather than the total number of surfaces in the scene (as in the traditional Z-buffer algorithm). This is achieved by combining octree subdivision of the object space with an image-space pyramid built over a Z-buffer. If an octree partition is completely hidden by objects which were considered previously and are already in the Z-buffer, then all of the sub-partitions and objects in that partition can be ignored.
Each element in the pyramid contains the farthest Z value (distance from the viewer) of the pixels corresponding to that element. This enables fast rejection of many objects and octree partitions: an object may be ignored if its nearest point is farther away than the Z value associated with the smallest pyramid area containing the object’s projection.

To utilize temporal coherence when rendering animation sequences, the visible octree partitions are remembered each time a frame is produced. For the next frame, the algorithm initially renders all the objects in the partitions remembered from the previous frame, and uses their depths to create the initial Z pyramid. This pyramid is expected to reject much more objects and octree partitions than a Z pyramid built from scratch, because most of the visible objects have already been rendered.

The speedup found by Greene et al. for this temporal coherence method is 1.5–2.

3 Ray-Object Intersection

When ray tracing a scene, most of the processing time is spent on ray-object intersection queries: given a light ray and a collection of objects in 3D space, find the first object intersected by the ray. There are numerous algorithms to answer this query efficiently in a static scene; most such algorithms utilize object coherence by dividing space into a set of volumes, and testing a ray only against objects which intersect some volume penetrated by the ray. See, e.g., [9, 13, 15].

Of course, all these technique are applicable to ray tracing animation sequences. However, greater speedup may be achieved by utilizing temporal coherence. For example, since the model objects move gradually, the same division of space into volumes might be usable in consecutive frames.

3.1 4D Space-Time Techniques

Both Glassner [14] and Gröller and Purgathofer [17] suggest regarding a dynamic scene in 3D as a static one in 4D. This requires a priori knowledge of all the trajectories and the deformations that the objects will undergo during the animation sequence. Given this information, data structures for efficient 3D ray-object intersections can be extended to 4D by adding time as the fourth dimension.

Glassner generalizes traditional 3D hierarchical bounding volume techniques to 4D hypervolumes. This is more general than simply assigning a validity period to each 3D bounding volume, because some of the faces of the 4D bounding hypervolumes can be neither parallel nor perpendicular to the time axis, giving a bounding volume some ability to “follow” its enclosed object as it moves. An implementation of this technique achieved 1.25–2 speedup for simple test scenes. The cause of this speedup is that the bounding volumes are computed once, in a preprocessing stage, rather than repeatedly for each frame.

Gröller and Purgathofer extend Arvo and Kirk’s ray classification method [1] to utilize temporal coherence in animation of CSG models. The original method involves hierarchically subdividing the 5-dimensional space of all possible light rays (3D origin point + two direction angles), and finding the set of objects potentially intersected by rays in each subdivision partition. The extension to this algorithm adds a time of generation to each ray, making the entire ray space 6-dimensional. Additionally, with each CSG object found for some ray-space subdivision partition, the algorithm keeps 3D bounding volumes which contain the object for various periods of time, ranging from a single frame to the entire period represented by the partition. These bounding volumes allow the partition to be split efficiently.
The speedup attained by an implementation of this technique on two simple test scenes was 1.6-1.8.

### 3.2 The Look-Ahead Algorithm

Müller’s look-ahead algorithm [23] requires the model and the trajectories of the objects and the viewpoint to be known in advance for the duration of the animation sequence. It finds validity periods for ray-object intersections by examining kinetic hulls, the three-dimensional traces of surfaces over a period of time. This saves intersection tests for various kinds of rays: direct viewing rays, shadow, reflection and refraction rays.

The detailed formulation of the look-ahead algorithm, as given in [23], is rather cryptic. Essentially, it finds an upper limit for each ray-object intersection’s validity period by executing some efficient static-scene intersection algorithm (e.g. one of those listed at the beginning of Section 3) on scenes composed of the kinetic hulls of the original scene’s moving surfaces over exponentially increasing periods of time: one frame, two frames, four frames etc. The intersection’s exact validity period is then found by binary search.

The calculation of kinetic hulls may not be trivial for certain surfaces and movements. One suggested compromise is to replace the kinetic hulls by simpler approximations, which may be easier to find and to intersect with rays.

### 3.3 Compiled Ray Tracing

Most animation sequences contain both time-dependent and time-independent properties. A time-independent property is one which holds for the entire duration of the sequence, and is implied by the model. For example, a rigid object’s geometry is time-independent, while its position may be time-dependent.

Ray tracers usually act as interpreters, evaluating both the time-dependent and the time-independent attributes for every frame. In contrast, Hödicke’s CRT (Compiled Ray Tracing) system [18] compiles some of the time-independent aspects of the scene into parallel C language programs. The result is a specialized ray tracer, specifically suited for the particular scene at hand. This ray tracer then interprets the time-dependent aspects to render the animation sequence. Performance is improved over conventional ray tracers because of the usual speed advantage of compilation vs. interpretation.

CRT generates a separate program for each object in the scene model. The input and output of this program contain the world space coordinates of rays, as well as the first known intersection of each ray with an object. The program transforms each input ray to the coordinate system of the program’s object, and checks if the ray intersects the object. If it does, and the new intersection point is closer to the ray’s origin than the old one, the ray’s intersection data are updated accordingly. The ray is then transformed back to world coordinates and output.

The generated programs use an eager variation of Arvo and Kirk’s ray classification algorithm [1] for efficient calculation of ray-object intersections: the 5D space of all possible rays is hierarchically subdivided for each generated program. For each subdivision partition, the set of object faces which may be intersected by rays in the partition (or a small superset of that set) is found. Subdivision continues to some maximum depth, or until the set of faces associated with a partition is smaller than a minimum threshold.

After all the programs have been generated, CRT executes them as parallel processes, and connects them to form a ray-object intersection pipeline. If the number of processors allows, more than one process may be spawned for each program.
The technique is applicable to rigid objects—no deformations are allowed during the animation sequence. For the scenes tested, CRT achieves 1.3–1.4 speedup per frame, compared to a commercial ray tracer running on the same hardware. However, the generation of C language code is a considerable overhead. For the test scenes, the break-even point (the least number of frames for which CRT would be more efficient than a commercial renderer) is 15–22 frames.

3.4 Object-Space Algorithms

Chapman et al.’s object-space algorithm [5] computes the object intersections of each ray for the entire duration of the animation sequence, rather than separately for each frame. This yields a validity period for each ray-object intersection. Performance is improved because the validity periods are usually longer than one frame.

In the algorithm’s preprocessing stage, a 3D bounding volume is constructed for every object. These volumes are static, and large enough to contain their respective objects throughout the sequence.

After preprocessing, the algorithm deals with each pixel for the entire duration of the animation sequence before moving on to the next pixel. For every pixel, its primary ray is intersected with the scene objects. The start and end times of each intersection, as well as the trace left by the ray across the object’s surface, are calculated. The bounding volumes created at the preprocessing stage are used to accelerate this step.

After the intersections of the ray with objects are found, they are sorted by their start times. From each intersection’s validity period, the algorithm discards the periods during which it is occluded by some other intersection; the remainder are the periods during which the intersection is the one closest to the ray’s origin. If the intersection spawns further secondary rays, they are traced, recursively, for the intersection’s validity period (rather than for the entire sequence). Finally, the ray’s originating pixel is shaded for the validity period, using the trace information.

This algorithm is only suitable for production of off-line animation sequences, because it does not render all of the pixels of a single frame consecutively. It is restricted to flat, polygonal models, and object transformations are restricted translation only (without rotation, scaling and deformation), to simplify the calculation of the traces.

The algorithm was tested on one non-trivial scene, and achieved speedup by a factor of 7.9. Performance may be poorer for scenes with numerous reflections and refractions, since in this case the validity period of each intersection shortens and approaches a single frame.

4 Ray Tracing

The algorithms described in the previous section address the problem of ray-object intersection query in ray tracing. Those techniques accelerate the calculation of a pixel’s color, but every pixel in each frame must still be calculated separately. It is possible, however, to accelerate the entire ray tracing process, rather than just the ray-object intersections. For example, some methods look for pixels (or for entire regions within the image) which retain their color between frames, and therefore needn’t be ray traced at all. More generally, image regions may be found to have changed, but in a consistent and predictable way. For instance, an entire area in the image may move 5 pixels to the left and become 2% darker.

Methods which find unchanged (or consistently changed) image regions are also suitable for compression of animation sequences: instead of storing each frame entirely, only the differences from its predecessor can be maintained.
4.1 Image-Space Algorithms

Badt's image-space temporal coherence algorithm [2] is one of the first attempts at utilizing temporal coherence in ray tracing. As its name implies, this algorithm exploits image regions which remain unchanged between frames by examining the images themselves; this is useful mainly when the viewpoint does not change from frame to frame, and only a few objects in the scene move or change.

The first frame of an animation sequence is rendered fully. In each succeeding frame, every pixel is initially assigned the same color as the corresponding pixel in the previous frame. A random sample of the pixels is then chosen and ray-traced. For each randomly chosen pixel, if the consequent color differs from the initial color by more than some small, fixed threshold, then the rendering process is flooded from that pixel to neighboring pixels, both in the same frame and in preceding and succeeding frames. The flooding continues until it reaches pixels whose color changed by less than the threshold. If previously rendered frames cannot be changed, due to storage or application limitations, then rendering is flooded only to following frames; this is called half-flooding (as opposed to full-flooding, where rendering is also flooded to preceding frames).

This simple algorithm was not tested. It is expected to produce inaccurate results, because some changed image regions may be missed, especially if they are small and short-lived. Such a region can occur if a small object moves rapidly across the frame. In the case of half-flooding, a changed image region may be found and corrected after several frames, causing flicker. The pixel sampling rate can be increased to minimize these problems, at the expense of reduced savings relative to full rendering.

Chapman et al.'s image-space algorithm [4] is based on the assumption that it is possible to find an upper bound on the frequency of visual phenomena occurring in an animation sequence, or perceived by a human viewer. Let this bound be \( n \) frames, where \( n > 1 \). Then it is sufficient to initially render just one in every \( n \) frames. If a pixel has the same color in frames 1 and \( n \) of an \( n \)-frame sequence, then, by the assumption, that pixel can be given the same color in all intervening frames. Otherwise, the \( n \)-frame interval is binary searched to find the frame at which the pixel changes color: the pixel is rendered at frame \((n+1)/2\), and the then process continues recursively at intervals \([1, (n+1)/2]\) and \([(n+1)/2, n]\).

This algorithm is applicable to any model, but it is suitable only for off-line animation rendering, since it does not produce all of the pixels of a frame consecutively. Inaccurate results may be produced, because a pixel may have the same color in two images \( n \) frames apart, and still have a different color in intermediate frames. This might happen, for example, if a small object moves rapidly across the field of view. It is interesting to note that Badt's image-space temporal coherence algorithm may fail under the same conditions. It appears that image-space algorithms generally fail for such sequences, since it is impossible to accurately calculate the color of "missing" pixels by analyzing the images alone.

According to empirical studies reported in [4], for non-trivial sequences, less than 50% of the viewers noticed the difference between the exact sequence and that produced by the image-space algorithm if \( n \leq 8 \); the speedup for \( n = 8 \) was 1.56–3.12.

4.2 The Reprojection Algorithm

Badt's reprojection algorithm [2] is most useful when his image-space temporal coherence algorithm is not, i.e. when the viewpoint moves and the objects are static. The algorithm was initially designed for scenes containing only diffuse reflectors. The first frame of an animation sequence is
jitter-sampled: each pixel is considered a rectangle, and the ray cast for that pixel goes through a randomly chosen point within the rectangle. The first intersection of each ray with an object, called a hit-point, is found and recorded, together with the color calculated for the ray’s pixel.

In the next frame, the hit-points are reprojected to the new image plane by a perspective transformation, and two images are produced: the reprojection image, every pixel of which has the average colors of all hit-points reprojected onto it; and the hit-point count image, in which the “color” of each pixel is the count of hit-points reprojected onto it. The hit-point count image is then filtered to reduce noise. Pixels whose value in the filtered hit-point count image differs from the expected average value (one reprojected hit-point per pixel) by more than some fixed amount are considered “bad pixels,” because they are suspected of viewing a part of the scene which was either revealed or concealed between the frames. Each such bad pixel is ray-traced, and its resulting color is written into the reprojection image. The updated reprojection image is the new frame.

If not all of the surfaces in the scene are diffuse reflectors, additional information must be stored with each hit-point to calculate specular highlights. For Phong shading, this information includes directions of reflected light from light sources and Phong surface exponent. If the scene also includes reflections and refractions, then the surface normal must also be associated with every hit-point, and additional rays have to be cast from each hit-point on a reflective or refractive surface.

The reprojection algorithm was tested on a single pair of frames, and achieved a speedup of 2.4 for the rendering of the second frame. It should be noted that the proposed solution for reflective or refractive objects amounts to full ray tracing, with its associated high computational cost. No improvement in performance is expected for such objects.

### 4.3 Image Subdivision

Jevans [20] describes an object-space temporal coherence algorithm suitable for animation sequences where the viewpoint is stationary. The key idea behind this algorithm is to keep track of which space subdivision voxels are viewed by each pixel, and which objects are intersected by every voxel. This enables the identification and re-rendering of just those pixels which potentially view changed objects.

In practice, it would take too much storage space and time to find the exact pixels in which the image changed between frames. Instead, the image plane is divided into fixed-size blocks of pixels, and the algorithm finds which blocks may have changed since the previous frame: an image block can change only if it views some changed voxel, either directly or indirectly by a secondary ray. (A changed voxel is one in which an object appeared, disappeared, moved, deformed or changed its surface attributes.) Image blocks which may have changed are re-rendered; the rest of the blocks are copied from the previous frame.

Essentially, any subdivision scheme may be used; the algorithm was implemented with adaptive, hierarchical voxels and a $16 \times 16$ image block grid. It was tested on a single animation sequence, 351 frames long. Performance improves gradually during the sequence, achieving 15.2 speedup for the last frame and average 3.7 speedup for the entire sequence. This impressive improvement is partly due to the scene chosen for the test: in this scene, just a small part of the image changes from frame to frame, and there are little or no reflections and refractions.

### 4.4 Image Interpolation

Maisel and Hégron [22] present an object-space temporal coherence technique which interpolates image regions between key frames. The image transformation parameters for regions influenced by
a single object are calculated directly from that object’s motion; for other regions, approximate linear interpolations are computed from the regions’ positions in the key frames.

The algorithm renders key frames at regular intervals through the animation sequence, and maintains the ray tracing tree of each key-frame pixel. After producing two key frames, the velocity of each pixel’s image-plane movement is computed. A pixel is said to be influenced by an object if the object intersects any ray initiated by the pixel, whether direct-viewing, shadow-testing, reflection or refraction. If a pixel is influenced by a single object in the first key frame, then its velocity is computed analytically from the object’s and viewer’s velocities, and used to generate the corresponding pixels between the key frames.

Pixels influenced by more than one object are grouped into connected regions such that all pixels in a region have the same ray tracing tree, i.e. they are influenced by the same objects and in the same order. The edges of each region are found by an edge-detection algorithm, and an affine transformation is found between pairs of key frame regions with identical ray tracing trees. This affine transformation is used to interpolate image regions between the key frames.

This technique is applicable only if object and viewer trajectories are known in advance, because each key frame (except the first) is calculated before the intermediate frames which precede it in the animation sequence. The trajectories have to be continuous, and the objects must be polyhedra.

There are several potential sources of inaccuracy in the method. For pixels influenced by more than one object, the transformation of image regions is linear with respect to time, and therefore inexact. (As an example of a scene where linear image interpolation is inaccurate, consider a textured, revolving cylinder. The compression of the image of a region on the cylinder is not time-linear: it is quickest when the region is just rotating into view or out of view, and slowest when the region is facing the viewer.) Worse yet, the calculation of the transformation itself is inaccurate for objects whose visible parts change between key frames. For instance, a static background texture seen through the hole of a moving torus will be rendered as moving with the torus! Furthermore, since there are two different image region interpolation schemes, seams may be visible between regions interpolated by different schemes.

The reason these deficiencies were not detected in [22] may be that the test scene used was very simple, containing a single reflective surface. In more typical scenes, most pixels might be influenced by more than one object, and their velocities will be calculated by the slower, less accurate method. This is because an object influences a pixel if it intersects any of its secondary rays, including shadow-testing rays; therefore, a pixel can be influenced by more than one object even if its direct-viewing ray hits a diffuse object.

The algorithm achieved 2.1 speedup for a single intermediate frame of a simple test scene.

A similar approach to Maisel and Hégron’s treatment of image areas influenced by a single object was presented in Müller’s image interpolation algorithm [23]. However, Müller’s interpolation is time-linear in these image areas, and he does not consider areas affected by more than one object.

5 Radiosity

The radiosity method readily lends itself to “walkthrough” animation sequences, where the objects are static and only the viewpoint moves. Since all the surfaces in the model are assumed to be diffuse reflectors, the surface patch radiosities are viewpoint-independent; once they have been computed, the scene may be rendered from any position by simple projection and hidden surface removal, which can be performed much faster than the initial radiosity calculation.

In the naïve approach to animated radiosity sequence generation, if any of the surfaces in the
model changes its reflectivity or emittance, then the computation-intensive radiosity calculation must be repeated. Worse still, if some of the objects in the model change their positions or geometries, then the form factors (the intervisibility relationships between surface patches) must also be re-calculated.

The following algorithms optimize the rendering of radiosity animation sequences by attempting to focus the re-calculation of the radiosities on those patches which changed the most.

5.1 The Back-Buffer Algorithm

Baum et al.’s back-buffer algorithm [3] uses temporal coherence to accelerate the calculation of form factors. It requires prior knowledge of the volume swept by each moving object. Given such knowledge, it is possible to predict which form factors between static surface patches will remain constant throughout the animation sequence, because no moving object will ever come between them.

At the preprocessing stage, the algorithm creates two run length encoded pixel maps for each static patch \( p \). In the base image, each pixel’s value is the identity of the static patch \( q \) closest to \( p \) in that pixel’s direction, provided that \( q \) is not occluded from \( p \) by any moving object’s swept volume. The back item-buffer is the base image’s complement: it shows which static patch \( q \) is closest to \( p \), provided \( q \) is occluded by some swept volume.

Additionally, a projection mask is prepared at the preprocessing stage for each static patch \( p \) and moving object \( o \). The projection mask is a run length encoded bitmap showing which portions of \( o \)’s swept volume are not occluded from \( p \) by any static surface.

During the rendering process, the back-buffer algorithm finds form factors from each static patch \( p \) to all dynamic patches by first intersecting the projection of each moving object \( o \) on \( p \) with the appropriate projection mask, then examining the remaining dynamic object projections against each other (since one moving object may hide another). The form factors from \( p \) to other static patches are found by combining \( p \)’s base image with those parts of \( p \)’s back item-buffer not occluded by any of the dynamic object portions just found. The form factors from dynamic patches are re-calculated each frame, as in the naïve approach.

This algorithm examines \( 2SD + D^2 \) patch pairs for each frame (where \( S \) is the number of static patches in the model and \( D \) is the number of dynamic patches), and is therefore \( O(FS^2) \) times more efficient than the naïve method for an \( F \)-frame animation sequence. Empirical tests showed 9 to 25 speedup for scenes containing a single moving object. Less impressive speedup may be expected for scenes with more moving objects. Memory requirements are quite high—4 and 9 MB for scenes with 900 and 1200 static patches, respectively.

5.2 Radiosity Redistribution

George et al. [12] presented the radiosity redistribution method for rendering radiosity images of dynamic environments. A similar algorithm, incremental radiosity, was described at the same time by Chen [6]. The main application of these techniques is not animation, but interactive systems where the user can add, delete, move or change objects. It is desirable to respond to such changes in reasonable time, and to show some initial approximation of the updated image even if the model is changed before the calculations from previous changes have converged. This may be accomplished by finding only the changes in the radiosities, rather than re-calculating all the radiosities from scratch.

When a new object is added to the model, the emittance of every luminous patch on the object is first shot out, as in progressive radiosity [8]. Then, the “most significant patch” (heuristically, the
patch which radiates the most energy on the new object’s center) is repeatedly found, its radiosity is shot to the patches on the new object, and the negative of its radiosity is shot to the patches in the new object’s shadow (found using a shadow volume [24]). This continues until the most significant patch’s radiosity is smaller than some threshold level. Finally, the radiosity changes found above are distributed to the rest of the scene by the usual progressive radiosity algorithm (slightly modified to handle negative energies).

Object deletion is handled similarly. A change in an object’s position, geometry or surface properties is handled by deleting the object and re-inserting it with the new attributes.

To handle changes in the model even before the updating of the radiosities due to previous changes has converged, the geometry changes in the scene must be maintained in a list. The list enable the algorithm to determine, for each patch whose radiosity has changed, the scene geometry relative to which the patch’s old radiosity was calculated. This list is called a log in [12] and a geometry queue in [6].

The results of this technique show gradually diminishing inaccuracies in the calculated radiosities, compared to the converged values. George et al. reported 6.1–12.4 speedup, relative to progressive radiosity, for the addition of a new object, while Chen found 1.2–3.6 speedup for geometry changes and 1.1–21.8 speedup for luminance or reflectance changes.

5.3 View Interpolation

As mentioned at the beginning of Section 5, after the patch radiosities have been computed, walkthrough sequences can be rendered by projection and hidden surface removal, which are much less time consuming than the initial radiosity calculation. However, these calculations still require too much time to be performed on-line. Chen and Williams’ view interpolation technique [7] uses morphing to interpolate between different images of a stationary scene viewed from a moving viewpoint.

The morphing technique (short for metamorphosis) simultaneously interpolates shape and texture between two images or video sequences. Image-space morphing, in which the interpolated shape is merely a collection of points or lines in the image space, is widely used for special effects in movies such as Terminator II: Judgment Day and in video clips like Michael Jackson’s Black or White. Object-space morphing, where the shapes are the actual 3D objects in the scene, is more complicated, and is still under research.

The preprocessing stage of the view interpolation algorithm consists of rendering images from some camera positions (e.g., from positions spaced at regular intervals throughout the environment), and calculating the pixel motion vectors between each pair of images rendered from adjacent positions. (The model’s depth information is used to find these vectors.) A quadtree is built over each image, grouping together pixels with similar motion vectors (up to some small threshold).

To render an image from a viewpoint \( V \) which lies in between the positions used at the preprocessing stage, the motion vectors from the pre-rendered positions close to \( V \) are linearly interpolated to \( V \), and the images themselves are transformed using these interpolated vectors. In cases of collision between pixels originating from different pre-rendered images, the Z value of the pixels is compared and the one with the smaller \( Z \) takes precedence.

The interpolation is only accurate if the camera moves parallel to the image plane. Otherwise, the interpolated view is inaccurate, since the image transformation is non-linear. For example, planar surfaces undergo a perspective transformation, rather than a linear one.

The view interpolation algorithm has very high memory demands, because images and pixel motion quadtrees must be maintained for every viewing position considered at the preprocessing
stage. Furthermore, the algorithm is formulated for static scenes; to generalize it to scenes with moving objects, the trajectories of these objects must be known in advance, so images may be pre-rendered for different object locations. This would further increase the memory requirements.

On the positive side, the time needed to interpolate an image is independent of the model complexity. Additionally, the entire model is not needed to compute the pixel motion vectors—only depth information is required. Therefore, the technique might also be used for walkthroughs of real environments photographed from several positions. The necessary depth information may be obtained by a depth-sensing camera or by photogrammetry.

References


11


