3-D ANIMATION TUTORIAL
NIPPON COMPUTER GRAPHICS CONFERENCE
(NICOGRAPH 82)
Tokyo, November 1982

Dr. Alvy Ray Smith
Computer Graphics Project Leader

Lucasfilm Ltd.

ABSTRACT

The organization of a 3-dimensional computer animation system is described. The basic theoretical foundations of 3-D graphics are covered with emphasis on the practical applications. This includes the following topics: antialiasing (sampling and filtering, spatially and temporally), properties of light and materials (colors, reflection, and transparency), and analytic geometry (splines, polygons, quadrics, and patches). The major subsystems of a typical system are described in some detail: modeling (rigid, articulated, and procedural models), animation (of camera and model), subdivision (hierarchies of primitives, fractals), visible-surface decision (culling, hidden-surface removal, shadows), texture generation (for backgrounds and surfaces), rendering (antialiasing, texturing, shading, bump-mapping), and compositing (matting). The importance of simplified logistics and good human interfaces is emphasized.

Introduction

3-D (three-dimensional) animation in computer graphics may be described by subdividing the implied tasks into two main parts, those dealing with modeling and those dealing with rendering. By *modeling* is meant the creation of the 3-D database which serves as the "world" to be portrayed in a synthetic computer graphics sequence. *Rendering* is a frame-by-frame realization of the database. Of course, one database may be rendered in an infinity of ways. Loosely speaking, modeling is the interactive, fast, black-and-white side of the 3-D animation process, and rendering is the batch-mode, slow, color side. The speed distinction may disappear in the future as color computing hardware becomes cheaper and faster.

1. Modeling

One of the most difficult parts of computer animation is getting the object for animation into the computer - i.e., the creation of the database. There are at least two main ways to accomplish this modeling process, input scanning and interactive synthesis from primitives.

1.1. Digitization

Input scanning, or digitization, can mean several things. The most direct 3-D digitization technique for a given object is to simply enter the 3-D coordinates of the object into the computer. Special devices, "3-D digitizers", are just becoming available. Without such a device, direct entry is forbiddingly difficult.

3-D reconstruction from 2-D information is more common. For example, the carefully drawn plans or blueprints of the object may be digitized using the readily available tablets or (2-D) digitizers. Several views, say, two or three orthogonal projections, are entered this way, and

the computer is used to derive 3-D coordinates from the given information. Jim Blinn used this method to enter the Voyager spacecraft model into the Jet Propulsion Laboratory computers for his Jupiter/Saturn flyby simulations.

In another technique, grids are projected on the object of interest which is then photographed from two or more different angles with cameras in a known spatial arrangement. The photographs are hand-digitized as before at the grid points. Enough information is provided the computer to obtain 3-D coordinates from triangulation. This method was used by Information International Inc. to enter actor Peter Fonda's head into their computers for the movie *Futureworld.*

There are several other variations on these triangulation methods, but the main thing to notice is that all the digitization processes are quite tedious. There are also the problems of organizing the mass of data so entered into a usable database, removing spurious points, and smoothing out the noise typical of this kind of point measurement.

1.2. Synthesis from Primitives

Very accurate models may be created directly into a machine using interactive programs for synthesizing objects from primitive shapes. This is called "solids modeling" in the context of CAD/CAM, but in 3-D animation for movies much of the "machinery" typically included in CAD/CAM systems may be omitted. This includes dimensioning and tolerancing and insistence on realizable objects (by machining a piece of metal, for example).

1.2.1. Primitives

Some common primitives are the quadrics - sphere, cone, cylinder, ellipsoid, paraboloid, hyperboloid - and their superquadric generalizations, the torus (sometimes included as an honorary quadric), patches - bilinear, biquadratic, but particularly, bicubic and the rational versions of these - and polygons. Other primitives are point spots, lines, and surfaces of revolution (with silhouettes defined by polygons or splines). Many of these might eventually be subdivided into and realized as, say, polygons, but in the modeling stage they are manipulated in their higher-order, intuitive form.

1.2.2. Procedural Models

Some other basic building blocks of models which are too sophisticated to be called primitives are fractals and procedural models. A fractal model is really a primitive model to which a controlled randomizing process has been applied to create a more detailed model. So "fractals" should probably be listed in the Model Operations section below. A large class of models are not describable as combinations of primitives or as fractally deformed combinations of primitives. This general class of program-defined or procedural models, includes the fires, by Bill Reeves of Lucasfilm, used in the movie *Star Trek II - The Wrath of Khan* (which also includes a good example of the fractal technique employed by Loren Carpenter of Lucasfilm).

1.2.3. Model Operations

The synthesis technique of database creation requires that an artist or designer create objects from the primitives or procedural models available to him. This implies there is some interactive station, or at least a keyboard-driven language, available to him which gives him copies, or "instances", of the primitives and provides him with a set of operations for positioning and combining these instances.

Some typical operators include the "boolean" operators of intersection, union, and set difference. These are particularly popular in CAD/CAM contexts. They are difficult to realize in the case of higher-order surfaces such as bicubic or rational bicubic patches because of the difficulty in deriving analytic expressions for intersection curves. Numerical, or iterative, approaches would be used instead.

Another set of operators is what I shall call "articulation" operators. Their purpose is to force a hierarchy on a set of primitives. For example, at the New York Institute of Technology, there are programs for arranging ellipsoids and cylinders into tree structures representing (but not necessarily) humans or animals. The nodes of the tree are the joints of the corresponding animal. This is the reason for calling these articulation operators. Included are operators which allow a user to manipulate or traverse the tree structures created with other operators.

Underlying any interactive 3-D viewing situation are the rotation, scaling, translation, skewing, and perspective transformations which are the given language of 3-D computer graphics. It is well-known that one 4x4 matrix will accomplish all these transformations simultaneously or sequentially. Some hardware commonly available, notably the Evans & Sutherland Picture System, has a 4x4 matrix multiplier built into its hardware since it is such a common operation in graphics. Very briefly, if a model is thought of as a set of 3-D points, then these points are turned into 4-D points by appending a unity 4-th coordinate, the so-called *homogeneous* coordinate. Then a 3-D transformation is accomplished by multiplying a matrix of form

$$\begin{matrix} s & S & S & p \\ S & s & S & p \\ S & S & s & p \\ t & t & t & g \end{matrix}$$

times each 4-D point in the model. (s stands for scale, S for skew, t for translate, p for perspective, and g for global scale. Rotates are a combination of s and S entries.) Then the 4-D points are converted back into 3-D points by dividing the first three coordinates by the homogeneous coordinate. The articulated tree models frequently have a 4x4 transformation matrix associated with each node to specify relative placement and size.

1.3. Animation

The discussion of graphical database creation has so far centered on the generation of still frames. One of the powers of, and principal motivations for, computer graphics is time modeling or "animation". Strictly speaking, the term "animation" means "giving life to" and is the domain of artists we call animators. In the context of computer graphics, it means only "giving motion to", and this is how we shall use the term here. The problem is conveniently broken into the two subproblems of animating the model and animating the camera which views the model.

1.3.1. Model Animation

The usual notion of animation in 3-D computer graphics is called *keyframe animation*. Only selected, important frames are modeled using some flavor of static modeler and articulation operators, and the computer is used to generate the "inbetween" frames. For example, in complex animation perhaps every third or fourth frame is provided as a keyframe to the inbetweening program, which then fills in the missing two or three frames between keyframes. Surprisingly, computer-aided animation is easier in 3-D than it is in 2-D. This is because the 3-D model contains depth information which is missing from a 2-D model and which must be deduced by the animation program. This is, in general, a difficult if not impossible problem requiring human intelligence. No one has yet written an artificially intelligent 2-D computer inbetweener.

An adequately complete animation program - assuming keyframe creation is accomplished elsewhere - includes the following facilities: The ability to specify path of movement, placement of frames in time along this path, and real-time playback of generated frames. Also typically needed are means for changing frame numbers attached to frames, for changing the number of inbetween frames, for copying motion used in one set of frames to another set of frames, and for creation of new keyframes, called *breakdowns*, from old inbetween frames when it becomes clear that the given set does not give fine enough control. Note that this implies the ability to pass data bidirectionally between a keyframe modeler program and the animation program (assuming they are different modules).

Since models typically become quite complex, even in line-drawing form, there must be some way to represent primitives in a simplified fashion while exercising the animator program. For example, if spheres are represented by lines of longitude and latitude, then the number of these lines might be reduced, or only dots might be used, or even just an enclosing box.

Smooth motion paths are best specified by passing splines (piecewise cubic polynomials here) through keyframe parameters. The two most useful splines currently known are the *B-spline* and the *Catmull-Rom spline**. Both of these are cubic splines with local control. That is, a change in a keyframe parameter being splined affects the curve only at the frames between the given keyframe and the two keyframes before and after the given keyframe. A change affects only four keyframes of regardless of how many there are altogether. The B-spline is an *approximating* spline and the Catmull-Rom spline is an *interpolating* spline. An approximating spline passes near the parameter values at the keyframes (the "knot" values of the spline) but not necessarily through them, but an interpolating spline is guaranteed to pass through them. The approximating spline tends to be more graceful than the interpolating spline because it preserves second-order continuity. The interpolating spline may have undesired kinks in it.

These two splines are easily specified. Given a list of x-coordinates, and a parameter u which will take us along the spline connecting (or approximately connecting) one coordinate x_0 to the next x_1 as the parameter is varied from 0 to 1, a new x-coordinate is obtained from each value of u from the four nearest given x-coordinates (two behind, two ahead, along the curve) by $U*M*X^T$, where

$$U = [\, u^3 \; u^2 \; u \; 1 \,],$$

$$X = [\, x_{-1} \; x_0 \; x_1 \; x_2 \,],$$

and M is the "magic matrix"

$$\frac{1}{2} * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

for Catmull-Rom splines or

$$\frac{1}{6} * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

for B-splines. The y-coordinates and z-coordinates of a given set of points to be interpolated would be treated similarly, as would any other parameter to be smoothly interpolated through the animation - e.g., the angle of rotation about the z-axis.

1.3.2. Camera Animation

Animating the camera's view of the objects or objects making up a synthetic scene is much like the model animation discussed above. In fact, it can be combined with it, but I have broken it out as a separate part of the animation process because it is usually thought of as an independent process. The terminology tends to be different, for example. Whereas one tends to use commands such as "rotate 30 degrees about x (y)" for animating a model, the related command to the camera might be "rotate 30 degrees in elevation (azimuth)".

The so-called "viewing parameters" are placed under control for camera animation but are typically set to some standard setting for model animation where the model changes but the camera doesn't budge. These parameters are related to the perspective transformtion applied to the

*The cubic Catmull-Rom spline is also called an Overhauser spline because of independent discovery. Both are piecewise polynomial (i.e., spline) generalizations of the well-known Lagrange interpolants.

synthetic scene: field-of-view (which can cause "fish-eye" lens distortions, if desired), near and far clipping planes, aspect ratio (e.g., Panavision, Vistavision, or video frame shape).

It is convenient to provide users with several "cameras" so that a scene can be viewed from several different vantage points. These different cameras may also be thought of as keyframes for the camera animation. Another convenient tool is an *oracle camera* which views the scene with the other cameras added to it. They might be represented, for example, by their viewing frustrums.

The animation of the path of the camera and any other parameters describing it is done using B-splines or Catmull-Rom splines just as for models.

1.4. Modeling Interfaces

In all cases, the interface to the modeler (static, articulated, or animated) is of utmost importance. There are very few successful interfaces, where success is measured by grace and power. Grace in turn is measured by economy of motion, intuitiveness of control mechanisms, and the "feel" of the actual, physical controls. For example, it is difficult to write an convenient interface which allows a user to "attach this sphere at this point to that cone at that point". It is easy to write a program which assumes only one coordinate system - that attached to, and moving with, the object being constructed - but it is difficult to write one that references the fixed screen coordinate system, or allows the user to go back and forth between both systems as ease dictates.

A class of modelers which are particularly difficult to interface are the patch design systems. Patches - and, particularly, rational bicubic patches - have many degrees of freedom. They obtain their power from this freedom. The most promising technique for interaction with these modeling primitives (or any primitives, actually) is via stereo pairs of 3-D line drawings. No such system is currently in use. The "head-mounted display" of the University of Utah was the most serious attack which has been made on this problem.

2. Rendering

After a set of models has been digitized or synthesized, combined into scenes, and animated, we presumably have a set of frames in some conventional 3-D database format which then has to be rendered frame-by-frame into final 2-D form in full color with all hidden surfaces removed.

One of the most powerful tricks used in computer graphics is that of *texture mapping*. This is the wrapping of a 2-D picture - painted, scanned in, or algorithmicly generated - onto a 3-D surface to give it much more detail than the modeling process feasibly admits. I include in the Rendering section of this tutorial the generation of the texture maps used for this process since this is basically a color surface problem and not a black-and-white 3-D object definition problem. This leads naturally to a subdivision of the Rendering domain into the intrinsically 2-D and intrinsically 3-D coloring problems.

2.1. 2-Dimensional Problems

2.1.1. Texture Generation

The richest source of readily available imagery for simulated scenes is the real world. Since it is generally difficult to model the real world with anything like its complexity, synthetic models are enriched by texture mapping 2-D digitized real-world scenes onto their surfaces. 2-D input scanning is simpler than 3-D - it requires only one camera, no triangulation, and cheaper equipment - but it suffers from some of the same problems. Noise is the principal problem. Some of this can be removed by a flat-field correction accomplished by scanning in a white sheet of paper, or, in the case of transparencies, scanning in an empty aperture and multiplying it times the desired image pixel-by-pixel. Various filters for edge sharpening and contrast enhancement may also be used.

Another source of textures are the so-called "paint" programs which allow an artist to paint into a computer memory in such a way as to feel as if he were painting directly onto a color video monitor. These programs may be quite elaborate. In fact, a full paint program is really a system of programs. Such a system typically includes basic (simulation of) painting, filling of areas of arbitrary shape, line and curve drawing aides, picture saving and restoring, magnification, cursoring, color palette setup, and brush definition programs. Some paint stations include 2-D input scanning also. There are now several commercially available paint stations: "Superpaint" from Aurora Systems Inc. in San Francisco, "Images" from the Computer Graphics Lab Inc. in New York, and "Paintbox" by Quantel in Great Britain, to name a few. The most sophisticated such program has recently been written by Tom Porter at Lucasfilm.

The third technique of texture definition is algorithmic. This is, of course, as general as can be imagined in the context of computer programming. For example, a program which generates z as a function of x and y and represents z with shades of a color, or different colors, is a powerful source of often intriguing patterns for texture-mapping.

It is usual that a texture is stored in a computer as a set of samples taken at equally spaced values of x and y. Textures may go through quite large scale changes in the mapping process onto 3-D surfaces. Theoretically it is possible to reconstruct a set of samples, make a change of scale on the reconstruction, and resample with no loss of information. Practically, however, this is done only approximately so as not to take too long computationally. One speedup technique is to prescale a texture which may go through many scale changes in an animated sequence. The prescaling is typically done to powers of two. That is, the texture is prescaled by factors of 2, 4, 8, ..., and 1/2, 1/4, 1/8, ... Then when a scale change is required the texture already nearest in size is used for the slow, final mapping. This technique was first described by Ed Catmull at the University of Utah many years ago and has been refined into the *mipmap* technique by Lance Williams of New York Institute Technology. It was also used by Tom Duff at Lucasfilm in the "Genesis Demo" sequence in *Star Trek II.*

The mechanics of assigning textures to surfaces is a large problem in itself which has nowhere yet been adequately solved. The Earth-like planet in the Genesis Demo sequence was created by painting a texture with the Lucasfilm paint program (Chris Evans did the painting). Then Tom Duff mapped this texture onto a sphere for the finished planet. Spheres are one kind of surface for which the texture assignment problem has been solved.

2.1.2. Matting

Another 2-D color rendering problem is that of compositing several 2-D pictures into a single final frame. Since interesting scenes tend to be complex, they are typically broken into several *elements.* For example, in the Genesis Demo sequence every frame is composed from two to five elements. A frame showing a projectile impacting a planet surface has a starfield element, a planet element, an explosion element, and a shockwave element. Each of these is the final result of a 3-D rendering problem, but none is a complete frame. A final 2-D *matting,* or compositing, step is applied to combine the elements. This is straightforward because a soft-edged matte is generated with each element as part of the 3-D rendering process.

A *matte* is simply a picture which is 0 wherever another picture of the same size is to be transparent, 1 where it is to be opaque, and a fraction between 0 and 1 for corresponding partial transparency. Let a be a color from picture A and b be a color from picture B. A is to be placed over B with a registered over b. Let α be the partial transparency of pixel a - i.e., α is the pixel in the matte for A which is in registration with pixel a. The operator for combining the two pictures according to the given matte is used so often in computer graphics that it has a special name, *lerp,* short for "linear interpolation". Lerp is defined to be the pixel-by-pixel operation

$$\alpha*A + (1-\alpha)*B = B + \alpha*(B-A).$$

That is, it is the linear interpolation of B to A by amount α known to be between 0 (transparent) and 1 (opaque).

Matting is one way of partitioning a scene so that hidden surface removal becomes simple. The more general and difficult problem of hidden surfaces in 3-D is covered in the next section.

2.2. 3-Dimensional Problems

The amount of space I have alloted per section in this tutorial is roughly a measure of the importance of the contents of a section in the overall process of 3-D computer animation. I say this because the 3-D problems discussed in this section tend to be the focus of computer graphics. This is because they are the most difficult problems, requiring the most sophisticated algorithms and using the largest number of computer cycles, of all those in a full 3-D animation system. Nevertheless, they are just part of the final picture.

2.2.1. Hidden Surface Removal

In general, the removal of hidden surfaces from a synthetic scene cannot be solved with simple tricks such as 2-D matting of elements discussed above. Consider a complex scene which has been animated and is ready for rendering into a frame. Assume this scene is of complexity approaching that of the real world and that it is to be rendered at film resolution. No one has yet decided what the optimal film resolution for digital pictures is, but it is safe to say that there will be millions of pixels involved (with the range being anywhere from 1 to 64 million). Further assume that all primitives used in the modeling stage are subdivided into polygons for this rendering stage. This is not necessarily the case but it is done often enough to make our example reasonable. So each leaf, car, face, blade of grass, rock, river, etc. is subdivided sufficiently so as to show no sampling effects. By looking around in the real world, it is apparent that depth complexity, the number of surfaces crossed by a straight line from the eye to infinity, is something between 10 and 100 typically. So we are talking about potentially many millions of polygons of arbitrary shape and orientation stacked about, say 20, deep at any one pixel. The analysis of which surfaces hide which in this scenario is to be solved for every frame! A movie has 100,000 to 150,000 frames so it is easy to see just how computationally intensive fully realized 3-D computer animation may be and why so much work has been, and still is, put into finding algorithms for solving the hidden surface problem.

I will not attempt to analyse the problem fully here but just indicate some of the major approaches. There are many tricks for reducing the complexity of a scene before the brute-force hidden-surface algorithms are put to work. One is the matting decomposition already discussed. This is just a special case of what is called *clustering,* or breaking a scene into modules which can be treated completely independently of one another.

A process of *culling* may quickly reduce the "environment" of objects which have to be considered in a frame. This is simply the removal from the working database of all objects which are clearly offscreen or behind the camera.

The use of *coherence* can greatly reduce the amount of computation. Coherence is a measure of how much a frame is like the one just solved, or how much a scanline is like the one just processed, etc. In other words, many algorithms are sped up by noticing when a computation is to be performed again and saving the current result for that later time. There are many different types of coherence which may be exploited.

In the brief summary of the major hidden-surface solution techniques below I will use the terms *object space* and *image space* frequently. Object space is the coordinate system used for modeling the scene, or perhaps a translation-rotation of that system into a more convenient one such as a coordinate-system based at the camera. It is in any case, a 3-D space with no perspective for describing the modeled scene with real numbers and continuous curves (at least to within the floating-point accuracy of a computer). Image space is the coordinate system of the plane in which the final image of the model is projected. It implies perspective projection and representation of the projected model by numbers at or near the resolution of the display (integer representation may even be used in a computer). Although the image is 2-D, a third dimension obtained from the perspective projection is kept for depth comparisons.

2.2.1.1. Ray Tracing

Some of the most beautiful computer graphics pictures so far obtained have used *ray tracing* algorithms for hidden-surface detection and removal. The idea is to follow all the rays of light emitted from light sources in a modeled scene through all their reflections and refractions by objects in the scene until they strike the image plane and then the eye or camera. Thus the laws of optics are applied faithfully. For computational purposes, these rays are traced in the reverse direction, from the eye to the emitting source. Since a ray can bounce off any object in the modeled scene, the scene cannot be culled. For example, a mirrored sphere in the camera's view may reflect what is behind the camera - and the camera. This type of algorithm is performed in object space, tends to be slow, and makes neighborhood integration for antialiasing filters difficult (see section on Antialiasing below), but it provides shadows and handles transparency well. It can be used for any surface primitive for which intersection with a ray can be solved. Essentially, a new hidden-surface problem has to be solved every time a ray bounces off an object, so the problem is recursively difficult.

2.2.1.2. Depth Buffer

A very simple hidden-surface solution uses a large piece of memory called a *Z buffer*, or *depth buffer*. A Z buffer has one location for each pixel in the final image, so this type of algorithm is intrinsically an image space algorithm. A pixel portion of a surface in a scene is written into the final image only if its z-coordinate is less than that stored in the Z buffer at that point. (It is fairly common practice in computer graphics to use an image space coordinate system with x horizontal, y vertical, and z increasing into the screen away from the viewer.) If the new pixel hides the old, then its z value replaces that of the old pixel in the Z buffer.

The benefits of this approach are: No ordering of the environment is required - i.e., no sorting is required. Several different programs can be used to generate objects independently which can then be resolved in depth with one Z-buffer pass. The method is not based on any one kind of primitive. The program is very simple to write.

The approach fails to properly render transparency, and it does not support antialiasing nor shadowing. However, Lance Williams of NYIT has a method of approximately antialiasing a Z-buffered picture and has shown how two uses of the Z-buffer - one for the viewer and one for the light source - can be used for shadows.

2.2.1.3. Sorting and Clipping

The majority of algorithms which have been devised for hidden-surface resolution are polygon based. There have been perhaps twenty to thirty such algorithms devised so far which differ by the order of the sort performed, the type of coherence exploited, and whether image space or object space is used.

A typical example of this type algorithm assumes a culled database which has been sorted in order of the maximum y for each polygon. Now the data space is subdivided for further sorting. For example, it may be divided into scanlines - that is, into horizontal slices which are to be resolved into one horizontal scanline of a display device or medium. In such *scanline oriented* algorithms the polygons which intersect the given scanline, defined by a plane passing through the scanline and the viewpoint, are further sorted on their minimum x coordinates. Then a variety of depth sorting techniques are used to make the final resolution. Since depth information may change only slightly from scanline to scanline, information on one line may be used to simplify computations on the next. This is an example of the use of coherence.

Another way to subdivide the space for further sorting is by dividing the image plane into a grid of squares and solving within the squares. The database of polygons is *clipped* against the current square of interest to subdivide polygons into subpolygons with new edges coinciding with the current square's edge. A square which is particularly difficult to solve may be further subdivided into smaller squares before final resolution is attempted. Some algorithms use polygons in the scene to clip against rather than an arbitrary grid of squares.

Some of these algorithms handle shadows, others do not. Some handle transparency, others don't. Most can be made to deal appropriately with antialiasing. They are sort intensive and tend to bog down for large numbers of polygons (millions). Some are plagued by a rapid increase in the number of subpolygons generated at intermediate steps, and others by the extreme shapes these subpolygons may take (slivers). In all cases, they are not designed to solve the optics problems solved by the ray-tracing algorithms. They essentially handle only one bounce of a ray.

2.2.2. Coloring

After the hidden-surface problem has been resolved, there still remains the actually rendering of color at each pixel in the visible surfaces, which may be only transparently visible. We discuss here the many factors that may be used for determining the color at a pixel.

2.2.2.1. Lighting Models

An environment may include several light sources each of which affects the color of an object, either directly or by shadowing or by filtering through partially transparent objects or partially reflective objects. A light source may be local and distributed over space or it may reside at a point at infinity. It may be in the scene viewed or off-camera. It may be colored. Simple computer graphics tends to use one white light source located at infinity and out of camera range.

A typical model for illumination by a light source will have a specular component, a diffuse component, and an ambient component. The specular component is that seen when the viewpoint is at or near the point where the angle of reflectance of a light ray from a surface is equal the angle of incidence. It dominates on shiny surfaces. The diffuse component is more uniformly distributed over an illuminated surface and dominates on matte surfaces. The ambient component is a constant term added in to raise black to a low gray. It captures the notion of a low background glow due to many reflections of many low-level light sources. All the lighting models substitute approximating terms for what would require very complex integrations for truly accurate modeling.

One common approximation which should, however, be avoided for good color renditions is that which gives the specular, diffuse, and ambient components the color of the light source. Real objects modify these three components in three different ways. For example, metals have specular highlights colored by the metal not the light source (and almost no diffuse component). Striking improvements in computer graphics scenes are obtained by the simple measure of using better lighting models.

2.2.2.2. Texture Mapping

As has already been mentioned, a great amount of complexity may be apparently added to a synthetic scene by mapping a 2-D picture, or "texture", onto a 3-D surface. The logistics of doing this mapping may be one of the most difficult parts of a 3-D animation exercise. Textures tend to require large amounts of disk space. Since disks are as yet relatively slow devices, it is important to have texturing algorithms which minimize disk accesses.

2.2.2.3. Bump Mapping

A simple and very powerful way of adding further complexity to a scene is based on the same principle as is texture mapping. This principle is that if the amplitude of the detail is small relative the amplitude of the object with the detail, then a satisfactory approximation is obtained by reducing the detail amplitude to zero - i.e., by using a 2-D representation of the detail. Most lighting models vary the shade of an object as a function of the normal to the surface of the model. *Bump mapping,* or *normal perturbation,* simulates wrinkles and bumps of low amplitude in a surface by appropriately wiggling the surface normal in the vicinity of the bumps and wrinkles as if they really existed in the model. Only at the silhouette of the object is the secret given away - there are no bumps or wrinkles actually visible - but with enough complexity viewers tend to ignore this fact.

2.2.2.4. Environment Mapping

An alternative to ray-tracing is what is sometimes called *environment mapping*. If an object does not move with respect to its environment, then a texture map may be computed which takes optics fully into account. When this texture is mapped onto the surface at rendering time, the effect is the same as if ray-tracing had been performed. In fact, it has, but only once, not for every frame.

2.2.3. Antialiasing

I have mentioned antialiasing several times in this tutorial. It is one of the most important topics in computer graphics and one still not fully understood by the community of graphics users. Unfortunately, it is much simpler to ignore antialiasing when writing a rendering program than to think it out and include it. As we have seen, some algorithms do not even allow incorporation of antialiasing. These will fall away as the importance of antialiasing becomes widely known.

Aliasing is most often seen as "stairsteps" or "jaggies" in computer-generated frames. In motion, these jagged edges run along the edge destroying the illusion that it is the edge of a hard object. Other manifestations of aliasing are polygons that flash on and off, moire patterns, and "strobing" or temporal aliasing.

2.2.3.1. Spatial

Aliasing is a sampling problem. Computer graphics typically uses samples of real surfaces rather than continuous surfaces. This is because the surface is eventually displayed in a pixelated form which is equivalent to saying a pixel is a sample of a continuous surface. Sampling theory says that samples are good enough for representation of a continuous surface if there are no direction changes in the surface within the distance between pixels or samples. A stricter way of saying this is that the sampled surface should have no frequencies, in the Fourier sense, of higher spatial frequency than the sampling grid. Unfortunately, polygons, which are very frequently used in computer graphics, have very high frequencies near their edges. Some kind of local filtering or averaging must be performed to rid the surface of these high frequencies else they will appear as ragged edges in a sampled display.

2.2.3.2. Temporal

A movie is a set of samples taken at 24 or 30 times per second of a 3-D event, where one of the dimensions is time. Just as in the spatial case, aliasing will occur if motion changes occur with higher frequency than 24 or 30 times per second. The solution for time is called *motion blur*. Unfortunately, motion blur is at this time usually omitted from computer graphics, it being a difficult problem to solve. However, the stars and fires in the Lucasfilm *Star Trek II* sequence were motion blurred to prevent strobing in time.

3. Managing

A subject which I will not treat is that of primary importance, the original creativity, the creation of the story, characters, look, colors, composition, etc. A subject which will receive less coverage here than probably it should, taking back seat to the more glamorous aspects of computer animation, modeling and rendering, is that of management and logistics. It becomes very apparent, however, in any real computer animation that the management of resources - disks, magtapes, cpus, people, film units, painting stations, etc. - may create a severe bottleneck unless properly thought out. In fact, one of the principal uses of the computer in computer animation is the management of these resources.

An *exposure sheet* is commonly used to specify what elements and what backgrounds go into an animated sequence, frame-by-frame. The exposure sheet contains information about compositing and camera or videotape control (advance, skip, reverse, etc.) Facilities for exposure sheet generation and maintainance are mandatory and can become surprisingly complex. For example,

mistakes always happen. It is important that any exposure sheet facility allow easy recovery in case of error. With complex relationships between elements and where they are stored, this can be very difficult and lead to further errors in the recording.

References

Rather than directly cite the many articles from which the information above is distilled, I have opted to list a body of books and conference proceedings which include most of what may be found in the literature, or at least give pointers into the literature.

[1] SIGGRAPH Proceedings, 1977-82. [The Special Interest Group on Computer Graphics, of the ACM (Association for Computing Machinery), holds an annual convention, usually in July or August, and issues a proceedings of the papers presented.]

[2] Barnhill, Robert E., and Riesenfeld, Richard F. *Computer Aided Geometric Design,* Academic Press, Inc., San Francisco, 1974.

[3] Beatty, John C., and Booth, Kellogg S. *Tutorial: Computer Graphics,* IEEE Computer Society, New York, 1982. (Second Edition).

[4] Faux, I.D., and Pratt, M.J. *Computational Geometry for Design and Manufacture,* John Wiley & Sons, New York, 1979.

[5] Freeman, Herbert. *Tutorial and Selected Readings in Interactive Computer Graphics,* IEEE Computer Society, New York, 1980.

[6] Foley, J.D., and Van Dam, A. *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Menlo Park, California, 1982.

[7] Newman, William M., and Sproull, Robert F. *Principles of Interactive Computer Graphics,* McGraw-Hill, San Francisco, 1979. (Second Edition).

[8] Rogers, David F., and Adams, J. Alan. *Mathematical Elements for Computer Graphics,* McGraw-Hill, San Francisco, 1976.