6

Three-dimensional Scene Representations: Modeling, Animation, and Rendering Techniques

Uğur Güdükbay and Funda Durupınar

Department of Computer Eng., Bilkent University, 06800, Bilkent, Ankara, Turkey

Modeling the behavior and appearance of captured three-dimensional (3D) objects is a fundamental requirement for scene representation in a three-dimensional television (3DTV) framework. By using the data acquired from multiple cameras, it is possible to model a scene with high quality visual results. In fact, 3D scene capturing and representation phases are highly correlated. Information acquired from the capturing phase can be employed in the representation phase by using computer graphics and image processing techniques. The resultant model then allows the users to interact with the scene, not just remain observers but be participants themselves. Thus, the main considerations for the quality of a scene representation technique are basically the accuracy of the technique about how the results correspond to the original scene and the efficiency of the technique as real-time performance is required.

3D shape modeling is an essential component of scene representation for 3DTV. Time-varying mesh representations provide a suitable way of representing 3D shapes. With these methods, the static components of a scene are constructed only once and the other objects are modeled as dynamic components, thus the computational time to represent 3D scenes is reduced. Polygonal meshes are efficiently used in shape modeling due to their built-in representation in hardware. Thus, they are suitable for applications such as 3DTV where real-time performance is required. Alternatively, volumetric representations can be used in shape modeling. The basic volume elements, voxels, of a 3D space correspond to the 2D pixels of an image. Volumetric techniques require large amounts of data in order to represent a scene or object accurately. Images acquired from multiple calibrated cameras provide the necessary information for volumetric models. Thus, these methods are intuitive for 3DTV. However, recent research shows that point-based approaches are the most suitable shape modeling techniques for 3DTV. The reason is that results of 3D data acquisition methods such as laser scans already represent the scene in a point-based manner.

3D scene representation has two components: *geometry* and *texture*. Geometry representation is handled by modeling the shape of an object or a scene. Since the scenes mostly contain dynamic objects that move and deform in different ways, modeling the motion becomes important. Animation techniques that have potential for real-time hardware implementations are promising approaches to be used in a 3DTV framework. Texture representation is handled by the underlying rendering technique. Scan-line rendering techniques are suitable for 3DTV as they are hardware-supported and efficient. In addition, image-based rendering is a very successful and promising rendering scheme for 3DTV as it directly makes use of the captured images.

This chapter provides introductory knowledge for the modeling, animation, and rendering techniques used in computer graphics. It is not an exhaustive survey of these topics and includes only representatives of each, focusing on techniques relevant to 3DTV. The interested reader is referred to the references for an in-depth discussion of the topics covered.

The chapter is organized as follows. First, different 3D scene representation techniques, namely mesh-based representations, volumetric methods, and point-based techniques, will be discussed. Then, we will explain animation techniques for modeling object behavior. Finally, we will discuss illumination models and rendering techniques for 3D scenes containing different types of objects and lighting conditions.

6.1 Modeling

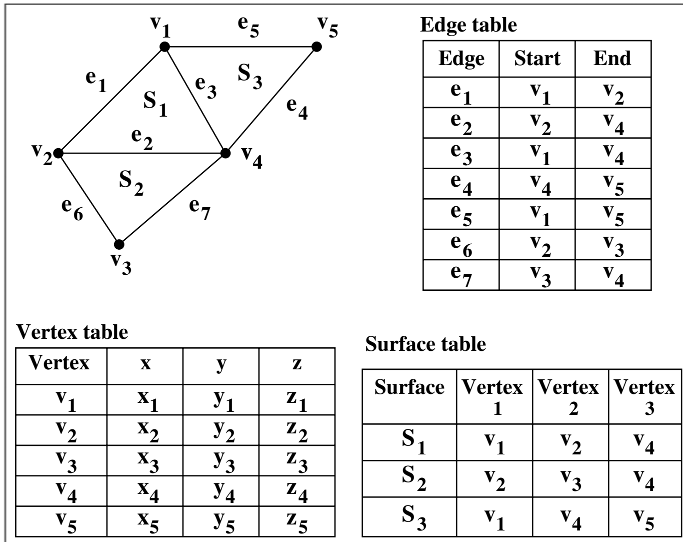
There are two main approaches to represent the shape of arbitrary free-form objects. The first approach, which is called *Constructive Solid Geometry*, models the shapes of free-form objects as a composition of geometrically and algebraically defined primitives, such as polygons, implicit surfaces, or parametric surfaces. This approach uses Boolean operations to combine regular shapes and is widely used as a Computer-Aided Design tool. The second approach deforms regular shapes using deformation techniques, such as regular deformations [1] and Free-Form Deformations [2] to obtain irregular, free-form objects.

Before going into the details of different shape representation techniques based on Euclidean geometry, we will say a few words about modeling the shapes of natural objects. Natural objects, such as mountains, clouds, and trees, cannot be described using equations since these objects do not have regular shapes; their irregular or fragmented features cannot be realistically modeled using the methods based on Euclidean geometry [3]. *Fractal-geometry* methods use procedures to model such objects [4]. L-systems (Lindenmayer systems) provide a mathematical formalism for realistic modeling of plants and plant generation. The basic idea is to define complex objects, like plants,

by successively replacing parts of simple initial objects using a set of rewriting rules. The rewriting rules are applied in a parallel fashion for different parts of the objects [5].

6.1.1 Polygonal Mesh Representations

The surface of a 3D object can be approximated using a number of planar polygons. A polygonal approximation to a 3D object has faces, edges, vertices and normal vectors to identify the spatial orientation of the polygon surfaces. These are stored in geometric data tables. A vertex table stores the x, y, and z-coordinates of the vertices. Surfaces, or polygons, are stored in surface tables, which contain pointers to the vertex tables for each vertex comprising that polygonal surface. Edge tables are useful for wireframe drawing purposes and they also represent edges using pointers to the vertex tables [3]. Mostly, triangles are used for polygonal approximations of objects since triangles can be processed in hardware using graphics cards in today's computers. Figure 6.1 shows a simple object and its corresponding vertex, edge and surface tables. In addition, there are also some attributes associated with vertices and faces such as the degree of transparency, surface reflectivity, and texture characteristics, which are stored in attribute data tables. These are necessary for shading polygonal surfaces. The normal vector of a polygonal surface is calculated by taking the cross product of two non-colinear vectors lying on the polygonal surface. The vertex normals are calculated by taking the average of the face normals sharing a vertex.



Edge table

Edge	Start	End
e_1	v_1	v_2
e_2	v_2	v_4
e_3	v_1	v_4
e_4	v_4	v_5
e_5	v_1	v_5
e_6	v_2	v_3
e_7	v_3	v_4

Vertex table

Vertex	x	y	z
v_1	x_1	y_1	z_1
v_2	x_2	y_2	z_2
v_3	x_3	y_3	z_3
v_4	x_4	y_4	z_4
v_5	x_5	y_5	z_5

Surface table

Surface	Vertex 1	Vertex 2	Vertex 3
S_1	v_1	v_2	v_4
S_2	v_2	v_3	v_4
S_3	v_1	v_4	v_5

Fig. 6.1. A polygonal object and its vertex, edge and surface tables

When the polygonal approximations of objects are very large, containing millions of polygons, level-of-detail approximations of the models become inevitable. Polygonal model simplification is the main tool to obtain different levels of detail of polygonal models. Progressive mesh representations that store different level-of-detail approximations of large models are used to visualize complex models using view-dependent visualization techniques. These techniques are used to display the models by using the suitable level of detail according to the current viewpoint so that the polygons that do not contribute to the final image are not processed by the graphics pipeline [6, 7, 8]. Figure 6.2 shows a sphere rendered with two different levels of detail.

6.1.2 Parametric Surfaces

A parametric surface is defined as a mapping from 2-space to 3-space since each parametric surface can be defined using two parameters. Parametric surfaces are represented with the following equation:

$$\underline{X}(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}, \quad \begin{matrix} u_0 \geq u \leq u_1 \\ v_0 \geq v \leq v_1 \end{matrix} \tag{6.1}$$

Normal vectors for parametric surfaces can be calculated by taking the cross product of the surface tangent functions. Surface tangent functions can be found by taking the partial derivatives of the parametric surface function with respect to the surface parameters. As an example, the derivation of the parametric normal vector equation for the unit sphere is given in the following equations.

$$\underline{X}(u,v) = \begin{bmatrix} cos(u)\ cos(v) \\ cos(u)\ sin(v) \\ sin(u) \end{bmatrix}, \begin{matrix} -\frac{\pi}{2} \leq u \leq \frac{\pi}{2} \\ -\pi \leq v < \pi \end{matrix} \tag{6.2}$$

$$\underline{N}(u,v) = \frac{\partial \underline{X}}{\partial u} \times \frac{\partial \underline{X}}{\partial v} \tag{6.3}$$

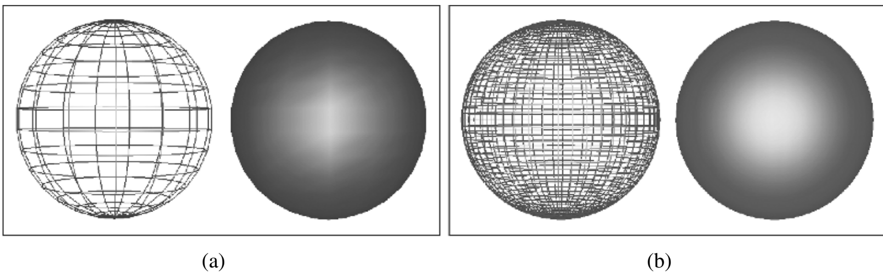


Fig. 6.2. Level-of-detail example on wireframe and smooth-shaded spheres. (**a**) low-resolution; (**b**) high resolution

$$\underline{N}(u,v) = \begin{bmatrix} -sin(u)\ cos(v) \\ -sin(u)\ sin(v) \\ cos(u) \end{bmatrix} \times \begin{bmatrix} -cos(u)\ sin(v) \\ cos(u)\ cos(v) \\ 0 \end{bmatrix} \tag{6.4}$$

$$\underline{N}(u,v) = \begin{bmatrix} cos^2(u)\ cos^2(v) \\ cos^2(u)\ sin^2(v) \\ sin^2(u) \end{bmatrix} \tag{6.5}$$

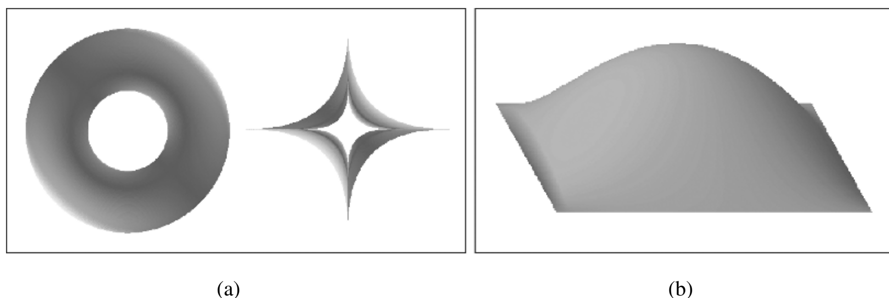
Each coordinate of a point on a parametric surface can be calculated independently from other coordinates; this makes the parametric surfaces attractive for generating polygonal approximations for object surfaces. This is generally done by sampling a regular grid on the parameter space and then calculating the points on the parametric surface by plugging the parameter values at the grid locations into the parametric surface functions for each coordinate. The coordinates of the points on the parametric surface are stored in a two-dimensional array that corresponds to the grid for parameter values. Then, the polygons (triangles) are implicitly obtained by forming triangles on the grid. Such kinds of polygonal approximations are called *regular meshes* since the polygons are formed using neighboring grid points in a regular way and the polygon information is not stored explicitly. The problem with parametric surfaces is that we only know the parametric surface functions for a limited set of regular objects.

Examples of parametric surfaces that can be used for representing primitive objects are quadrics, superquadrics [9], and bi-cubic surfaces, such as B-spline, Hermite, Bézier, etc. [10, 11]. Figure 6.3 shows examples of parametric surfaces, namely supertoroids with different parameters (a) and a Bézier surface (b).

6.1.3 Implicit Surfaces

An implicit surface equation has the following form:

$$f(x,y,z) = 0. \tag{6.6}$$



(a) (b)

Fig. 6.3. Examples of parametric surfaces: (**a**) supertoroids with different parameters; (**b**) a Bézier surface

Implicit surfaces divide the space into object interior and exterior regions. They allow us to talk about the solids defined by the interior of the implicit surfaces. Implicit surfaces are especially useful for collision detection and response in computer animation and ray surface intersection tests for rendering applications such as ray tracing. However, they are not suitable for generating polygonal approximations for the surfaces of the objects.

Collision detection applications generally require to test whether a point $\mathbf{p}$ is inside or outside of a surface, for which we can use the implicit equation of the surface.

$$\text{if} \quad \begin{cases} f(\mathbf{p}) = 0, & \mathbf{p} \quad \text{is on the surface.} \\ f(\mathbf{p}) > 0, & \mathbf{p} \quad \text{lies outside the surface.} \\ f(\mathbf{p}) < 0, & \mathbf{p} \quad \text{lies inside the surface.} \end{cases} \tag{6.7}$$

Implicit surface equations are also used for ray-surface intersection tests. A ray is represented parametrically as

$$\mathbf{r}(t) = \mathbf{r}_0 + t\,\mathbf{v} \tag{6.8}$$

where $\mathbf{r}_0$ is the ray origin, $\mathbf{v}$ is the direction vector of the ray, and t is the ray parameter. Then, we can test whether a ray intersects an implicit surface $f(x, y, z) = 0$ by substituting the parametric ray equation into the implicit surface equation and solving for the ray parameter t:

$$f(\mathbf{r}_0 + t\,\mathbf{v}) = 0 \tag{6.9}$$

6.1.4 Subdivision Surfaces

Subdivision surfaces is another popular surface modeling scheme. The idea of subdivision surfaces was first introduced by Catmull and Clark [12] and Doo and Sabin [13] independently in 1978. Other notable subdivision schemes are Loop [14], Butterfly [15], and $\sqrt{3}$-Subdivision [16]. Algorithmic definition of subdivision surfaces distinguishes them from standard spline surfaces. Subdivision surfaces resemble both polygon meshes and patch surfaces, and they take the best aspects of each representation technique. For instance, they can represent smooth surfaces with arbitrary topology and can be rendered smoothly owing to the well-defined surface normal, unlike low-resolution polygonal geometry. Simplicity, efficiency, and ease of implementation are the main advantages of subdivision surfaces.

Subdivision surfaces are constructed through recursive splitting and averaging operations. Splitting is performed by dividing a face into new faces and averaging is performed by taking a weighted average of neighboring vertices to obtain a new vertex. Splitting and averaging operations are shown in Fig. 6.4. The Doo-Sabin Subdivision Scheme is illustrated in Fig. 6.5 and the Catmull-Clark Subdivision Scheme is illustrated in Fig. 6.6. The results of applying various subdivision schemes to a cube are shown in Fig. 6.7.

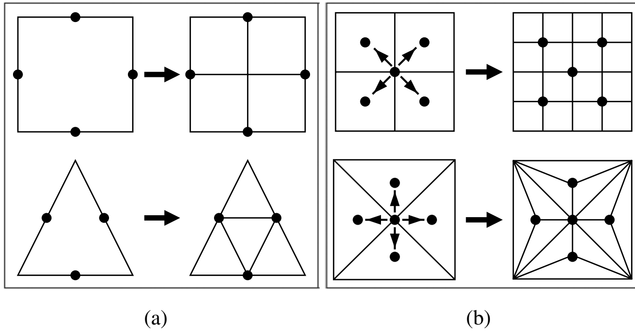


Fig. 6.4. Subdivision operations: (**a**) recursive splitting; (**b**) averaging

The shape of a subdivision surface is determined by a structured mesh of control points and a set of subdivision rules prescribing a procedure for refining the mesh to a finer approximation. The subdivision surface itself is defined as the limit of repeated recursive refinements. Subdivision surfaces satisfy all the usual requirements for surface representation that confront computer graphics practitioners. Starting with an initial polygonal mesh of arbitrary topology, a subdivision scheme is used to generate a new mesh that is the initial mesh for the next refinement. The repetitive application of this process will generate a sequence of polygonal meshes whose limit may be a smooth surface, assuming that appropriate conditions are satisfied [17]. This makes subdivision surfaces suitable as a multi-resolution mesh representation where switching between coarser and finer refinements can be easily achieved. The recursive nature of subdivision surfaces provides control over different levels of detail through adaptive subdivision. However, this nature also introduces a weakness for the modeling of sharp features such as creases or corners. Recently, some new techniques that perform modifications and additions to the subdivision rules have overcome this problem [18].

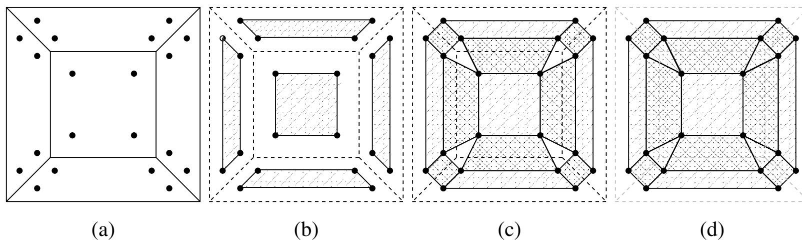


Fig. 6.5. The Doo-Sabin subdivision scheme: (**a**) generate new vertices with respect to Doo-Sabin subdivision masks; (**b**) form new faces inside the old faces by connecting the generated vertices; (**c**) form new faces for each edge in the coarser mesh by connecting the four new vertices adjacent to an old edge; (**d**) form new faces for each vertex in the old mesh by connecting the new vertices adjacent to an old vertex

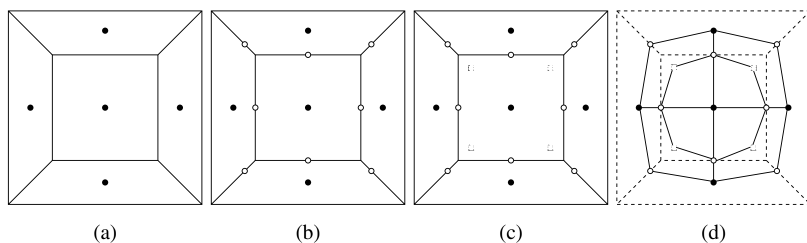


Fig. 6.6. The Catmull-Clark subdivision scheme: (**a**) generate new vertices for each face; (**b**) generate new vertices for each edge; (**c**) move each original vertex to a new location; (**d**) form new faces using the generated vertices

6.1.5 Point-based Representations

Points were first introduced as rendering primitives by Levoy [19] in 1985. As new display elements, points are also known as surfels [20]. Due to their

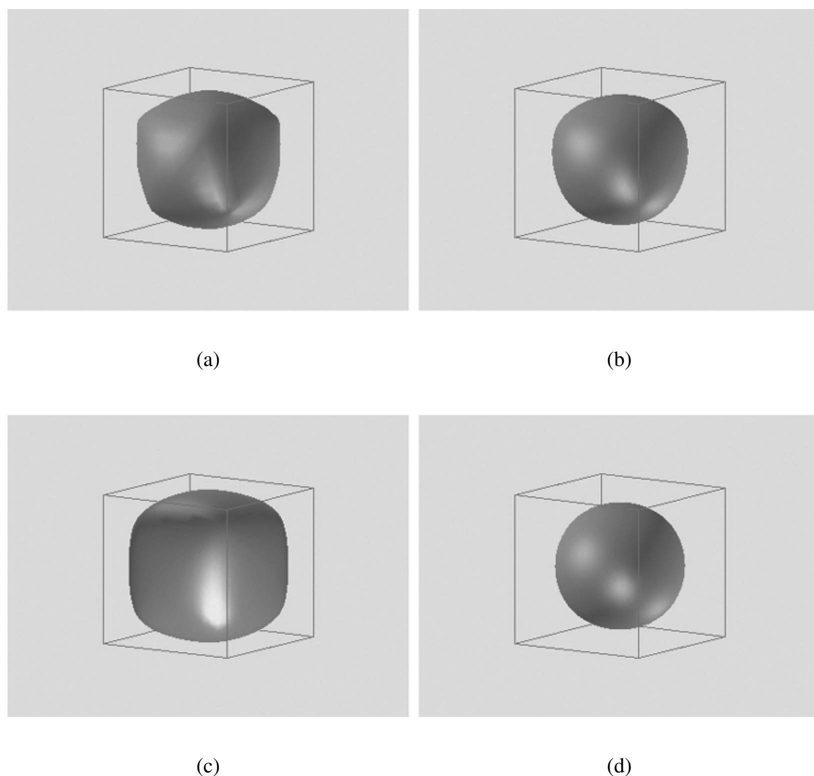


Fig. 6.7. Results of applying various subdivision schemes to a cube: (**a**) $\sqrt{3}$-Subdivision; (**b**) Loop Subdivision; (**c**) Doo-Sabin Subdivision; (**d**) Catmull-Clark Subdivision. The control mesh is the unit cube drawn in wireframe. Courtesy of Tekin Kabasakal

structural simplicity and flexibility, point samples are used to model shapes. Although point-based representations utilize more modeling primitives, since the primitives are simple and do not require explicit connectivity or topology information, these methods are efficient alternatives to mesh-based representations. Point sets do not have a fixed continuity class, contrary to meshes, which have piecewise linear C^0 connectivity. The continuity problem for meshes is handled by smoothing techniques such as applying Gouraud shading or subdivision operations. In contrast, point-based methods specify connectivity information implicitly through the spatial interrelation among the points [21]. Point-based modeling is in some sense similar to image-based modeling as it takes different views of an object as input and reconstructs the surface. However, point samples require more geometric information than image pixels and they are view-independent [22]. Moreover, the ease of insertion, deletion and repositioning of point samples makes these techniques suitable for dynamic settings with frequent changes of model geometry [23].

Point-based representations can be grouped into two: *piecewise constant point sampling* and *piecewise linear surface splats* [24]. Studies in the first group include Point Set Surfaces (PSS) [21, 25, 26]. PSS are used to represent shapes by taking a weighted average of the points. Normally, they can only be applied to regular samples due to the weighting scheme, which is based on a spatial scale parameter. Adamson et al. extend PSS to irregular settings by generalizing the weighting scheme [26]. Fleishman et al. [21] describe a progressive scheme, which reduces the amount of data required and improves modeling and visualization. They develop a simplification scheme for point sets to construct a base point set that represents a smoother version of the original shape. Then, they perform adaptive surface refinement.

Reconstruction of continuous surfaces from the irregularly-spaced point samples without losing visual quality is an important challenge for point-based methods. Moreover, hidden surface removal and transparency issues should be correctly handled. These difficulties have been overcome by the introduction of surface splats, first proposed by Zwicker et al. [27]. Surface splatting uses samples of the surface of an object to represent it [28]. Surface splats provide better visual quality and more efficiency by using an Elliptical Weighted Average (EWA) filter, which reduces aliasing artifacts. The performance limitations of this technique, which was originally purely software-based, have been overcome recently by utilizing the latest GPU technology. Botsch et al. discuss the capabilities of GPUs for hardware-based surface splatting in [24].

6.1.6 Volumetric Representations

Spatial subdivision techniques provide a natural way to represent solid objects and 3D scenes. These techniques simplify many calculations on solid objects and 3D scenes, such as boolean operations on solid objects to create complex objects from simpler ones, collision detection for animation, ray/surface intersections for raytracing, occlusion detection for the visualization of urban

scenery, etc. The only disadvantage of these techniques is the high storage cost since a solid object or a 3D scene is represented using a three-dimensional array. The high storage cost of spatial subdivision data structures are alleviated by using an adaptive subdivision of space instead of a uniform subdivision.

The unit element of a three-dimensional space is called a voxel. One common method to represent solid objects or 3D scenes is to use *octrees*, which are hierarchical tree structures. The three-dimensional space is partitioned into eight regions (octants), where each region corresponds to a node of the tree structure. Each octant is further subdivided recursively, if necessary. In case of a regular subdivision, the subdivision process terminates when a pre-defined depth is reached. In adaptive subdivision, the subdivision process terminates if the octant is completely unoccupied or a minimum resolution is obtained for the cells. The nodes of the octree structure point to the parts of the scene, or the solid object, contained in the part of the space to which that node corresponds. The octree representation is shown in Fig. 6.8.

Another spatial subdivision method to represent solid objects and 3D scenes volumetrically is Binary Space Partitioning (BSP) trees. The main idea is to adaptively partition the space into two regions with a plane. BSP trees are more efficient than octrees as they reduce the tree depth. They are especially useful for applications that require the subdivision of space into regions containing an equal number of scene objects.

Spatial subdivision techniques can be used for different types of object representations, including polygon meshes and surface patches. Different algorithms, such as intersection tests, traverse the octree structure recursively starting from the root. Details of spatial subdivision techniques can be found in [29].

Voxel-based representations are also used to reconstruct an environment from images obtained by multiple calibrated cameras. These representations generally use a regular 3D voxel array or an octree subdivision and the 3D scene is represented as a set of occupied voxels. These voxels can be colored and transparent and the surface normals associated with occupied voxels are stored for rendering purposes. Volume rendering techniques can be used to render such voxel-based 3D scenes. Unless the voxels are very small, rendering the surfaces of voxel-based 3D data produces a blocky appearance. Thus,

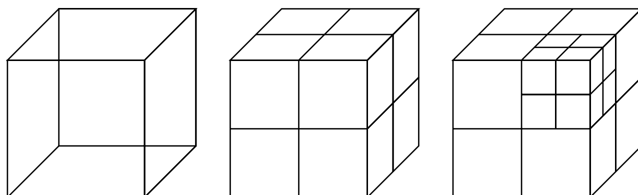


Fig. 6.8. The octree representation

refinement techniques should be applied to the meshes describing the surface to obtain a plausible appearance.

Some volumetric 3D reconstruction techniques compute the outer-bound approximation of the scene geometry, called *visual hull*, from silhouette images [30, 31, 32]. These techniques are applicable to images where foreground-background segmentation at each reference view is possible. The silhouette is the 2D projection of the corresponding 3D foreground object. The parts of the surface of the object that also lie on the surface of the visual hull can be reconstructed using silhouette-based approaches.

6.2 Animation

An illusion of motion is created when slightly different images are viewed in succession. Animation is the process of organizing and filming immobile objects to produce the images necessary to create such an illusion of movement. Animation techniques can be categorized into two main groups: *traditional animation* and *computer animation*.

Cartoon movies are the most widespread of traditional animation examples. They are produced by the method called *cel animation*. Cel animation is performed by the animators who draw and paint each frame by hand. Cartoon films have been an important sector of the entertainment industry since the 1930's, a consequence of the success of the Walt Disney Studios.

The second animation category is computer animation. Computer animation can be further subdivided into two groups: *computer-assisted animation* and *computer-generated animation* [33]. Computer-assisted animation is the computer-aided counterpart of traditional 2D cel animation. Papers, paint, brushes and various drawing materials are replaced by computers, scanners, cameras, mice, etc. The computer is mainly used for *cell painting* and *inbetweening*. In this way, traditional cartoon animation can be performed more efficiently and economically. Computer-generated animation is also known as true computer animation, where images are generated by means of rendering a 3D model. Motion is produced by modifying the model over time. The models have various parameters such as polygon vertex positions, spline knot positions, joint angles, muscle contraction values, colors, and camera parameters. Animation is performed by varying the parameters over time and rendering the models to generate the frames along the way [34].

Fundamental principles of traditional animation, such as *squash and stretch*, *timing and motion*, *anticipation*, *staging*, *follow through and overlapping action*, *straight ahead action and pose-to-pose action*, *slow in and out*, *arcs*, *exaggeration*, *secondary action*, and *appeal* [35], can be formalized and used as high level constructs in computer animation systems. In this way, most of the burden of generating realistic animation is left to the computer since the elements of an animated character move in harmony according to these

constructs. The application of these principles ensures that the characters have a personality appealing to the audience.

6.2.1 Hierarchical Approaches

Hierarchical modeling approaches store a 3D scene in the form of a tree or a graph structure. A very important property of these hierarchical approaches is that they unify modeling and animation. These representations store the primitive objects, including the lights and cameras, that make up the scene hierarchy (specified in the objects' local coordinate system) and the transformations to place them in world coordinates, in the nodes of a graph or a tree. Representative examples of such hierarchical techniques are scene graphs and scene tree representations. Virtual Reality Modeling Language (VRML), Java3D, and Open Scene Graph are widely used scene graph Application Programming Interfaces [36]. Figure 6.9 illustrates the scene tree representation for a 3D scene.

Transformation hierarchies is a modeling technique to represent articulated structures, such as humans and robots. It uses tree structures to represent articulated bodies. An intermediate node contains 3D transformation(s) that apply to all the children of that node. The leaf nodes correspond to primitive objects. Hierarchical modeling is implemented by using a matrix stack where the transformation matrices in the hierarchy are stored in the matrix stack. A recursive algorithm traverses the model hierarchy and calculates the composite transformations that correspond to the intermediate nodes. The algorithm stores the composite transformation matrices at the intermediate nodes of the tree structure by pushing them onto the stack so that they can be popped and re-used for the other branches of the same node. The primitives in the leaf nodes are drawn by applying the composite transformation sequence from the root to that node. Transformation hierarchies do not let the animator control the end-effectors of an articulated structure. They cannot handle

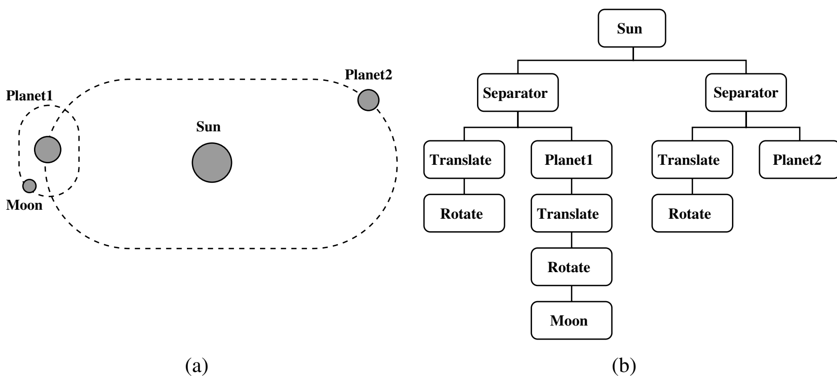


Fig. 6.9. (**a**) A 3D scene; (**b**) corresponding scene tree representation

closed-kinematics chains, such as keeping the feet on the ground. They cannot handle general constraints. Although there are more sophisticated techniques to model articulated structures, such as inverse kinematics, hierarchical modeling is a principal tool for modeling and animation [37].

6.2.2 Keyframing

One of the biggest problems in traditional cel animation is the necessity to draw and paint each frame by hand, which makes it highly labor-intensive. Lead animators, who want to work more efficiently, only draw the most important frames, which are called the keyframes. Then, low-level animators draw the remaining frames between the keyframes.

Computer animation, on the other hand, makes use of the computer to generate both the keyframes and the inbetween frames. The keyframes of a bouncing ball can be seen in Fig. 6.10, where the ball is depicted on the ground, at the highest point, and on the ground, respectively. Inbetween frames can be generated by interpolation techniques. One of these techniques is linear interpolation. If linear interpolation is used to generate inbetweens for a moving object, the object moves with constant velocity. Discontinuities and sudden leaps can be observed in the motion. In order to have a smooth motion, curve interpolation techniques such as Hermite or B-spline curves can be used. The inbetweens generated with different interpolation techniques can be seen in Fig. 6.11.

A bouncing ball does not have the same velocity throughout its path; the closer it is to the ground, the faster it moves. Thus, in order to obtain more realistic results, it is not sufficient to specify the path alone, but the velocity changes as well. In addition, various other properties of the object, such as its shape and color, may change during the motion. Figure 6.12 shows the motion of a deformable bouncing ball.

6.2.3 Physically-based Modeling and Animation

Methods used for modeling the shape and appearance of objects are not suitable for dynamic scenes where the objects are moving. The models do not interact with each other or with external forces. In real life, the behavior and form of many objects are determined by their physical properties, such as mass, damping, and the internal and external forces acting on the object.

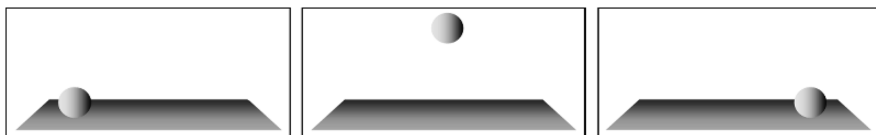
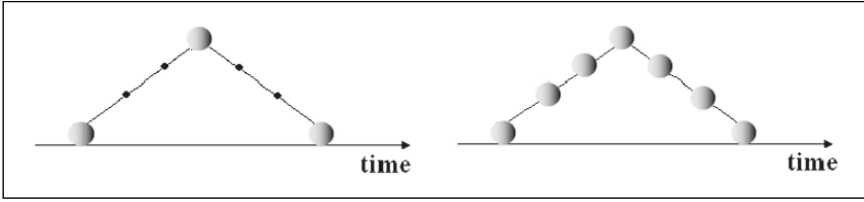
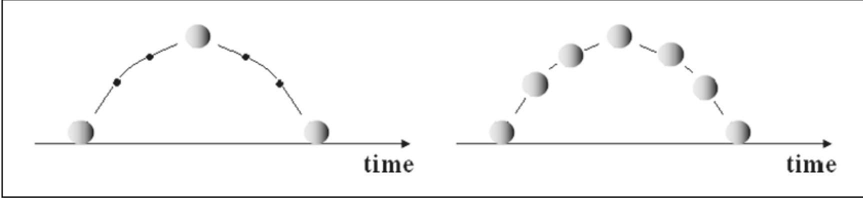


Fig. 6.10. The keyframes from the animation of a bouncing ball



(a)



(b)

Fig. 6.11. The inbetweens from the animation of a deformable bouncing ball generated with different interpolation techniques: (**a**) linear interpolation; (**b**) spline interpolation

The rigidity (or deformability) of the objects is determined by the elastic and inelastic properties (such as internal stresses and strains) of the material.

If we want to realistically animate the objects, we must model the physical properties of the objects so that they follow pre-defined trajectories and interact with the other objects in the environment, just like real physical objects. Physically-based techniques achieve this by adding physical properties to the models, such as forces, torques, velocities, accelerations, mass, damping, kinetic and potential energies, etc. Physical simulation is then used to produce animation based on these properties. To this end, the solution of the equations of motion is required so that the course of a simulation is determined by the initial positions and velocities of the objects, and by the forces

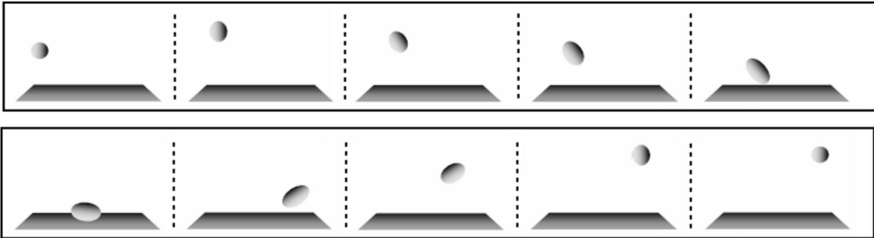


Fig. 6.12. The motion of a deformable bouncing ball

and torques applied to the objects as it moves. Today, physical simulations are widely used in the film industry and in game development and there are efficient techniques to approximate the physics involved.

When several objects are simultaneously involved in a computer animation, we encounter the problem of detecting and controlling object interactions. In such an animation, we may have more than one object moving around, or we may have impenetrable obstacles (such as walls) that do not move. When no special attention is paid to object interactions, the objects will sail through each other; this is usually not physically reasonable and produces a disconcerting visual effect. Whenever two objects attempt to penetrate each other (i.e., the surface of one object comes into contact with the surface of a second object), a collision is said to occur [38, 39].

The general requirement that arises then is an ability to detect collisions. Some animation systems at present do not provide even minimal collision detection; they require the animator to visually inspect the scene for object interactions and respond accordingly. This is time consuming and difficult even for keyframe or parameter systems where the user explicitly defines the motion; it is even worse for procedural and dynamic animation systems where the motion is generated by functions and laws defining their behavior. Although automatic collision detection is expensive to code and to run, it is a considerable convenience for animators, particularly when more automated methods of motion control, such as dynamics or behavioral control, are used.

The other related issue is the response to a collision once it is detected. Even keyframe systems could benefit from automatic suggestions about the motion of objects immediately following a collision; animation systems using dynamic simulation must respond to collisions automatically and realistically. Linear and angular momentum must be preserved, and surface friction and elasticity must be reasonable. An elaborate discussion of collision detection and response can be found in [40, 41].

6.2.3.1 Constraint-based Methods of Animation

Constraints provide a unified method to build objects and to animate them. The models assemble themselves as the elements move to satisfy the constraints. Constraints provide a way to specify the behavior of physical objects in advance without specifying their exact positions, velocities, etc. In other words, constraints are partial descriptions of the objects' desired behavior. So, given a constraint, we must determine the forces to meet the constraint and then find forces to maintain the constraint. A good deal of research has been done towards the use of constraint-based methods to create realistic animation [42, 43, 44, 45]. Many constraint-based modeling systems have been developed, including constraint-based models for the human skeleton [46] (in which the connectivity of segments and limits of angular motion on joints are specified), the *energy constraints* [47], and the *dynamic constraints* [48]. Examples of constraints are *point-to-nail constraint*, which is used to fix a

point on a model to a user-specified location in space, *point-to-point (attachment) constraint*, which is used to attach two points on different bodies to create complex models from simpler ones, *point-to-path constraint*, which requires some points on a model to follow an arbitrary user-specified path, and *orientation constraint*, which is used to align objects by rotating them [48]. Figure 6.13 shows a cloth patch constrained from two corners waving with gravity and wind forces.

6.2.3.2 Deformable Models

Modeling the behavior of *deformable objects* is an important aspect of realistic animation. To simulate the behavior of deformable objects, we must approximate a continuous model by using discretization techniques, such as finite difference and finite element methods. For finite difference discretization, a deformable object could be approximated by using a grid of control points where the points are allowed to move in relation to one another. The manner in which the points are allowed to move determines the properties of the deformable object. For example, in order to obtain the effect of an elastic surface, the grid points can be connected by springs. In fact, mass-spring systems are one of the simplest, yet most effective ways of representing deformable objects and they are very popular. By changing the spring forces acting on the particles that comprise an object, different deformable behaviors can be simulated.

To animate nonrigid objects in a simulated physical environment, the methods of elasticity and plasticity theory can be employed. However, such techniques are computationally demanding. Elasticity theory provides methods to construct the differential equations that model the behavior of nonrigid objects as a function of time.

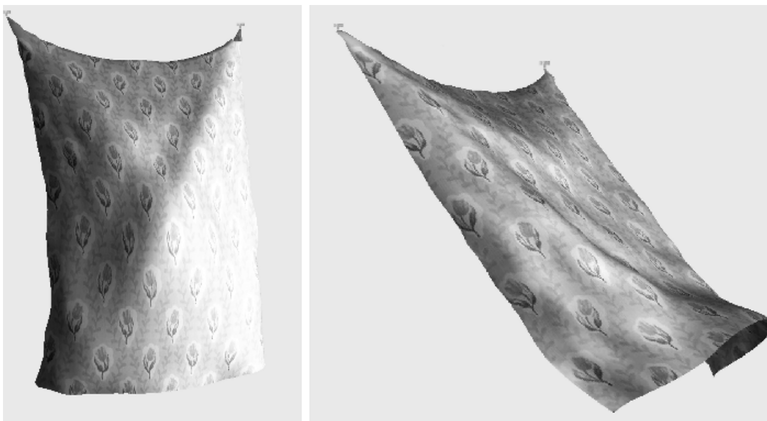


Fig. 6.13. A cloth patch constrained from two corners waving with the gravity and wind forces

To simulate the dynamics of elastically deformable models, there are two well-known approaches: the *primal formulation* [49] and the *hybrid formulation* [50]. These formulations use concepts from elasticity and plasticity theory and represent deformations of the objects using quantities from differential geometry, such as metric and curvature tensors [51]. The primal formulation works better for highly deformable materials since this formulation can handle nonlinear deformations; however the hybrid formulation is better for highly rigid materials since it can only handle small deformations that can be represented linearly.

To create animation with deformable models, the differential equations of motion must be discretized and the system of linked ordinary differential equations obtained from the discretization process must be solved as described in [50]. The *finite difference* or *finite element* methods can be used for the discretization process.

In addition to the approaches using elasticity theory to model the shapes and motions of deformable models, there are other approaches to model and animate deformable models. Witkin et al. formulate a model for nonrigid dynamics based on global deformations with relatively few degrees of freedom [42]. This model is restricted to simple linear deformations that can be formulated by affine transformations. In [52], Pentland and Williams describe the use of *modal analysis* to create simplified dynamic models of nonrigid objects. This approach breaks nonrigid dynamics down into the sum of independent vibration modes. It reduces the dimensionality and stiffness of the models by discarding high-frequency modes. Another method, based on physics and optimization theory, uses mathematical constraint methods to create realistic animation of flexible models [44]. This method uses reaction constraints for fast computation of collisions of flexible models with polygonal models, and it uses augmented Lagrangian constraints for creating animation effects, such as volume preserving squashing, and the molding of taffy-like substances. To model flexible objects, they use the finite element method. Thingvold and Cohen [53] define a model of elastic and plastic B-spline surfaces which supports both animation and design operations. The motion of their models is controlled by assigning different physical properties and kinematic constraints to various portions of the surface. Metaxas and Terzopoulos [54] propose an approach for creating dynamic solid models capable of realistic physical behaviors starting from common solid primitives such as spheres, cylinders, cones, and superquadrics [9]. Such primitives can deform kinematically in simple ways. To gain additional modeling power they allow the primitives to undergo parameterized global deformations (bends, tapers, twists, shears, etc.). Even though their models' kinematic behavior is stylized by the particular solid primitives used, the models behave in a physically correct way with prescribed mass distributions and elasticities. Metaxas and Terzopoulos also propose efficient constraint methods for connecting the dynamic primitives to make articulated models.

6.3 Rendering

Rendering techniques in computer graphics try to model the interaction of light with the environment to generate pictures of scenes [55]. This varies from implementation of the Phong illumination model, which is a first order approximation of the rendering equation [56], to very sophisticated global illumination techniques. More realistic renderings of the scenes can be obtained by using complex methods such as ray tracing [57, 58], or radiosity [59], and photon mapping [60], which calculate object-to-object interreflections, transmission, etc. Rendering techniques to be used in a 3DTV framework must generate realistic pictures and must be amenable to real-time implementations. A detailed discussion of real-time rendering can be found in [61].

6.3.1 Reflection and Illumination Models

Reflection models define the interaction of light with a surface. They take into account the material properties of the surface and the nature of the incident light, such as wavelength, the angle of incidence, etc. The reflective properties of materials are fully described by the Bidirectional Reflectivity Distribution Function (BRDF) [62]. BRDF is the ratio of the reflected radiance in a particular direction from a surface to the irradiance incoming from another direction to the surface. Each of the incoming and outgoing directions is represented with two angles (bidirectional). The BRDF is composed of *specular*, *uniform diffuse*, and *directional diffuse* components.

Illumination models define the nature of the light reflected from or refracted through a surface. Local illumination models only calculate the direct illumination from light sources on object surfaces. They do not consider object-to-object light interactions (reflections, transmissions, etc.). Light incident at a surface is composed of the reflected, scattered, absorbed and transmitted light. One of the most popular local illumination models used in computer graphics is the Phong illumination model. This model has three components:

- *Ambient light*: the amount of illumination in a scene which is assumed to come from any direction and is thus independent of the presence of objects, the viewer position, or actual light sources in the scene.
- *Diffuse reflection*: the light reflected in all directions from a point on the surface of an object. It does not depend on the viewer's position.
- *Specular reflection*: the component of illumination seen at a surface point of an object that is produced by reflection about the surface normal. It depends on the viewer's position and appears as a highlight.

When there is a single light source in the environment, the Phong illumination model is composed of these three components as (see Fig. 6.14):

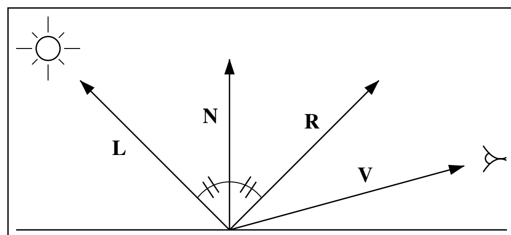


Fig. 6.14. Vectors used in the Phong illumination model

$$I = k_a i_a + [k_d(\mathbf{L} \cdot \mathbf{N}) i_d + k_s(\mathbf{R} \cdot \mathbf{V})^{n_s} i_s], \tag{6.10}$$

where

- i_a is the ambient intensity,
- i_d is the diffuse intensity of the light source,
- i_s is the specular intensity of the light source,
- k_a is the ambient reflection coefficient,
- k_d is the diffuse reflection coefficient,
- k_s is the specular reflection coefficient,
- $\mathbf{N}$ is the unit normal vector,
- $\mathbf{L}$ is the unit direction vector to the light,
- $\mathbf{R}$ is the unit reflection vector,
- $\mathbf{V}$ is the unit direction vector to the viewer,
- n_s is a shininess constant that decides how the light is reflected from a shiny point; it is very high for highly specular objects, such as mirror, which causes very shiny but small highlights.

The vectors used in the model are illustrated in Fig. 6.14. When there are multiple light sources in a scene, the contributions from the individual sources are summed as:

$$I = k_a i_a + \sum_{l=1}^{n} [k_d(\mathbf{N} \cdot \mathbf{L}_l) i_{l\,d} + k_s(\mathbf{R}_l \cdot \mathbf{V})^{n_s} i_{l\,s}] \tag{6.11}$$

6.3.2 Rendering Techniques

Rendering techniques are classified into *object-space* and *image-space* techniques. Object-space techniques calculate the intensity of light for each point on an object surface (usually represented using polygonal approximations) and then use interpolation techniques to interpolate the intensity inside each polygon. Flat shading, Gouraud shading [63], and Phong shading are in this category. They use local illumination models, e.g., the Phong illumination model [64], to calculate the intensities of points and a scan-line approach to

render the polygons. Radiosity is also an object-space technique; however, it is a global illumination algorithm that solves the rendering equation only for diffuse reflections. In contrast to object-space techniques, image-space techniques calculate intensities for each pixel on the image. Ray tracing is an image-space rendering algorithm. It sends rays to the scene from the camera through each pixel and recursively calculates the intersections of these rays with the scene objects.

To render a 3D scene, the visible parts of it for different views must be calculated. This requires the implementation of hidden surface algorithms together with rendering methods. Some rendering algorithms, such as ray tracing and radiosity, handle the visible surface problem implicitly while in others, such as Gouraud and Phong shading, that use local illumination models, it must be handled explicitly.

Images containing uniformly shaded objects are not very realistic since real objects have textures, bumps, scratches, and dirt on them. There are several rendering techniques that add realism to the rendering of uniformly shaded 3D scenes. Texture mapping [65, 66], environment mapping [67], and bump mapping [68] are representative examples of such methods.

Since scan-line renderers, such as Gouraud shading, are amenable to hardware implementations, they are more appropriate for the real-time display capabilities required for 3DTV than sophisticated rendering techniques, such as raytracing and radiosity. Image-based rendering is a recent and promising approach to the rendering of 3D scenes. Such techniques directly render new views of a scene from the acquired images, thus eliminating the need for an explicit scene representation phase.

6.3.2.1 Scan-line Renderers

Scan-line rendering is one of the most popular methods due to its low computational cost. Hardware implementation enables the rendering of very complex models in real-time, but even without hardware support, scan-line algorithms offer very good performance.

Scan-line algorithms work in object-space by iterating over the polygons (mostly triangles) of scene objects. First, the frame buffer, which holds the pixel intensity values, and the z-buffer, which manages pixel depth values relative to the camera, are initialized. Next, the polygons are painted by projecting them onto the screen and filling them by scan-converting into a series of horizontal spans. While iterating over the scan lines to paint a polygon, the intersection points of the scan line with the polygon edges are computed and the horizontal spans inside the polygons are painted pixel by pixel. For each pixel inside a polygon, intensity and depth values are calculated in order to paint each pixel correctly. Depending on the z-buffer depth value of a pixel, it can be colored or just skipped. If the depth of a polygon pixel is less than the value for the respective screen pixel in the z-buffer, the z-buffer is updated and

the pixel is colored by the corresponding value in the frame buffer, otherwise, it is ignored.

Flat shading: By using a local illumination model, e.g., the Phong illumination model, we can calculate an intensity value for the RGB color components at a single position for each polygon. We can then fill every projected polygon approximating an object with the intensity value calculated for this polygon. This method quickly generates a curved-surface appearance for an object approximated with polygons.

Gouraud shading: Flat shading generates intensity discontinuities along polygon edges. Although increasing the number of the polygons that compose an object gives a smoother appearance when flat shading is used, it requires more computational power. Gouraud shading was developed to generate a smooth appearance for objects using only a small number of polygons. Gouraud shading linearly interpolates the intensity values across the surface of a polygon. The basic steps of Gouraud shading are as follows:

- The vertex normal vectors are calculated by averaging the face normals surrounding the vertex as (see Fig. 6.15 (a)):

$$N_V = \frac{\sum_{i=1}^{n} N_i}{|\sum_{i=1}^{n} N_i|} \tag{6.12}$$

- An illumination model is applied to each vertex to calculate the vertex intensity. The brightness at each vertex is calculated.
- Each projected polygon is shaded by using a modified scan-line polygon filling algorithm. Moving from scan line to scan line, the intensity values of the pixels are linearly interpolated for each projected polygon. Any number of quantities can be interpolated at this step. For instance, colored surfaces are rendered by interpolating the color component R, G and B values. Figure 6.15 (b) illustrates how the intensity values are interpolated along the edges of the polygon and the pixels inside the polygon.

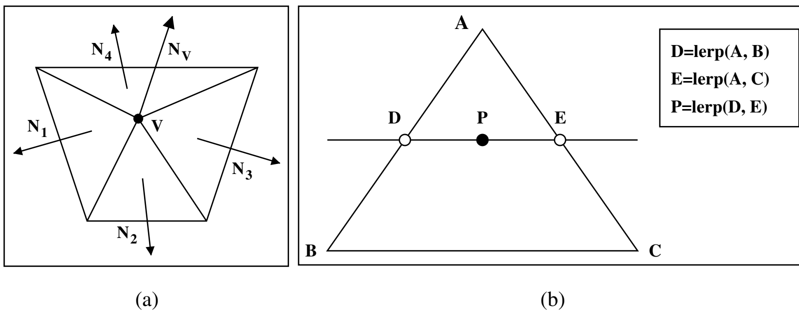


Fig. 6.15. Gouraud shading: (**a**) calculating vertex normals from face normals; (**b**) linear interpolation (*lerp*) of the intensities along the polygon edges and interiors

Gouraud shading is a simple and fast technique, which is supported by most of the graphics accelerators today. It does have some deficiencies as a result of the linear interpolation scheme; for instance, discontinuities appear as odd looking bright or dark bands, called Mach Bands, on the surface of the object. It also fails to give good results when the color changes quickly, e.g., specular highlights.

Phong shading: The disadvantages of Gouraud shading have been overcome by Phong shading. The basic steps of the Phong shading algorithm are as follows:

- The vertex normal vectors are calculated by averaging the surface normals surrounding the vertex. This step is the same as the first step in Gouraud shading.
- The vertex normals are linearly interpolated over the polygon surface.
- A modified version of scan-line polygon filling algorithm is applied to render projected polygons. An illumination model is used to calculate pixel intensities using the interpolated normal vectors.

Phong shading gives more accurate results than Gouraud shading; however, since the intensities are calculated explicity for each pixel, this method requires more computations. Object rendering techniques using local illumination models are illustrated in Fig. 6.16.

6.3.2.2 Ray Tracing

Ray tracing tries to imitate the light-object interactions in nature by modeling the behavior of photons emitted from light sources. When photons hit the objects, they bounce losing some of their energy. When the photons lose most of their energy, they are absorbed. If the objects are transparent or translucent, some of the light energy is transmitted. To imitate the behavior of photons for photorealistic image synthesis, we must take into account the effect of the photons that hit the image plane and come to our eyes. These photons emanate from the light sources and come to the image plane after successive bounces from the objects in the scene, thus contributing to the intensity and color of the pixels in the image. Photons that do not reach the image plane make no contribution to the image.

The trajectories that photons follow can be modeled with rays. Backward ray tracing starts from the eye and sends rays to the pixels in the image plane, instead of following the rays emitted from light sources, to avoid tracing the rays that do not contribute to the image. The light intensity of the image pixels are determined by the rate at which the photons hit and by their energies. The color of pixels are determined by the distribution of the wavelengths of incoming photons. The rays sent from the viewer (camera) to the image pixels are called *eye (pixel) rays*. If they hit a light source in the scene, we use the intensity of the light source to determine the intensity of the pixel. If the

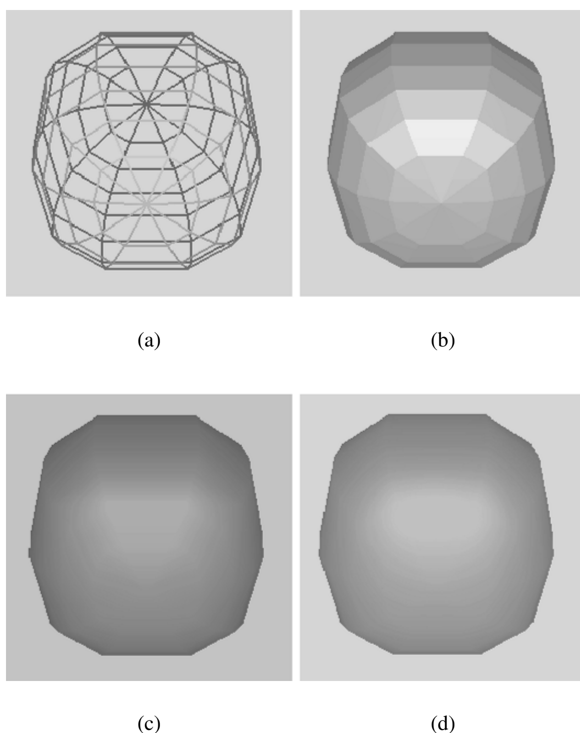


Fig. 6.16. Object rendering using local illumination models. (**a**) wireframe; (**b**) flat shading; (**c**) Gouraud shading; (**d**) Phong shading

ray does not hit anything in the scene, we set the intensity of the pixel to zero. If we hit a surface point, we recursively follow more rays to determine where the light striking that surface point came from. This is done by sending a *reflection ray* in the specular reflection direction at that point (which is calculated according to the incoming ray direction and the surface normal) and a *transmission ray* according to the theory of refraction (Snell's Law is used to calculate the transmission ray direction). We also send *illumination (shadow) rays* to the light sources to understand whether the surface point sees a light source or not. We add the contributions coming from reflection and transmission directions and the contribution of the light sources that see the point to find the intensity and color. The reflection and transmission rays are recursive rays, just like eye rays, in the sense that when they hit a surface point new reflection and transmission rays are fired. Illumination rays are not recursive. Figure 6.17 illustrates how backward ray tracing works [58]. Figure 6.18 depicts a raytraced scene.

In ray tracing, most of the time is spent for intersection calculations. Different objects need different ways to find the intersections. Ray/surface intersections can be easily found for the objects whose implicit functions are

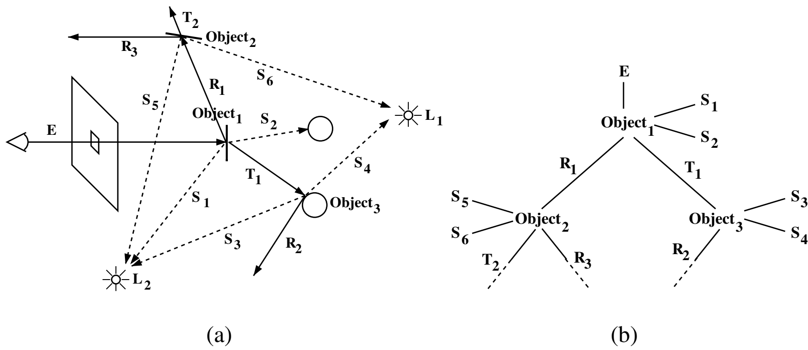


Fig. 6.17. Backward ray tracing. (**a**) An eye ray E sent from the eye to a pixel is traced through successive bounces in the scene. Reflection rays are labeled with R, transmission rays are labeled with T, and shadow rays are labeled with S. (**b**) Corresponding ray tree. Reprinted from [58] with permission. ©1988 Elsevier

known. Techniques proposed to accelerate ray tracing generally try to make intersection tests faster by using bounding boxes or reducing the number of intersection tests by utilizing bounding volume hierarchies and spatial coherence schemes. To make ray-object intersection tests faster, simple bounding volumes enclosing the objects are first tested with the rays. If the rays intersect with the bounding volumes, then real ray-object intersection tests are performed. Spatial coherence schemes first preprocess the scene to construct a spatial subdivision structure, such as a regular 3D grid (Spatially Enumerated Auxiliary Data Structure-SEADS) [69], uniform or adaptive octrees [70], or Binary Space Partition (BSP) trees [58]; the objects in the scene are stored in the nodes of the spatial subdivision structure. The ray tracing algorithm only

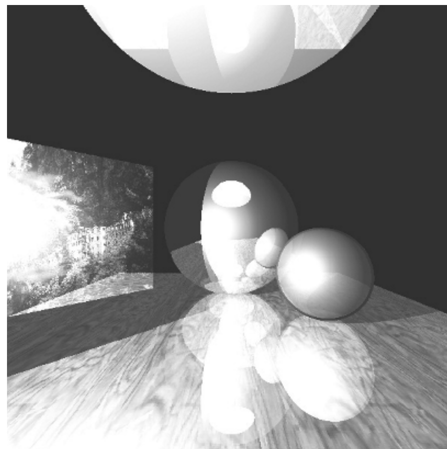


Fig. 6.18. An image generated with ray tracing. Courtesy of Okan Arıkan

makes intersection tests for the objects in the nodes of the spatial subdivision structure that the ray passes through.

Two other important acceleration techniques for ray tracing are *adaptive depth control* [71] and *first-hit speed-up* [72]. Ray tracing produces a ray tree for each eye ray, the depth of which increases with each reflection and transmission that does not leave the scene. Since rays at low levels contribute little to the image, adaptive depth control stops firing reflection and transmission rays when the computed intensity for a point becomes less than a certain threshold. This is checked for an intersection point by multiplying the specular reflection and transmission coefficients for the intersections up to that point and comparing it with a pre-defined threshold.

Even for highly reflective scenes, the average ray tree depth does not exceed two if we use adaptive depth control. Since most of the intersection calculations are done in the step, Weghorst proposed to use a z-buffer algorithm as a pre-processing step to determine the first hit. Then the ray tracing algorithm is executed by using the intersection points for the objects that are stored in the z-buffer.

Ray tracing can only handle specular reflections where the light sources are point light sources (although there are some variations of ray tracing, like *distributed* ray tracing, that increase the realism of the rendering by adding spatial aliasing, soft shadows, and depth-of-field effects, by firing more rays and distributing the ray origins and directions statistically based on probability distribution functions) [73].

6.3.2.3 Radiosity

The main motivation for radiosity is to accurately model the diffuse object-to-object reflections since most real environments consist mainly of objects that reflect light diffusely. A very large proportion of the light energy comes from direct illumination from light sources and diffuse reflections. For photorealistic image synthesis, the physical behavior of light must be modeled. Since the intensity and distribution of light is governed by energy transfer and conservation principles, these must be taken into account to accurately simulate the physical behavior of light transport between light sources and materials in a scene [59].

Radiosity is a method to determine the intensity of light diffusely reflected within an environment. It is an object-space algorithm that solves for the intensity at discrete points or surface patches within an environment. The solution is thus independent of the viewer position. The radiosity solution (which are intensities of patches in the environment) is then input to a rendering algorithm (such as Gouraud shading) to compute the image for a particular view position. This final phase does not require much computation and different views are easily obtained from the view-independent solution. This makes radiosity very attractive for dynamic scenes, e.g., architectural walkthroughs, where the geometry is fixed but the viewer position is dynamic [74, 75].

The main assumption of the method is that all the surfaces in the scene are perfect diffuse (Lambertian) reflectors. Unlike ray tracing, radiosity also assumes that the surfaces in the scene are decomposed into polygonal patches. Light sources and other objects are treated uniformly; the patches may be emitters (area light sources) or other objects that do not emit light.

Radiosity, B, is defined as the energy leaving a surface patch per unit area per unit time and is the sum of emitted and the reflected energy. The radiosity B_i of a patch i is given by

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{dA_j dA_i} dA_j, \tag{6.13}$$

The form factor, $F_{dA_j dA_i}$, determines the fraction of energy leaving dA_j that arrives on dA_i. The integral is over all patches j in the environment. R_i is the fraction of the incident light that is reflected from the patch i in all directions, called the *reflectivity* of the patch i. We can discretize an environment into n patches and assume the radiosity and emittance over a patch is constant. If we replace $F_{A_j A_i}$ by F_{ji} to simplify the notation, the radiosity of a discrete patch is given by

$$B_i A_i = E_i A_i + R_i \sum_{j=1}^{n} B_j F_{ji} A_j \tag{6.14}$$

The reciprocity relationship between two patches is given by

$$F_{ij} A_i = F_{ji} A_j \quad \text{and} \quad F_{ij} = F_{ji} \frac{A_j}{A_i} \tag{6.15}$$

Then, the radiosity equation becomes

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij} \tag{6.16}$$

For an environment containing n patches, we have a linear system of equations for the radiosities of the patches

$$\begin{bmatrix} 1 - R_1 F_{11} & -R_1 F_{12} & \cdots & -R_1 F_{1n} \\ -R_2 F_{21} & 1 - R_2 F_{22} & \cdots & -R_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -R_n F_{n1} & -R_n F_{n2} & \cdots & 1 - R_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \tag{6.17}$$

The emittance values (E_i) are non-zero for only light sources and the reflectivities (R_i) are known. The form factors F_{ij} are calculated based on the geometry of the patches. The form factors for a patch can be calculated analytically by placing a hemisphere around the patch and using the relative orientation and distance from this patch to the other patches. However,

this is only possible for very simple geometries. In most cases, approximation methods, such as the hemi-cube approach [76], are used to calculate the form factors. Note that the form factors F_{ii} are zero for planar or convex patches. Since the form factors from a patch to all other patches add up to 1 ($\sum_{j=1}^{n} F_{ij} = 1$) and R_i is always less than 1, the matrix in the linear system of (6.17) is diagonally dominant and guaranteed to converge [75].

The classical radiosity algorithm calculates the radiosity of the patches one at a time by *gathering* the radiosities from all other patches. In this approach, it is not possible to obtain an intermediate solution for the patches during the solution of the radiosity algorithm. Another variant of the radiosity algorithm, called *progressive refinement* radiosity [77], updates the radiosity of all patches in a scene by *shooting* the radiosity of a patch to all other patches. In this way, the radiosity of all the patches are updated simultaneously and it is possible to obtain intermediate solutions during the solution of the algorithm. If these partial solutions are rendered, the scene is lit progressively. This idea can be further elaborated by sorting the patches with respect to their emittance values. If the patches with higher emittance values (light sources) are processed first by shooting their radiosities to the other patches, it is possible to obtain very good approximations of the final images in the earlier steps.

Hierarchical radiosity is another improvement to reduce the computational complexity of the classical radiosity algorithm [78]. The dominant term in the computational complexity of the algorithm comes from form factor calculations, which are $(O(n^2))$ for a scene containing n patches since we have to compute the form factors from each patch to all other patches. During the solution, hierarchical radiosity computes the light interactions between separated groups of patches (clusters) as a single interaction. Thus, it starts with a set of coarse initial patches and forms a quadtree with respect to the form factor estimations. Some of the patches are then subdivided on-the-fly according to the form factor estimations and brightness values, and the radiosity solution is refined. Figure 6.19 shows two scenes rendered using hierarchical radiosity.

There are attempts to combine ray tracing and radiosity. Wallace et al. describe a multi-pass method where an extended radiosity solution is applied in the first pass and a ray tracing solution is applied in the second pass. The method successfully calculates the effects of different light transport mechanisms: diffuse-to-diffuse, diffuse-to-specular, specular-to-specular, and specular-to-diffuse, to some extent. It makes certain assumptions about the rendered scenes, e.g., that the number of specular surfaces is limited and that they cannot see each other, in order to prevent infinite reflections [79].

6.3.2.4 Photon Mapping

Photon mapping is a new approach to the global illumination of the scenes, which makes realistic rendering more affordable. Photon mapping uses forward ray tracing (i.e., sending rays from light sources) to calculate reflecting and refracting light for the photons. It is a two-step process (distributing the photons

Fig. 6.19. Images of the University of California, Berkeley Soda Hall (Rooms 380 and 420) generated with hierarchical radiosity. Courtesy of Ali Kemal Sinop

and rendering the scene) that works for arbitrary geometric representations, including parametric and implicit surfaces; it calculates the ray-surface intersections on demand. Figure 6.20 shows an image generated with photon mapping.

6.3.2.5 Image-based Rendering

Unlike the approaches described above that render a 3D scene composed of objects modeled with different geometric modeling techniques, there is another rendering approach, called image-based rendering (IBR), that directly renders a scene from the pre-acquired photographs. High-quality visualization results

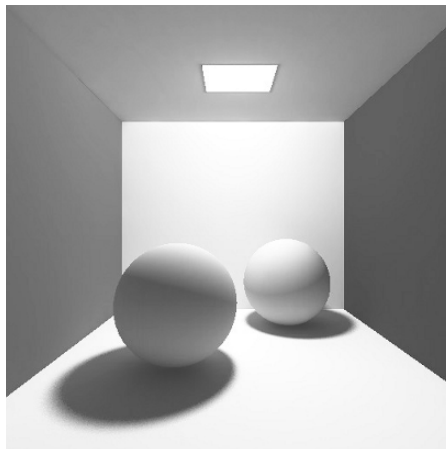


Fig. 6.20. An image of the Cornell Box generated with photon mapping. Courtesy of Atılım Çetin

can be obtained depending on both the quality and quantity of the reference images. The main motivation for IBR is to reduce the modeling bottleneck, since the creation of an object or scene model is a highly demanding task and it is expensive to represent all the surface details with geometric primitives.

The roots of IBR date back to texture and environment mapping techniques. In addition to their original functions of approximating reflections of the environment on a surface, environment maps are also used to display an outward-looking view of the environment from a fixed location with varying orientation [80]. Chen [81] uses such a technique by employing 360-degree cylindrical panoramic images to construct a virtual environment. Camera panning and zooming are simulated by digitally warping the virtual environment.

Unfortunately, interpolation between two images by warping fails in cases where previously occluded areas become visible. Another interpolation approach is to use corresponding feature points between two images and thus to compute the depth of each pixel by using the information of the camera positions. Chen and Williams [82] describe the view interpolation technique that performs morphing on adjacent images to create an image of a new in-between viewpoint. The method uses the camera's position and orientation and the range data of the images to determine a pixel-by-pixel correspondence between the images. The correspondence maps between two successive images are computed and they are stored as a pair of morph maps. The precomputation of the morphing provides efficiency. Another method, Layered Depth Images, solves the occlusion problem by associating more than one depth value to a pixel. These values correspond to the depth of each surface layer that a ray through the pixel intersects [83].

A 5D function that describes the intensity of light observed from every position and direction in 3D space is called the "plenoptic function" [84]. The plenoptic function is defined as:

$$p = P(\theta, \phi, V_x, V_y, V_z)$$

where (V_x, V_y, V_z) represent a point in space, θ represents the azimuth angle and ϕ represents the elevation angle. It is also possible to include the time parameter to the plenoptic function in a dynamic scene. IBR aims to reconstruct the plenoptic function from a set of images. In fact, the plenoptic function describes the set of all possible environment maps for a given scene in computer graphics terminology [85]. Once this function is obtained, the reconstruction of the scene becomes straightforward.

Levoy and Hanrahan propose a technique called "Light Field Rendering" that is based on the idea of interpreting the input images as 2D slices of the light field, which is a 4D function based on the plenoptic function [80]. The light field characterizes the radiance as a function of position and direction in unobstructed space. Generating new views corresponds to extracting and resampling a slice. Lumigraph is a similar method that also uses a 4D function,

which is a subset of the plenoptic function [86]. Lumigraph enables the generation of new images of an object independent of the geometric complexity or illumination conditions of the scene or object. McMillan and Bishop [85] also present an IBR system that is based on the sampling, reconstruction and resampling of the plenoptic function.

IBR only requires the acquisition of photographs; thus scene and object representation is comparably easy [87]. Standard geometric and lighting techniques sometimes lack the proper models to simulate some real-world shading and appearance effects. Since IBR methods do not require explicit geometric models to render real-world scenes, they can reproduce real-world shading and appearance effects faithfully without having to explicitly model them. Although IBR methods have significant memory requirements (e.g., light fields) and their computational complexity is very high, the computational cost of interactively viewing the scene is independent of the complexity of the scene. Moreover, IBR techniques can also combine real-world photographs with computer-generated images to be used as pre-acquired images. Thus, with all its advantages, IBR is a promising approach to be used in a 3DTV framework. However, there are many challenges, such as feature correspondence, camera calibration, and the construction of plenoptic functions, that need to be addressed for IBR to be applicable as a general rendering technique for complex dynamic scenes [88].

6.3.2.6 Volume Rendering

Volumetric data contains scalar values for 3D locations in space. The 3D locations for which the volume data are defined determines the type of the volumetric data. If the scalar values are defined for a regular 3D array of locations, the data can be represented in the form of *structured grids* where the connnectivity between the vertices is defined implicitly. If the distribution of data points do not follow a regular pattern, the connectivity of the vertices should be defined explicitly. These *unstructured grids* are generally represented by using tetrahedral cells.

Volume rendering techniques are classified as *direct* and *indirect*. Indirect volume rendering methods, such as Marching Cubes [89], extract an intermediate geometric representation of the surfaces from volume data and render them using surface rendering methods. Indirect methods are faster and more suitable for applications where the visualization of the surfaces of the volume data is important. Visual hull techniques can also be regarded as an indirect volume rendering approach since they extract and render the surface of the scene geometry. Direct volume rendering techniques render the volume data without generating an intermediate representation; thus facilitating the visualization of the inside of a material, such as partially transparent body fluids. Structured volume data can be directly visualized in real-time using special-purpose hardware [90].

Direct volume rendering algorithms for unstructured grids are classified as *image-space*, *object-space* and *hybrid*. Image-space methods traverse the image-space by casting a ray for each pixel. The ray is followed inside the volume to sample and compose the volume data along the ray. In object-space methods, the volume is traversed in object-space and the cells are depth-sorted with respect to the current viewpoint. Then, the cells are projected onto the image-plane in sorted order and their contributions are composited. In hybrid methods, the volume is traversed in object order and the contributions of the cells to the final image are accumulated in image order [91].

Volumetric datasets can also be rendered by using advanced per-pixel operations available in the rasterization stage and in the graphics hardware [92]. Although this approach works for both structured and unstructured grids, it is much more successful for structured grids since it can use 3D textures. In both cases, it avoids any polygonal representation using per-pixel operations.

Currently, volume visualization techniques are most commonly used in medical imaging and scientific simulations, such as Computational Fluid Dynamics or geophysical simulations. The improvement of 3D scene capture technologies and high-performance computers provide easy acquisition of volume data so that volume visualization techniques can be used in other applications, such as 3DTV. However, this requires real-time implementation of these techniques and although there are specialized hardware for direct volume rendering of structured data, direct volume rendering of unstructured data, namely the tetrahedral mesh representations, is still far from being real time.

6.4 Conclusions

3D shape modeling is an indispensable component of scene representation for 3DTV. Dynamic mesh representations provide a suitable way of representing 3D shapes. Polygonal meshes are amenable to hardware implementations; thus, they are suitable for a 3DTV framework where real-time performance is required. Volumetric representations provide a good alternative for 3DTV because images acquired from multiple calibrated cameras provide the necessary information for volumetric models. Point-based representations are also promising for 3DTV because the results of 3D data acquisition methods such as laser scans already represent the scene in a point-based manner.

Since the scenes mostly contain dynamic objects, modeling the motion becomes important. Animation techniques that have potential for real-time implementations are promising approaches to be used in a 3DTV framework. Today, physically-based modeling and animation techniques are widely used in the film industry and in game development and there are efficient techniques to approximate the physics involved; thus these techniques have potential for use in 3DTV.

Since scan-line renderers are amenable to hardware implementations, they are more appropriate for the real-time display capabilities required for 3DTV

than sophisticated rendering techniques. Point-based software renderers can realistically render models containing millions of points in a second. Image-based rendering is a very successful and promising rendering scheme for 3DTV as it directly makes use of the captured images. However, there are many challenges to be addressed for IBR to be applicable as a general rendering technique for complex dynamic scenes.

Acknowledgment

This work is supported by the EC within FP6 under Grant 511568 with the acronym 3DTV.

References

1. A. H. Barr, "Global and local deformations of solid primitives," *ACM Computer Graphics (Proc. SIGGRAPH'84)*, Vol. 18, No. 3, pp. 21–30, Jul. 1984.
2. T. W. Sederberg and S. R. Parry, "Free-form deformation of solid geometric models," *ACM Computer Graphics (Proc. SIGGRAPH'86)*, Vol. 20, No. 4, pp. 151–160, Aug. 1986.
3. D. Hearn and P. Baker, *Computer Graphics with OpenGL, 3rd Edition*. Englewood Cliffs, NJ: Prentice Hall, 2003.
4. B. Mandelbrot, *Fractals: Geometry of Nature*. New York: Freeman Press, 1982.
5. P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants (The Virtual Laboratory)*. Springer, 1996.
6. H. Hoppe, "Progressive meshes," *ACM Computer Graphics (Proc. SIGGRAPH'96)*, pp. 99–108, Jul. 1996.
7. ——, "View-dependent refinement of progressive meshes," *ACM Computer Graphics (Proc. SIGGRAPH'97)*, pp. 189–198, Jul. 1997.
8. D. Luebke and C. Erikson, "View-dependent simplification of arbitrary polygonal environments," *ACM Computer Graphics (Proc. SIGGRAPH'97)*, pp. 199–208, Jul. 1997.
9. A. H. Barr, "Superquadrics and angle-preserving transformations," *IEEE Computer Graphics and Applications*, Vol. 1, No. 1, pp. 11–23, Jan. 1981.
10. R. Bartels, J. Beatty, and B. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Los Alamos, CA: Morgan Kaufmann, 1987.
11. P. Bézier, *Numerical Control — Mathematics and Applications*. London: John Wiley & Sons, 1972.
12. E. Catmull and J. Clark, "Recursively generated b-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, Vol. 10, No. 6, pp. 350–355, 1978.
13. D. Doo and M. Sabin, "Behaviour of recursive subdivision surfaces near extraordinary point," *Computer-Aided Design*, Vol. 10, No. 6, pp. 356–360, 1978.
14. C. Loop, "Smooth subdivision surfaces based on triangles," Master's thesis, Department of Mathematics, University of Utah, 1987.

15. N. Dyn, D. Levin, and J. A. Gregory, "A butterfly subdivision scheme for surface interpolation with tension control," *ACM Trans. on Graphics*, Vol. 9, No. 2, pp. 160–169, 1990.
16. L. Kobbelt, "$\sqrt{3}$-Subdivision," *ACM Computer Graphics (Proc. of SIGGRAPH'00)*, pp. 103–112, 2000.
17. D. Zorin and P. Schroder, "Subdivision for modeling and animation," *ACM SIGGRAPH Course Notes*, 2000.
18. A. Lee, H. Moreton, and H. Hoppe, "Displaced subdivision surfaces," *ACM Computer Graphics (Proc. SIGGRAPH'00)*, pp. 85–94, Jul. 2000.
19. M. Levoy and T. Whitted, "The use of points as display primitive," University of North Carolina at Chapel Hill, Tech. Rep. TR-85-022, 1985.
20. H. Pfister, M. Zwicker, J. Van Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," *ACM Computer Graphics (Proc. of SIGGRAPH'00)*, pp. 335–342, 2000.
21. S. Fleishman, D. Cohen-Or, M. Alexa, and C. Silva, "Progressive point set surfaces," *ACM Trans. on Graphics*, Vol. 22, No. 4, pp. 997–1011, 2003.
22. J. Grossman and W. Dally, "Point sample rendering," in *Proceedings of Eurographics Rendering Workshop*, pp. 181–192, 1998.
23. M. Pauly, L. Kobbelt, and M. Gross, "Point-based multiscale surface representation," *ACM Trans. on Graphics*, Vol. 25, No. 2, pp. 177–193, 2006.
24. M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's GPUs," in *Proceedings of Eurographics Symposium on Point-Based Graphics*, pp. 17–24, 2005.
25. M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva, "Point set surfaces," in *Proceedings of IEEE Visualization'01*, pp. 21–28, 2001.
26. A. Adamson and M. Alexa, "Anisotropic point set surfaces," in *Proceedings of AFRIGRAPH'06*, 2006.
27. M. Zwicker, H. Pfister, J. Van Baar, and M. Gross, "Surface splatting," *ACM Computer Graphics (Proc. of SIGGRAPH'01)*, pp. 371–378, 2001.
28. S. Rusinkiewicz and L. Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *ACM Computer Graphics (Proc. of SIGGRAPH'00)*, pp. 343–352, 2000.
29. H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, Vol. 16, No. 2, pp. 187–260, 1984.
30. A. Laurentini, "The visual hull concept for silhouette based image understanding," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 16, No. 2, pp. 150–162, 1994.
31. J.-M. Hasenfratz, M. Lapierre, J.-D. Gascuel, and E. Boyer, "Real-time capture, reconstruction and insertion into virtual world of human actors," in *Proceedings of Eurographics Vision, Video and Graphics Conference*, pp. 49–56, 2003.
32. G. Slabaugh, W. Culbertson, T. Malzbender, M. Stevens, and R. Schafer, "Methods for volumetric reconstruction of visual scenes," *International Journal of Computer Vision*, Vol. 57, No. 3, pp. 179–199, 2004.
33. R. Parent, *Computer Animation: Algorithms and Techniques.* Los Altos, CA: Morgan-Kaufmann, 2001.
34. A. Witkin, "Animation," in *Computer Graphics I Course Notes*, School of Computer Science, Carnegie-Mellon University, 1995.
35. J. Lasseter, "Principles of traditional animation applied to 3D computer animation," *ACM Computer Graphics (Proc. SIGGRAPH'87)*, Vol. 21, No. 4, pp. 35–44, Jul. 1987.

36. E. Angel, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley, 2006.
37. A. Witkin, "Hierarchical modeling," in *Computer Graphics I Course Notes*, School of Computer Science, Carnegie-Mellon University, 1995.
38. D. Baraff, "Analytical methods for dynamic simulation of non-penetrating rigid bodies," *ACM Computer Graphics (Proc. SIGGRAPH'89)*, Vol. 23, No. 3, pp. 223–232, Jul. 1989.
39. M. Moore and J. Wilhems, "Collision detection and response for computer animation," *ACM Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 4, pp. 289–298, Aug. 1988.
40. P. Jiménez, F. Thomas, and C. Torras, "3D collision detection: A survey," *Computers & Graphics*, Vol. 25, No. 2, pp. 269–285, 2001.
41. M. C. Lin and S. Gottschalk, "Collision detection between geometric models: A survey," in *Proceedings of IMA Conference on Mathematics of Surfaces*, pp. 37–56, 1998.
42. A. Witkin, M. Gleischer, and W. Welch, "Interactive dynamics," *ACM Computer Graphics (Proc. SIGGRAPH'90)*, Vol. 24, No. 4, pp. 11–22, Aug. 1990.
43. A. Witkin and W. Welch, "Fast animation and control of nonrigid structures," *ACM Computer Graphics (Proc. SIGGRAPH'90)*, Vol. 24, No. 4, pp. 243–252, Aug. 1990.
44. J. Platt and A. H. Barr, "Constraint methods for flexible models," *ACM Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 4, pp. 279–288, Aug. 1988.
45. A. Witkin and M. Kass, "Spacetime constraints," *ACM Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 4, pp. 159–168, Aug. 1988.
46. N. I. Badler, K. H. Manoochehri, and G. Walters, "Articulated figure positioning by multiple constraints," *IEEE Computer Graphics and Applications*, Vol. 7, No. 6, pp. 39–51, Nov. 1987.
47. A. Witkin, K. Fleischer, and A. H. Barr, "Energy constraints on parameterized models," *ACM Computer Graphics (Proc. SIGGRAPH'87)*, Vol. 21, No. 4, pp. 225–232, Jul. 1987.
48. R. Barzel and A. H. Barr, "A modeling system based on dynamic constraints," *ACM Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 4, pp. 179–188, Aug. 1988.
49. D. Terzopoulos, J. Platt, A. H. Barr, and K. Fleischer, "Elastically deformable models," *ACM Computer Graphics (Proc. SIGGRAPH'87)*, Vol. 21, No. 4, pp. 205–214, Jul. 1987.
50. D. Terzopoulos and A. Witkin, "Physically based models with rigid and deformable components," *IEEE Computer Graphics and Applications*, Vol. 8, No. 6, pp. 41–51, Nov. 1988.
51. M. P. Do Carmo, *Differential Geometry of Curves and Surfaces*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
52. A. Pentland and J. Williams, "Good vibrations: Modal dynamics for graphics and animation," *ACM Computer Graphics (Proc. SIGGRAPH'89)*, Vol. 23, No. 3, pp. 215–222, Jul. 1989.
53. J. A. Thingvold and E. Cohen, "Physical modeling with b-spline surfaces for interactive design and animation," *ACM Computer Graphics (Proc. SIGGRAPH'90)*, Vol. 24, No. 4, pp. 129–137, Aug. 1990.
54. D. Metaxas and D. Terzopoulos, "Dynamic deformation of solid primitives with constraints," *ACM Computer Graphics (Proc. SIGGRAPH'92)*, Vol. 26, No. 2, pp. 309–312, Jul. 1992.

55. D. Rogers, *Procedural Elements of Computer Graphics (2nd Edition).* Boston, MA: McGraw-Hill, 1997.
56. J. T. Kajiya, "The rendering equation," *ACM Computer Graphics (Proc. SIGGRAPH'86)*, Vol. 20, No. 4, pp. 143–150, 1986.
57. T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, Vol. 23, No. 6, pp. 343–349, 1980.
58. A. Glassner (editor), *An Introduction to Ray Tracing.* Academic Press, 1989.
59. C. Goral, K. Torrance, D. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," *ACM Computer Graphics (Proc. SIGGRAPH'84)*, pp. 213–222, 1984.
60. H. W. Jensen, *Realistic Image Synthesis Using Photon Mapping.* Addison Wesley, 2001.
61. T. Moller, E. Haines, and T. Akenine-Moller, *Real-Time Rendering (2nd Edition).* Natick, MA: A.K. Peters, Ltd., 2002.
62. F. Nicodemus, "Reflectance nomenclature and directional reflectance and emissivity," *Applied Optics*, Vol. 9, pp. 1474–1475, 1970.
63. H. Gouraud, "Continuous shading of curved surfaces," *IEEE Trans. on Computers*, Vol. C-20, No. 6, pp. 623–629, Jun. 1971.
64. B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, Vol. 18, No. 6, pp. 311–317, 1975.
65. J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *Communications of the ACM*, Vol. 19, No. 10, pp. 542–547, 1976.
66. P. Heckbert, "Survey of texture mapping," *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, pp. 56–67, Nov. 1986.
67. N. Greene, "Environment mapping and other applications of world projections," *IEEE Computer Graphics and Applications*, Vol. 6, No. 11, pp. 21–29, 1986.
68. J. F. Blinn, "Simulation of wrinkled surfaces," *ACM Computer Graphics (Proc. SIGGRAPH'78)*, Vol. 12, No. 3, pp. 286–292, Aug. 1978.
69. A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated ray-tracing system," *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, pp. 16–26, 1986.
70. B. Pradhan and A. Mukhopadhyay, "Adaptive cell division for ray tracing," *Computers & Graphics*, Vol. 15, No. 4, pp. 549–552, 1991.
71. R. A. Hall and D. P. Greenberg, "A testbed for realistic image synthesis," *IEEE Computer Graphics and Applications*, Vol. 3, No. 8, pp. 10–99, 1983.
72. H. Weghorst, G. Hooper, and D. P. Greenberg, "Improved computational methods for ray tracing," *ACM Trans. on Graphics*, Vol. 3, No. 1, pp. 52–69, 1984.
73. L. Cook, T. Porter, and L. Carpenter, "Distributed raytracing," *ACM Computer Graphics (Proc. SIGGRAPH'84)*, pp. 137–145, 1984.
74. I. Ashdown, *Radiosity: A Programmer's Perspective.* John Wiley & Sons, 1994.
75. A. Watt and M. Watt, *Advanced Animation and Rendering Techniques.* Addison-Wesley, 1992.
76. M. F. Cohen and D. P. Greenberg, "The hemicube: A radiosity solution for complex environments," *ACM Computer Graphics (Proc. SIGGRAPH'85)*, Vol. 19, No. 3, pp. 31–40, 1985.
77. M. F. Cohen, E. C. Chen, J. R. Wallace, and D. P. Greenberg, "A progressive refinement approach to fast radiosity image generation," *ACM Computer Graphics (Proc. SIGGRAPH'88)*, Vol. 22, No. 4, pp. 75–84, 1988.
78. P. Hanrahan, D. Salzman, and L. Aupperle, "A rapid hierarchical radiosity algorithm," *ACM Computer Graphics (Proc. of SIGGRAPH'91)*, Vol. 25, No. 4, pp. 197–206, 1991.

79. J. R. Wallace, M. F. Cohen, and D. P. Greenberg, "A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods," *ACM Computer Graphics (Proc. SIGGRAPH'87)*, Vol. 21, No. 4, pp. 311–320, 1987.
80. M. Levoy and P. Hanrahan, "Light field rendering," *ACM Computer Graphics (Proc. SIGGRAPH'96)*, pp. 31–42, 1996.
81. S. Chen, "Quicktime VR – An image-based approach to virtual environment navigation," *ACM Computer Graphics (Proc. of SIGGRAPH'95)*, pp. 29–38, 1995.
82. S. Chen and L. Williams, "View interpolation for image synthesis," *ACM Computer Graphics (Proc. of SIGGRAPH'93)*, pp. 279–288, 1993.
83. J. Shade, S. Gortler, L.-W. He, and R. Szeliski, "Layered depth images," *ACM Computer Graphics (Proc. SIGGRAPH'98)*, pp. 231–242, 1998.
84. E. Adelson and J. R. Bergen, "The plenoptic function and the elements of early vision," *Computational Models of Visual Processing.* Cambridge, MA: MIT Press, 1991.
85. L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," *ACM Computer Graphics (Proc. SIGGRAPH'95)*, pp. 39–46, 1995.
86. S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The Lumigraph," *ACM Computer Graphics (Proc. SIGGRAPH'96)*, pp. 43–54, 1996.
87. V. Popescu, "Forward rasterization: A reconstruction algorithm for image-based rendering," Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 2001.
88. S. B. Kang and H.-Y. Shum, "A review of image-based rendering techniqes," in *Proceedings of IEEE/SPIE Visual Communications and Image Processing (VCIP)*, pp. 2–13, 2000.
89. W. Lorensen and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *ACM Computer Graphics (Proc. SIGGRAPH'87)*, Vol. 21, No. 4, pp. 163–169, Jul. 1987.
90. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The VolumePro real-time ray-casting system," *ACM Computer Graphics (Proc. SIGGRAPH'99)*, Vol. 33, pp. 245–250, 1999.
91. H. Berk, C. Aykanat, and U. Güdükbay, "Direct volume rendering of unstructured grids," *Computers & Graphics*, Vol. 27, No. 3, pp. 387–406, 2003.
92. R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," *ACM Computer Graphics (Proc. of SIGGRAPH'98)*, Vol. 32, No. 4, pp. 169–179, 1998.