

1

Rendering

Renderers synthesize images from descriptions of scenes involving geometry, lights, materials and cameras. This chapter explores the image synthesis process, making comparisons with artistic rendering and with real-world cameras.

1.1 Artistic rendering

Using images to communicate is a notion as old as humankind itself. Ancient cave paintings portray scenes of hunts. Religious paintings depict scenes relating to gods, demons and others. Renaissance artists are credited with inventing perspective, which makes it possible to faithfully represent scene elements with geometric realism. Several modern art movements have succeeded in taking apart and reconfiguring traditional notions of form, light and space to create new types of imagery. Computer graphics, a comparatively new medium, significantly extends image creation capabilities by offering very flexible, powerful tools.

We live in a three-dimensional (3D) world, consisting of 3D space, light and 3D objects. Yet the images of such a 3D world that are created inside our eyes are distinctly two-dimensional (2D). Our brains of course are responsible for interpreting the images (from both eyes) and recreating the three-dimensionality for us. A film camera or movie camera does something similar, which is to form 2D images of a 3D world. Artists often use the term “rendering” to mean the representation of objects or scenes on a flat surface such as a canvas or a sheet of paper.

Figure 1.1 shows images of a torus (donut shape) rendered with sketch pencil (a), colored pencils (b), watercolor (c) and acrylic (d).

Each medium has its own techniques (e.g. the pencil rendering is done with stippling, the color pencil drawing uses cross-hatch strokes while the watercolor render uses overlapping washes) but in all cases the result is the same – a 3D object is represented on a 2D picture plane. Artists have an enormous flexibility with media, processes, design, composition, perspective, color and value choices, etc. in rendering their scenes. Indeed, many artists eventually develop their own signature rendering style by experimenting with portraying their subject matter in a variety of media using different techniques. A computer graphical renderer is really one more tool/medium, with its own vocabulary of techniques for representing 3D worlds (“scenes”) as 2D digital imagery.

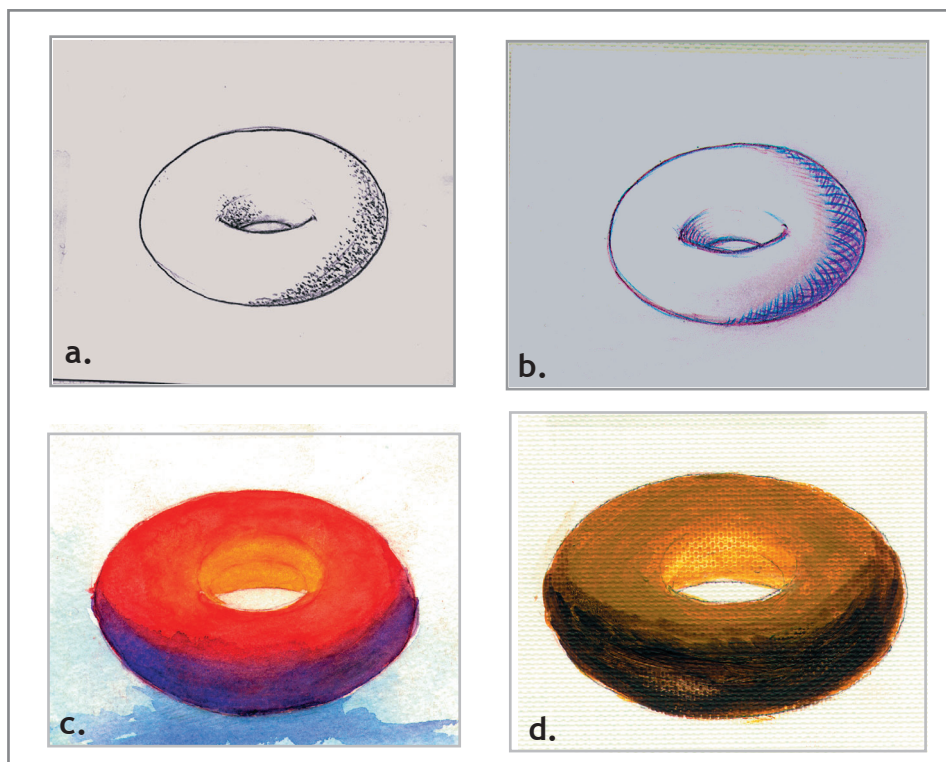


Figure 1.1 Different “renderings” of a torus (donut shape)

1.2 Computer graphical image synthesis

Computers can be used to create digital static and moving imagery in a variety of ways. For instance, scanners, digital still and video cameras serve to capture real-world images and scenes. We can also use drawing and painting software to create imagery from scratch, or to manipulate existing images. Video editing software can be used for trimming and sequencing digital movie clips and for overlaying titles and audio. Clips or individual images can be layered over real-world or synthetic backgrounds, elements from one image can be inserted into another, etc. Digital images can indeed be combined in seemingly endless ways to create new visual content.

There is yet another way to create digital imagery, which will be our focus in this book. I am of course referring to computer graphics (CG) rendering, where descriptions of 3D worlds get converted to images. A couple of comparisons will help make this more concrete. Figure 1.2 illustrates this discussion.

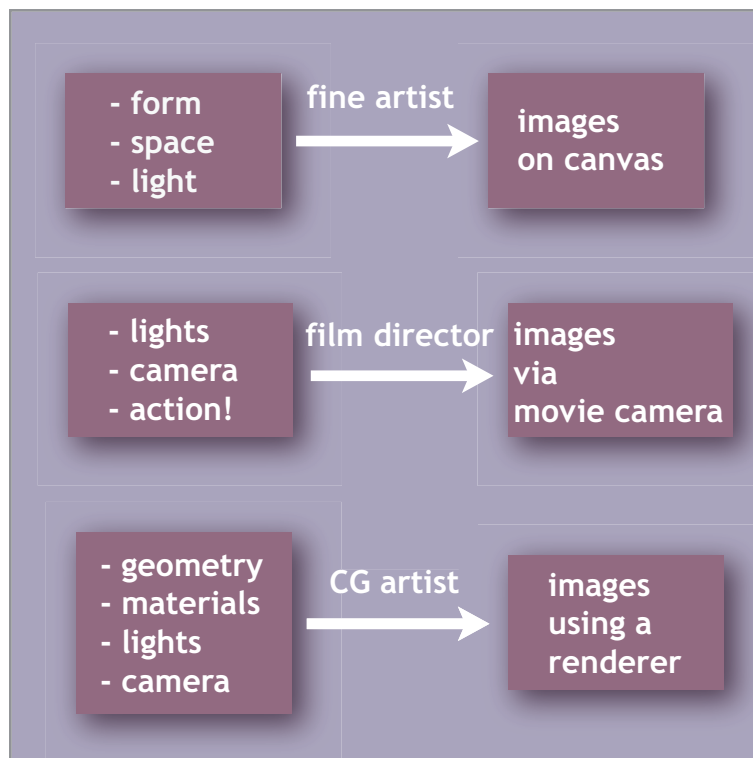


Figure 1.2 *Three routes to image synthesis*

Think of how you as an artist would render a scene in front of you. Imagine that you would like to paint a pretty landscape, using oil on canvas. You intuitively form a scene description of the things that you are looking at, and use creativity, judgment and technique to paint what you want to portray onto the flat surface. You are the renderer that takes the scene description and eventually turns it into an image. Depending on your style, you might make a fairly photorealistic portrait which might make viewers feel as if they are there with you looking at the landscape. At the other extreme you might produce a very abstract image, using elements from the landscape merely as a guide to create your own shapes, colors and placement on canvas. Sorry if I make the artistic process seem mechanical – it does help serve as an analogy to a CG renderer.

A photographer likewise uses a camera to create flat imagery. The camera acts as the renderer, and the photographer creates a scene description for it by choosing composition, lighting and viewpoint.

On a movie set, the classic “Lights, camera, action!” call gets the movie camera to start recording a scene, set up in accordance with a shooting script. The script is interpreted by the movie’s Director, who dictates the choice and placement of lights, camera(s) and actors/props in the scene. As the actors “animate” while delivering dialog, the movie camera renders the resulting scene to motion picture film or digital output media. The Director sets up the scene description and the camera renders it.

In all these cases, scene descriptions get turned into imagery. This is just what a CG renderer does. The scene is purely synthetic, in the sense that it exists only inside the machine. The renderer’s output (rendered image) is equally synthetic, being a collection of

colored pixels which the renderer calculates for us. We look at the rendered result and are able to reconstruct the synthetic 3D scene in our minds. This in itself is nothing short of wonderful – we can get a machine to synthesize images for us, which is a bigger deal than merely having it record or process them.

Let us look at this idea of scene description a bit closer. Take a look at Figure 1.3. and imagine creating a file by typing up the description shown using a simple text editor. We would like the renderer to create us a picture of a red sphere sitting on a blue ground plane. We create this file which serves as our scene description, and pass it on to our renderer to synthesize an image corresponding to the description of our very simple scene.

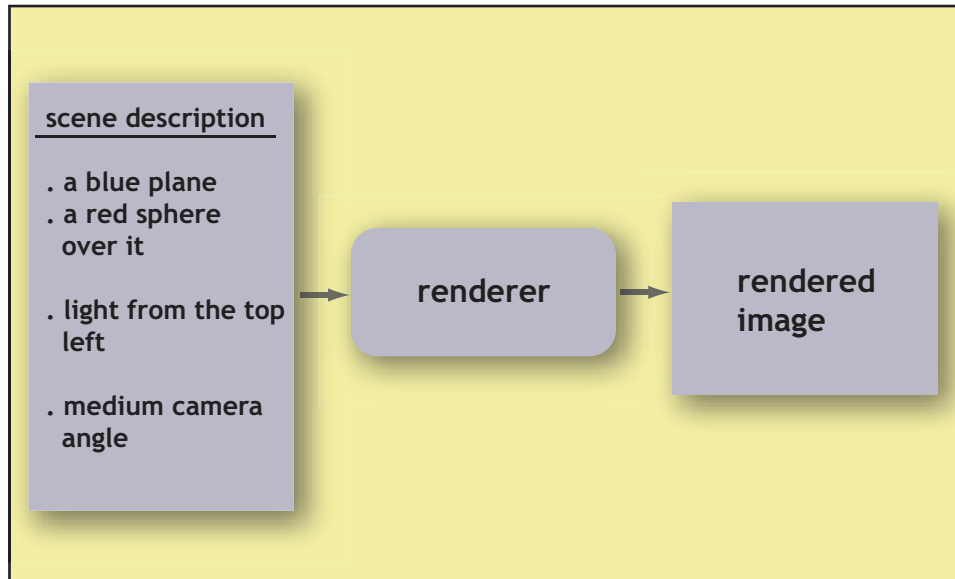


Figure 1.3 *A renderer being fed a scene description*

The renderer parses (reads, in layperson's terms) the scene file, carries out the instructions it contains, and produces an image as a result. So this is the one-line summary of the CG rendering process – 3D scene descriptions get turned into images.

That is how RenderMan, the renderer we are exploring in this book, works. It takes scene description files called **RIB** files (much more on this in subsequent chapters 3 to 8) and creates imagery out of them. **RIB** stands for **RenderMan Interface Bytestream**. For our purposes in this book, it can be thought of as a language for describing scenes to RenderMan. Figure 1.4 shows the **RIB** version of our simple red sphere/blue plane scene, which RenderMan accepts in order to produce output image shown on the right.

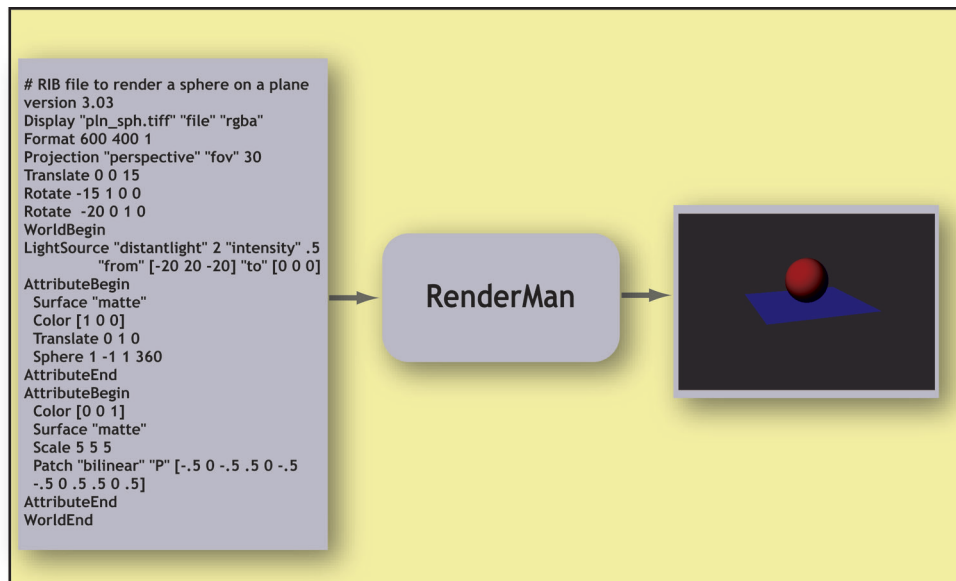


Figure 1.4 *RenderMan converts RIB inputs into images*

You can see that the RIB file contains concrete specifications for what we want RenderMan to do. For example “Color [1 0 0]” specifies red color for the sphere (in RGB color space). The RIB file shown produces the image shown. If we made derivative versions of RIB files from the one above (e.g. by changing the “Translate 0 0 15” to “Translate 0 0 18”, then to “Translate 0 0 21” and so on, which would pull the camera back from the scene each step, and by changing the “pln_sph.tiff” to “pln_sph2.tiff”, then to “pln_sph3.tiff”, etc. to specify a new image file name each time), RenderMan will be able to read each RIB file and convert it to an image named in that RIB file. When we play back the images rapidly, we will see an animation of the scene where the light and two objects are static, and the camera is being pulled back (as in a dolly move – see Chapter 6, “Camera, output”). The point is that a movie camera takes near-continuous snapshots (at 24 frames-per-second, 30 frames-per-second, etc.) of the continuous scene it views, while a CG renderer is presented scene snapshots in the form of a scene description file, one file per frame of rendered animation. Persistence of vision in our brains is what causes the illusion of movement in both cases, when we play back the movie camera’s output as well as a CG renderer’s output.

1.3 Representational styles

With the eye/camera/CG renderer analogy in mind, it is time to look at the different ways that renderers can render scene descriptions for us.

For the most part, we humans visually interpret the physical world in front of us fairly identically. The same is generally true for cameras, aside from differences in lenses and film/sensor type. Their inputs come from the real world, get processed through optical elements based on physical and geometric laws, leading to image formation on physical media. But this is not how CG renderers work. As you know by now, their inputs are scene descriptions. They turn these scene descriptions into imagery, via calculations embodied in rendering algorithms (recipes or procedures) for image synthesis. The output images are

really grids of numbers that represent colors. Of course we eventually have to view the outputs on physical devices such as monitors, printers and film-recorders.

Because rendered images are calculated, depending on the calculations, the same input scene description can result in a variety of output representations from the renderer. Each has its use. We will now take a look at several of the most common rendering styles in use. Each shows a different way to represent a 3D surface. By 3D we do not mean stereo-viewing, rather we mean that such a surface would exist as an object in the real world, something you can hold in your hands, walk around, see it be obscured by other objects.

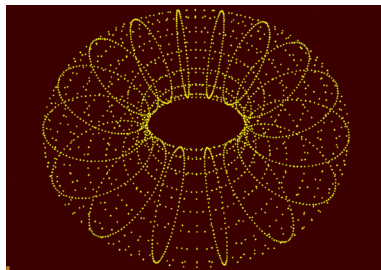


Figure 1.5 *Point-cloud representation of a 3D surface*

Figure 1.5 shows a point-cloud representation of a torus. Here, the image is made up of just the vertices of the polygonal mesh that makes up the torus (or of the control vertices, in the case of a patch-based torus). We will explore polygonal meshes and patch surfaces in detail, in Chapter 4. The idea here is that we infer the shape of a 3D object by mentally connecting the dots in its point cloud image. Our brains create in our mind's eye, the surfaces on which the dots lie. In terms of Gestalt theories, the law of continuation (where objects arranged in straight lines or curves are perceived as a unit) and the principle of closure (where groups of objects complete a pattern) are at work during the mental image formation process.

Next is a wireframe representation, shown in Figure 1.6. As the name implies, this type of image shows the scaffolding wires that might be used to fashion an object while creating a sculpture of it. While the torus is easy to make out (due to its simplicity of shape and sparseness of the wires), note that the eagle mesh is too complex for a small image in wireframe mode. Wireframe images are rather easy for the renderer to create, in comparison with the richer representations that follow. In wireframe mode the renderer is able to keep up with scene changes in real time, if the CG camera moves around an object or if the object is translated/rotated/scaled. The wireframe style is hence a common preview mode when a scene is being set up for full-blown (more complex) rendering later.

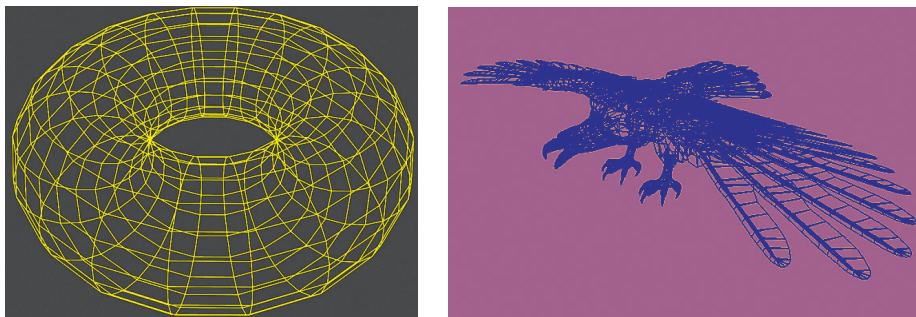


Figure 1.6 *Wireframe view*

A hidden line representation (Figure 1.7) is an improvement over a wireframe view, since the renderer now hides those wires in the wireframe that would not be visible as if they were obscured by parts of the surface near to the viewer. In other words, if black opaque material were to be used over the scaffolding to form a surface, the front parts of that surface would hide the wires and the back parts behind it. The result is a clearer view of the surface, although it is still in scaffolding-only form.

A step up is a hidden line view combined with depth cueing, shown in Figure 1.8. The idea is to fade away the visible lines that are farther away, while keeping the nearer lines in contrast. The resulting image imparts more information (about relative depths) compared to a standard hidden line render. Depth cueing can be likened to atmospheric perspective, a technique used by artists to indicate far away objects in a landscape, where desaturation is combined with a shift towards blue/purple hues to fade away details in the distance.



Figure 1.7 Hidden-line view - note the apparent reduction in mesh density

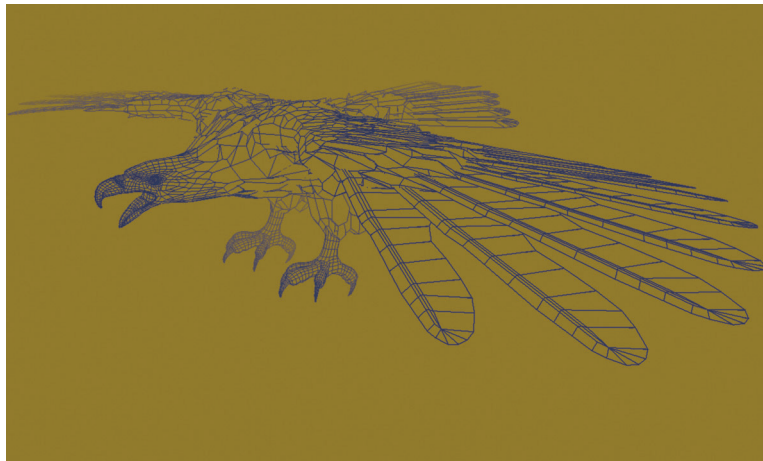


Figure 1.8 Hidden line with depth cue

A bounding box view (Figure 1.9, right image) of an object indirectly represents it by depicting the smallest cuboidal box that will just enclose it. Such a simplified view might be useful in previewing composition in a scene that has a lot of very complex objects, since bounding boxes are even easier for the renderer to draw than wireframes. Note that an alternative to a bounding box is a bounding sphere, but that is rarely used in renderers to convey extents of objects (it is more useful in performing calculations to decide if objects inter-penetrate).

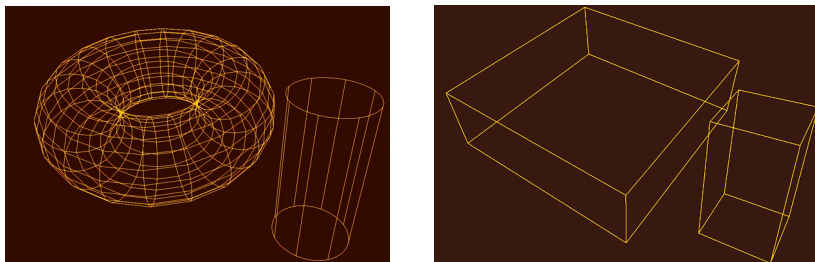


Figure 1.9 *Bounding box view of objects*

We have so far looked at views that impart information about a surface but do not really show all of it. Views presented from here on show the surfaces themselves. Figure 1.10 is a flat shaded view of a torus. The torus is made up of rectangular polygons, and in this view, each polygon is shown rendered with a single color that stretches across its area. The shading for each polygon is derived with reference to a light source and the polygon's orientation relative to it (more on this in the next section). The faceted result serves to indicate how the 3D polygonal object is put together (for instance we notice that the polygons get smaller in size as we move from the outer rim towards the inner surface of the torus). As with depth-cueing discussed earlier, the choice of representational style determines the type of information that can be gleaned about the surface.

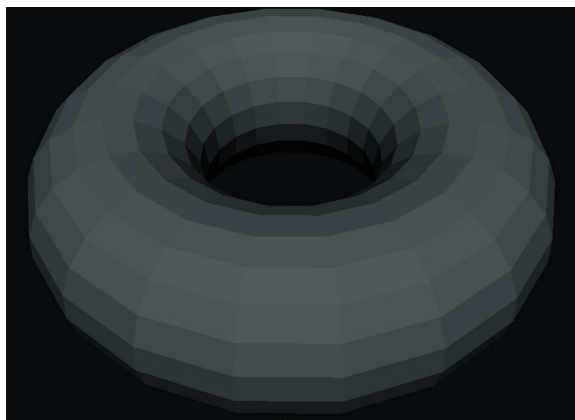


Figure 1.10 *Flat shaded view of a torus*

Smooth shading is an improvement over the flat look in Figure 1.10. It is illustrated in Figure 1.11, where the torus polygonal mesh now looks visually smoother, thanks to a better shading technique. There are actually two smooth shading techniques for polygonal meshes, called Gouraud shading and Phong shading. Of these two, Gouraud shading is easier for a renderer to calculate, but Phong shading produces a smoother look, especially where the

surface displays a highlight (also known as a hot spot or specular reflection). We will discuss the notion of shading in more detail later, in Chapter 8. For a sneak preview, look at Figure 8.33 which compares flat, Gouraud and Phong shading. On a historic note, Henri Gouraud invented the Gouraud shading technique in 1971, and Bui Tui Phong came up with Phong shading a few years later, in 1975. Both were affiliated with the computer science department at the University of Utah, a powerhouse of early CG research.

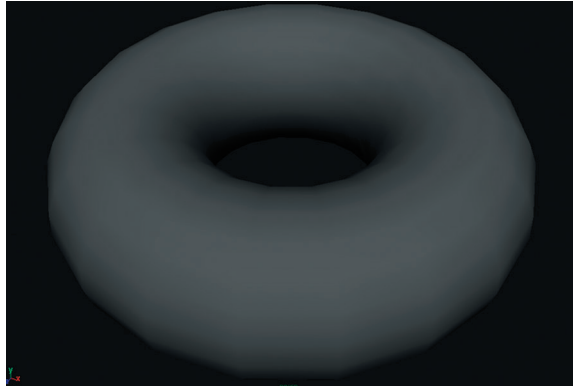


Figure 1.11 *Smooth shaded view*

A hybrid representational style of a wireframe superimposed over a shaded surface is shown in Figure 1.12. This is a nice view if you want to see the shaded form of an object as well as its skeletal/structural detail at the same time.

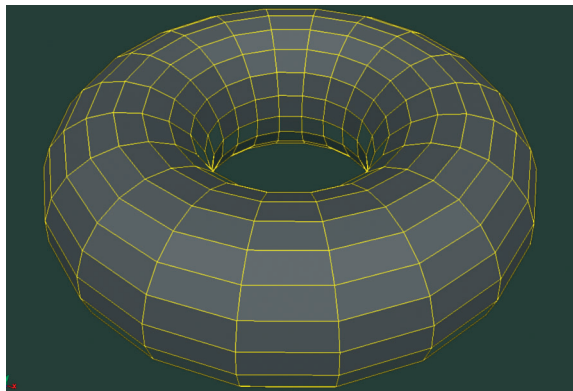


Figure 1.12 *Wireframe over smooth shading*

Also popular is an x-ray render view where the object is rendered as if it were partly transparent, allowing us to see through the front surfaces at what is behind (Figure 1.13). By the way, the teapot shown in the figure is the famous “Utah Teapot”, a classic icon of 3D graphics. It was first created by Martin Newell at the University of Utah. You will encounter this teapot at several places throughout the book.

Until now we have not said anything about materials that make up our surfaces. We have only rendered dull (non-reflective, matte) surfaces using generic, gray shades. Look around you at the variety of surfaces that make up real-world objects. Objects have very many properties (e.g. mass, conductivity, toughness) but for rendering purposes, we concentrate

on how they interact with light. Chapter 8 goes into great detail about this, but for now we will just note that CG surfaces get associated with materials which specify optical properties for them, such as their inherent color and opacity, how much diffuse light they scatter, how reflective they are, etc. When a renderer calculates an image of an object, it usually takes these optical properties into account while calculating its color and transparency (this is the shading part of the rendering computation – see the next section for more details).

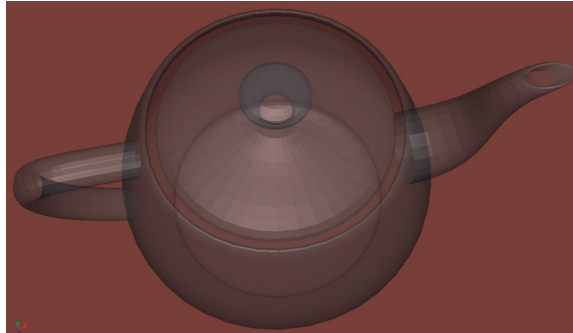


Figure 1.13 *An x-ray view of the famous Utah teapot*

Figure 1.14 shows a lit view of the teapot meant to be made of a shiny material such as metal or plastic. The image is rendered as if there were two light sources shining on the surface, one behind each side of the camera. You can deduce this by noticing where the shiny highlights are. Inferring locations and types of light sources by looking at highlights and shaded/shadowed regions in any image is an extremely useful skill to develop in CG rendering. It will help you light CG scenes realistically (if that is the goal) and to match real-world lights in filmed footage, when you are asked to render CG elements (characters/props) for seamless integration into the footage.



Figure 1.14 *Teapot in “lit” mode*

Figure 1.15 shows the teapot in a lit, textured view. The object, which appears to be made of marble, is illuminated using a light source placed at the top left. The renderer can generate the marble pattern on the surface in a few different ways. We could photograph flat marble slabs and instruct the renderer to wrap the flat images over the curved surface during shading calculations, in a process known as texture mapping. Alternately we could use a 3D paint program (in contrast to the usual 2D ones such as Photoshop) where we can directly paint the texture pattern over the surface, and have the renderer use that while shading. Or we could write a small shader program which will mathematically compute the

marble pattern at each piece of the teapot, associate that shader program with the teapot surface, and instruct the renderer to use the program while shading the teapot. The last approach is called procedural shading, where we calculate (synthesize) patterns over a surface. This is the approach I took to generate the figure you see. RenderMan is famous for providing a flexible, powerful, fun shading language which can be used by artists/software developers to create a plethora of appearances. Chapter 8 is devoted exclusively to shading and shader-writing.

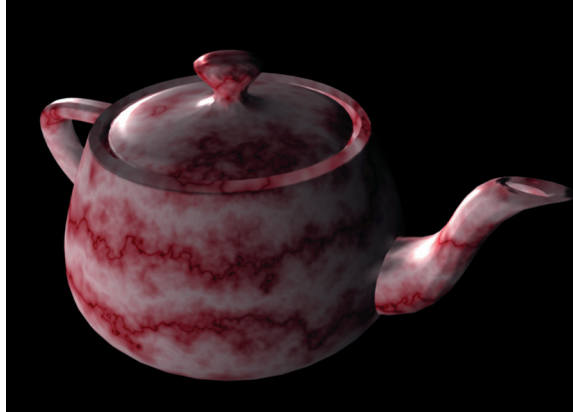


Figure 1.15 *Teapot shown lit and with a marble texture*

Are we done with cataloging rendering representations? Not quite. Here are some more. Figure 1.16 is a cutout view of the eagle we encountered before, totally devoid of shading. The outline tells us it is an eagle in flight, but we are unable to make out any surface detail such as texture, how the surfaces curve, etc. An image like this can be turned into a matte channel (or alpha channel), which along with a corresponding lit, shaded view can be used for example to insert the eagle into a photograph of a mountain and skies.



Figure 1.16 *Cutout view showing a silhouette of the object*

Since a renderer calculates its output image, it can turn non-visual information into images, just as well as it can do physically accurate shading calculations using materials and light sources. For instance, Figure 1.17 depicts a z-depth image where the distance of each visible surface point from the camera location has been encoded as a black to white scale. Points farthest from the camera (e.g. the teapot's handle) are dark, and the closest parts (the spout)

are brighter. People would find it very difficult to interpret the world in terms of such depth images, but for a renderer, it is rather routine, since everything is calculated instead of being presented merely for recording. Depth images are crucial for a class of shadow calculations, as we will see in Chapter 8 (“Shading”).



Figure 1.17 *A z-depth view of our teapot*

Moving along, Figure 1.18 shows a toon style of rendering a torus. Cartoons, whether in comic book (static images) or animated form, have been a very popular artistic rendering style for many decades. A relatively new development is to use 3D renderers to toon-render scenes. The obvious advantage in animation is that the artist is spared the tedium of having to painstakingly draw and paint each individual image – once the 3D scene is set up with character animation, lights, props, effects and camera motion, the renderer can render the collection of frames in toon style, eliminating the drawing and painting process altogether. In practice this has advantages as well as drawbacks. Currently the biggest drawback seems to be that the toon lines do not have a lively quality that is present in the frame-by-frame hand-generated results – they are a bit too perfect and come across as being mechanical, dull and hence lifeless. Note that the toon style of rendering is the 3D equivalent of posterization, a staple in 2D graphic design. Posterization depicts elements using relatively few, flat tones in favor of more colors that depict continuous, smooth shading. In both toon rendering and posterization, form is suggested using a well-chosen, small palette of tones which fill bold, simple shapes.

Improving toon rendered imagery is an area of ongoing research that is part of an even bigger umbrella of graphics research called non-photoreal rendering. Non-photoreal rendering (NPR for short) aims to move CG rendering away from its traditional roots (see Section 1.5) and steer it towards visually diverse, artistic representational styles (as opposed to photoreal ones).

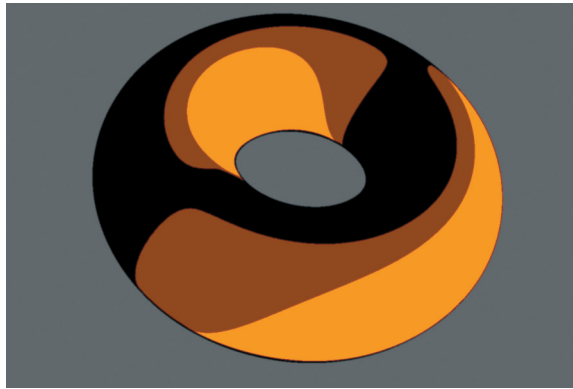


Figure 1.18 *Non-photoreal “toon” style rendering*

Here is our final sample of representations. Figure 1.19 shows the teapot again, this time with a process called displacement mapping. The surface appears made out of hammered sheet metal.

What gives it the hammered look? The imperfections on the surface do. The knob on the lid and the rim of the body in particular show that the surface is indeed deformed. But what is interesting is that the same Utah teapot used in previous illustrations was the one used here also. In other words, the object surface itself was not remodeled with imperfections, prior to rendering. What causes the realistic displacements is a displacement shader (a small piece of software), which together with a another surface shader for the metallic appearance was associated with the teapot surface and was input to RenderMan via a RIB file. RenderMan carried out the local surface modifications (displacements) by consulting the associated shader program, during rendering. Letting the user specify surface modifications during rendering is a significant capability of RenderMan which we will further explore in Chapter 8, “Shading”.

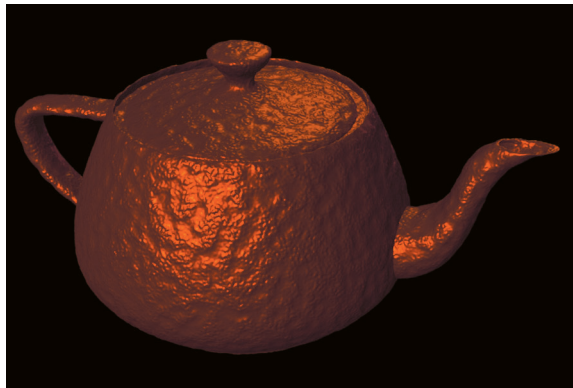


Figure 1.19 *Displacement mapped teapot*

We surveyed many ways a renderer can represent surfaces for us. But the list is not exhaustive. For instance none of the images had objects casting shadows with either hard or soft edges (e.g. as if the teapot were sitting on a plane and lit by a directional light source or maybe a more diffused light). There were no partially transparent objects (the x-ray view was a mere approximation) showing refraction (light-bending) effects. Surfaces did not show

other objects reflecting on them, nor did they bleed color on to neighboring surfaces. The point is that real-world surfaces do all these and even more. Many of the things just mentioned were faked in most renderers for years because the renderers were incapable of calculating them in a physically accurate manner, the way a camera would simply record such effects on film. Modern renderers (including version 11 of Pixar's RenderMan, the latest release at the time of this writing) are increasingly capable of rendering these effects without user-set up cheats, getting ever closer to the ideal of a CG renderer producing photoreal images of almost any scene description fed to them. Note that by definition, photoreal images are meant to be indistinguishable from camera-derived images of the real world. Speaking of photorealism, we know that cameras invariably introduce artifacts into the images they record. These artifacts stem from their physical/mechanical/electronic subsystems interfering with the image recording process. For instance, lens distortions and flares, chromatic aberrations, motion blur, etc. are camera-derived artifacts not present in images we see with our naked eyes. It is interesting that CG renderers are outfitted with additional capabilities that let users synthetically add such imperfections to the otherwise perfect images they render, in order to make those images look that much more photoreal. These imperfections are part of our subconscious cues that make recorded/rendered images appear real to us.

1.4 How a renderer works

Now we take a brief look at how a renderer accomplishes the task of converting a 3D scene description into an image.

As I have been pointing out, the entire process is synthetic, calculation-driven as opposed to what cameras and eyes do naturally. The CG renderer does not “look at” anything in the physical or biological sense. Instead it starts with a scene description, carries out a well-defined set of subtasks collectively referred to as the 3D graphics pipeline or the rendering pipeline, and ends with the generation of an image faithful to the scene described to it.

What follows is an extremely simplified version of what really goes on (renderers are very complex pieces of software where a lot of heavy-duty calculations and book-keeping goes on during the rendering process). In other books you might come across descriptions where some of the presented steps differ (this is especially true for the first part of the pipeline) but the following description will give you a feel for what goes on “under the hood” during rendering.

Figure 1.20 shows the pipeline as a block diagram where data flows from top to bottom. In other words the scene description enters at the top (this is the input), and the rendered image is produced in the bottom as the output.

The scene description consists of surfaces made of materials spatially composed, lit by light sources. The virtual camera is also placed somewhere in the scene, for the renderer to look through, to create a render of the scene from that specific viewpoint. Optionally the atmosphere (the space in which the scene is set) itself might contribute to the final image (e.g. think of looking at an object across a smoke-filled room) but we ignore this to keep our discussion simple. The renderer must now convert such a scene description into a rendered image.

The first few steps in the pipeline are space transformations. What we mean is this. Figure 1.21 shows two images, those of a cube and a cylinder. Each object is shown modeled in its own, native object space. For a cube, this space consists of the origin in the middle of the volume, and the mutually-perpendicular X, Y and Z axes are parallel to the edges of the cube. Cubes can be modeled/specified using alternate axes (e.g. one where the origin is in

one corner and the Z axis is along the volume diagonal) but what is shown above is the most common specification. The cylinder is usually modeled with respect to an origin in its center, with the Z axis pointing along the length or height. The surface description consists of parameters that describe the shape in relation to the axes. For our polygonal cube for instance, a description would consist of the spatial (x, y, z) locations of its eight vertices (e.g. -0.5, -0.5, -0.5 for one corner and 0.5, 0.5, 0.5 for the opposite corner) and vertex indices that make up the six faces (e.g. "1 2 3 4" making up a face). Loosely speaking, edges are the wireframe lines that make up our surfaces, vertices are the corners where edges meet and faces are the flat areas bounded by edges and vertices. Similar to our cube and cylinder, character body parts, props, abstract shapes and other modeled objects carry with them their own object or model axes, and their surfaces are described in relation to those axes.

When objects are introduced into a scene, they are placed in the scene relative to a common origin and set of axes called the world coordinate system. Figure 1.22 shows our cube and cylinder together in a little scene, where they are placed next to each other interpenetrating (to better illustrate some of the following steps).

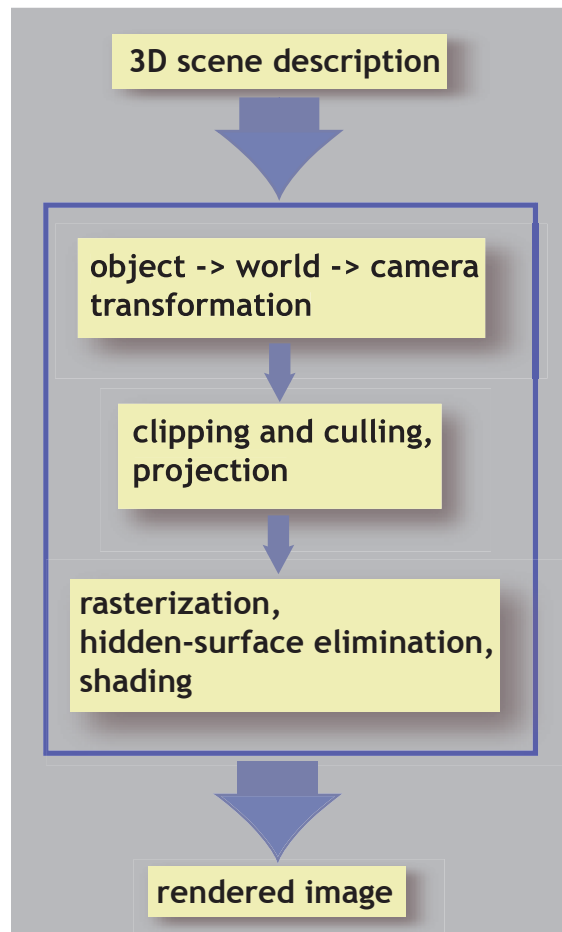


Figure 1.20 *Stages in a classical rendering pipeline*

What is important is that an object placed into a world coordinate system will in general undergo any combination of translation, rotation or scale in order to be placed in its

location. In other words, it is as if our cube and cylinder were placed at the world origin, with their model axes and the world axes initially coincident. From there, each object undergoes its own translation/rotation/scaling (chosen by the user setting up the scene) to end up where it is. This placement process is called object to world transformation. Note from the figure that the scene also contains a camera, which is itself another object which underwent its own translation/rotation/scaling to end up where it is, which in our case is in front of the cube and cylinder.

Since we will be viewing the scene through this camera which is near our objects, the next transformation is to relocate the world origin smack in the middle of the camera and point an axis (usually Z) along the camera's viewing direction. This is called the camera or eye coordinate system. The positions, orientations and sizes of the elements in the rest of the scene, which include both objects and light sources, can now be described in terms of this new coordinate system. The scene elements are said to undergo a world to camera space transformation when they are being described this way in terms of the new set of axes.

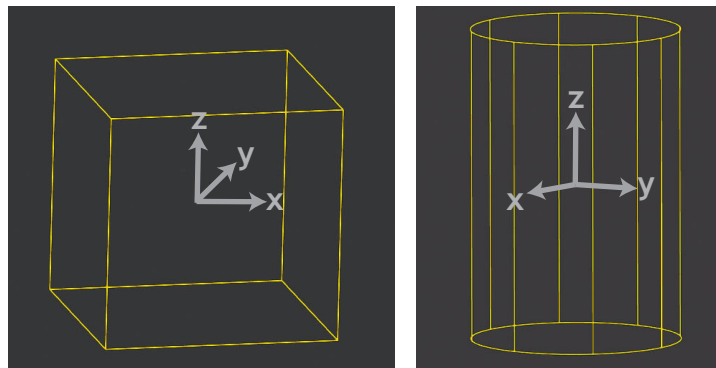


Figure 1.21 Object space views of a cube and cylinder

The view of the scene through our scene camera is shown in Figure 1.23. More accurately, what the camera sees is what is inside the rectangular frame, which is the image area. The reason is that just like animal eyes and physical cameras, CG cameras too have a finite field of view, meaning that their view is restricted to what is in front of them, extending out to the sides by a specified amount (called the view angle or field of view or FOV).

Take a look at Figure 1.22 again which shows the placement of the camera in the world. You see a square pyramid originating from the camera. This is called the view volume, which is the space that the camera can see. The view volume is not unbounded, meaning the camera cannot see things arbitrarily far away. A plane called the far clipping plane which is parallel to the image plane (not tilted with respect to it) defines this farther extent. Similarly, CG cameras will not be able to process surfaces placed very close to them either. So a near clipping plane is placed a short distance away from the camera, to bound the viewing volume from the near side. The result is a view volume in the shape of a truncated pyramid (where the pyramid's apex or tip is cut off by the near clipping plane). This volume is also known as the camera's view frustum.

Now that we have made the camera the center of the scene (since the new world origin is located there) and situated objects relative to it, we need to start processing the scene elements to eventually end up with computed images of them. In the diagram shown in Figure 1.20, this is the middle set of operations, namely clipping, culling, projection.

Figure 1.24 illustrates the result of the clipping operation, where the six planes of our viewing frustum (near and far clipping planes, and the top, bottom, left and right pyramid planes) are used to bound scene elements meant for further processing. Objects completely lying within the volume are left untouched. Objects lying completely outside are totally discarded, since the camera is not supposed to be able to see them. Objects that partially lie inside (and intersect the bounding planes) are clipped, meaning they are subdivided into smaller primitives for the purposes of retaining what is inside and discarding what falls outside. In Figure 1.24, the cylinder's top part is clipped away. Likewise, the cube intersects two bounding planes (the near clipping plane and a side plane) and is therefore clipped against each of them. What is left inside the view frustum can be processed further.

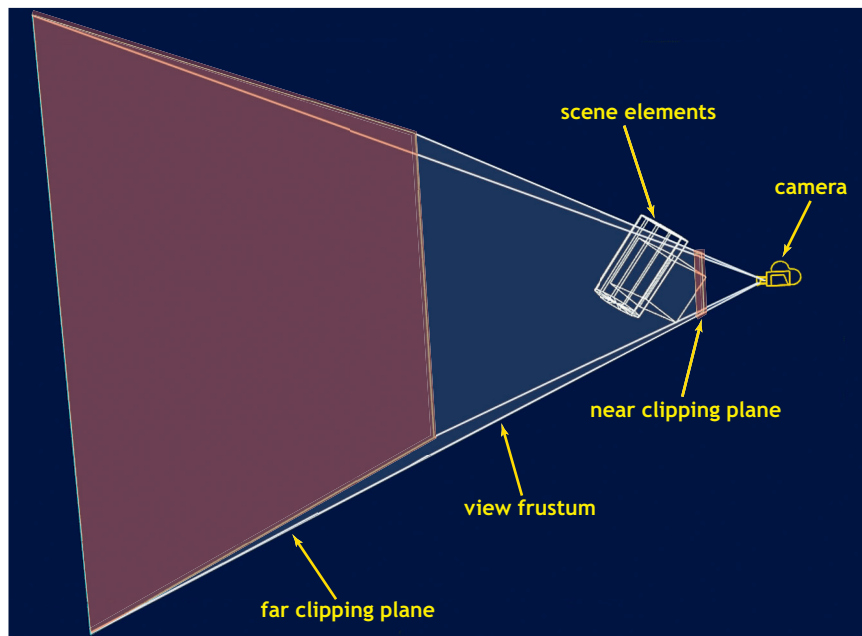


Figure 1.22 World space representation

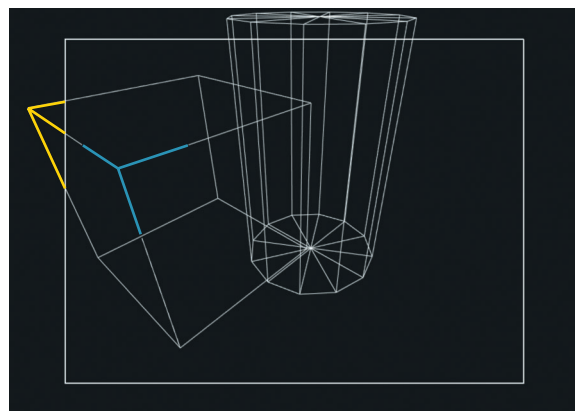


Figure 1.23 Camera view of the scene

The renderer further simplifies surfaces by discarding the flip sides of surfaces hidden by camera-facing front ones, in a step known as back face culling (Figure 1.25). The idea is that each surface has an outside and an inside, and only the outside is supposed to be visible to the camera. So if a surface faces the same general direction that the camera does, its inside surface is the one visible to the renderer, and this is what gets culled when another front-facing surface obscures it. Most renderers permit surfaces to be specified as being two-sided, in which case the renderer will skip the culling step and render all surfaces in the view volume.

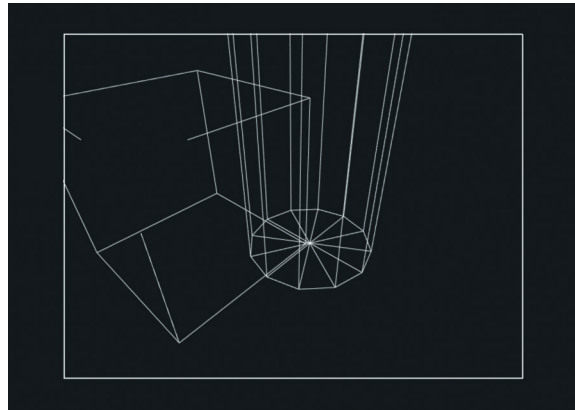


Figure 1.24 Results of the clipping operation

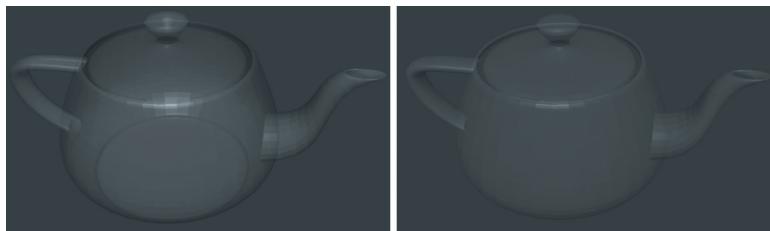


Figure 1.25 Back face culling - before and after

The step that follows is projection, where the remaining 3D surfaces are flattened (projected) on to the image plane. Perspective projection is the most common type of projection employed (there are several other types) to create a sense of realism. We see the effects of perspective around us literally all the time, e.g. when we see the two sides of a straight road converge to a single vanishing point far ahead, look up to see the straight rooflines of buildings slope down to vanishing points, etc. Figure 1.26 shows our cube and cylinder perspective-projected on to the CG camera image plane. Note the three edges shown in yellow converge to a vanishing point (off the image) on the right. Square faces of the cube now appear distorted in the image plane. Similarly the vertical rules on the cylinder appear sloped so as to converge to a point under the image.

Once the surfaces are projected on the image plane, they become 2D entities that no longer retain their 3D shape (although the 2D entities do contain associated details about their unprojected counterparts, to be used in downstream shading calculations). Note that this projection step is where the geometric part of the 3D to 2D conversion occurs – the

resulting 2D surfaces will subsequently be rasterized and shaded to complete the rendering process.

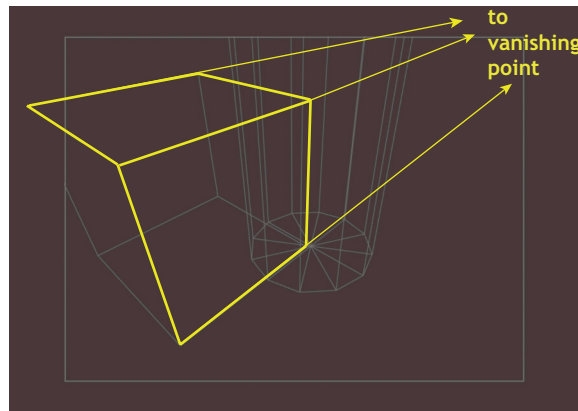


Figure 1.26 *The projection operation*

Before we leave the projection step in the pipeline, I would like to point out that using perspective is not the only way to project surfaces on to image planes. For instance, Figure 1.27 shows an isometric projection, where the cube and cylinder have been tilted towards the camera (from their original object space vertical orientation) and projected in such a way that their XYZ axes make a 120 degree angle on the projection plane (this is particularly evident in the cube). As a result the opposite edges of the cube retain their parallelism after projection, as do the walls of the cylinder. This type of projection is commonly used in engineering drawings of machine parts and in architectural drawings to show buildings, where retaining the parallel lines and planes helps to better visualize the structures (the sloping lines from perspective projection get in the way of our comprehension of shapes and angles).

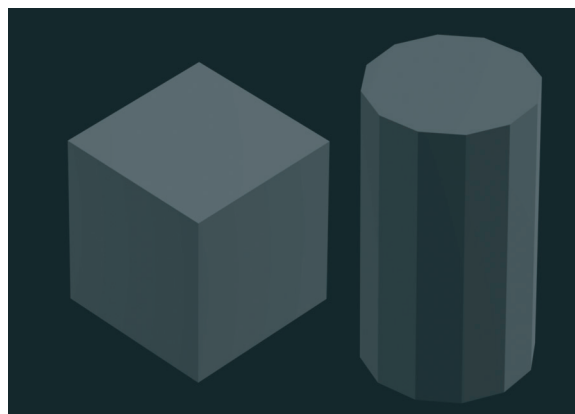


Figure 1.27 *An isometric (non perspective) projection*

We are now at the last stage of our pipeline. Hidden surface elimination, rasterization and shading are steps performed almost simultaneously, but for illustration purposes we will discuss each in sequence. Hidden surface elimination is shown in Figure 1.28. When we are done projecting our surfaces on the image plane, we need to process them in such a way

that surfaces that were in front of other surfaces correspondingly end up in front on the image plane as well. There is more than one technique to ensure this, and we will focus on the most common, successful technique that uses a portion of computer memory called a z-buffer (the “z” stands for the z-axis which points away from the camera and so is the axis along which relative depths of objects in the scene will be measured).

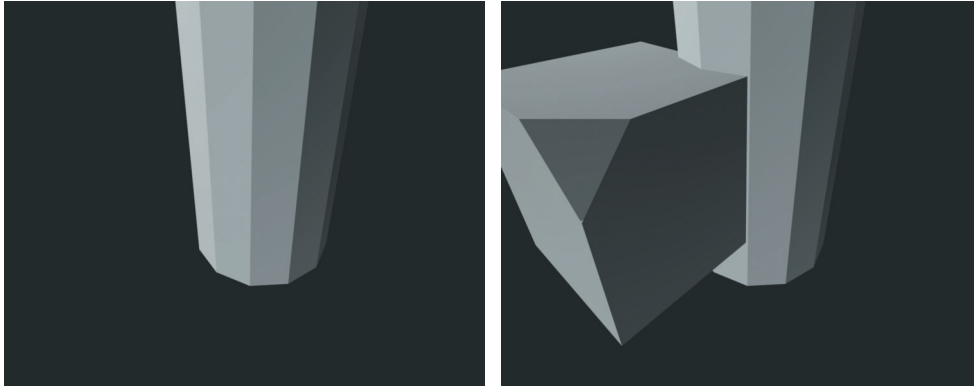


Figure 1.28 *Hidden surface elimination*

Think of a z-buffer as a block of memory laid out like a grid (in other words, a rectangular array of slots) for the renderer to store some intermediate computational results. How many slots? As many pixels as there will be in our final image (pixel count or “resolution” will be one of the inputs to the renderer, specified by the user as part of the scene description). What calculations will the renderer store in the z-buffer? Depth-based comparisons. Before anything is stored in the z-buffer by the renderer, the z-buffer set up process will initialize (fill with values during creation time) each slot to a huge number that signifies an infinite depth from the camera. The renderer then starts overwriting these slots with more realistic (closer) depth values each time it comes across a closer surface which lines up with that slot. So when the cylinder surfaces get processed (we have not processed the cube yet), they all end up in the z-buffer slots, as they are all closer compared to the “very far away from the camera” initial values. After being done with the cylinder, the renderer starts processing the cube surfaces. In doing so it discovers (through depth comparisons) that parts of the cube are closer than parts of the cylinder (where the two overlap), so it overwrites the cylinder surfaces’ depth values with those of the cube’s parts, in those shared z-buffer slots.

When this depth-sorting calculation is done for all the surfaces that are on the image plane, the z-buffer will contain an accurate record of what obscures what. In other words, surfaces that are hidden have become eliminated. Note that the renderer can process these projected surfaces in any arbitrary order and still end up with the same correct result. z-buffer depth sorting serves two inter-related purposes. The renderer does not need to shade (calculate color and opacity for) surface fragments that get obscured by others. In addition, it is important to get this spatial ordering right so the image looks physically correct. Note that the z-buffering just described is for opaque surfaces being hidden by other opaque surfaces, as is the case for the two objects we are considering. The overall idea is still the same for transparent objects but there is more book-keeping to be done, to accurately render such surfaces.

Together with the z-buffer, the renderer also deals with another rectangular array, called the raster (these arrays are all in memory when the renderer performs its tasks). The raster array is what will soon get written out as a collection of pixels that make up our final

rendered result. In other words, doing 3D rendering is yet another way to create a collection of pixels making up a digital image, other ways being using a drawing or paint program, scanning or digitally photographing/filming real-world objects, etc.

Figure 1.29 shows a magnified view of the raster, zoomed in on the cube-cylinder intersection. The rasterization step is where the projected surfaces become “discretized” into collections of pixels. In the figure you can see the individual pixels that go to make up a cube edge, a cylinder edge and the intersection curve between the objects. There is no colorization yet, just some indication of shading based on our light sources in the scene.



Figure 1.29 *Rasterization of surface fragments*

All that is left to do is shading, which is the step where the pixels in the raster are colorized by the renderer. Each pixel can contain surface fragments from zero or one or more surfaces in the scene. Figure 1.30 shows the shaded versions of the pixels in the previous figure. The pixels in the top left are black, a default color used by many renderers to indicate that there are no surfaces there to shade. The cylinder and cube show their yellow and blue edges, and at the intersection, each pixel contains a color that is a blend between the cube and cylinder colors.

Where do these colors come from? As mentioned before, objects in a scene are associated with materials, which are property bundles with information on how those surfaces will react to light. In our example, the cube and cylinder have been set up to be matte (non-glossy) surfaces, with light blue and light yellow-orange body colors respectively. The renderer uses this information, together with locations, colors and intensities of light sources in the scene, to arrive at a color (and transparency or opacity) value for each surface fragment in each pixel. As you might imagine, these have the potential to be lengthy calculations that might result in long rendering times (our very simple example renders in no time at all).

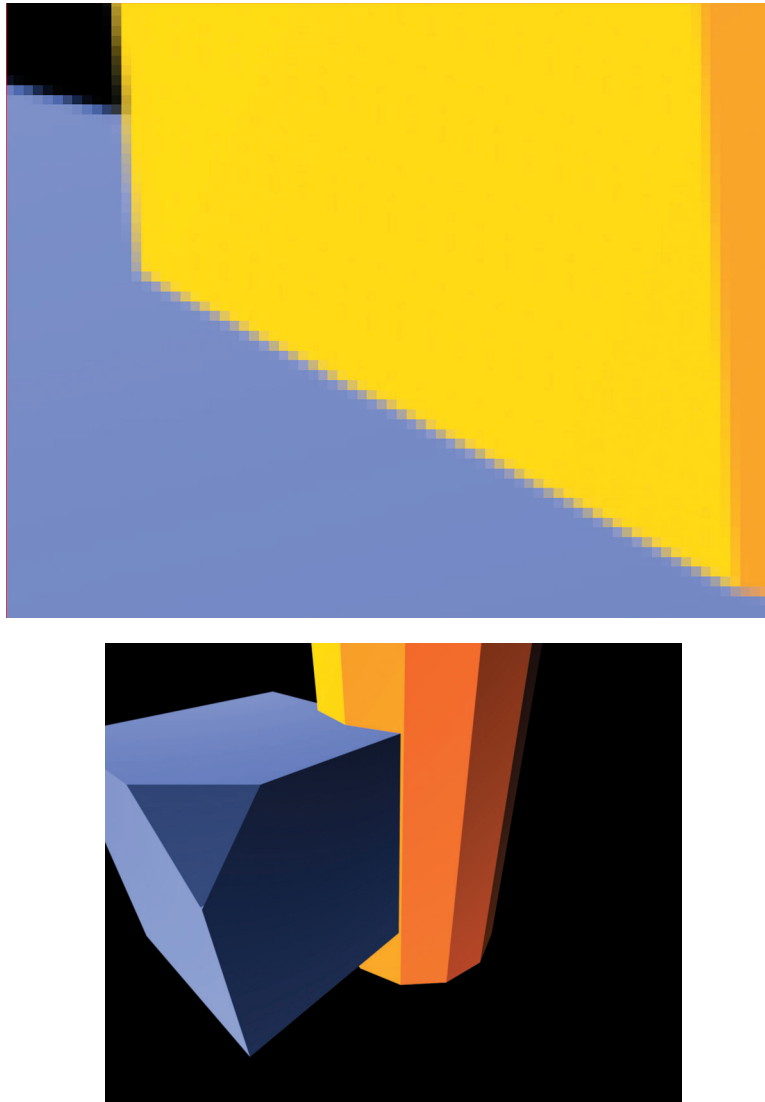


Figure 1.30 *The shading step, and our final result*

Once shading calculations are done, the renderer sends the pixel data (the final result of all the steps we discussed) to the screen for immediate viewing or creates an image file on disk with the data. The choice of destination for the calculated result is inferred by the renderer from the input scene description. Such a rendered result of our basic cube/cylinder scene is shown at the bottom of Figure 1.30.

The pipeline just described is that of a scanline z-buffer renderer, a staple workhorse in modern rendering implementations. While it is good for shading situations where the scene description calls for localized light interactions, it is unsuitable for cases that require global illumination. Chapter 8 (“Shading”) has more on global illumination – here I will just say that the term serves as an umbrella for a variety of rendering algorithms that all aim to capture more realistic interactions between surfaces and lights. For instance, our pipeline

does not address reflections, refractions, color bleeding from one surface to another, transparent shadows (possibly colored), more complex light/surface interactions (e.g. subsurface scattering), certain focussed-light patterns called caustics, etc.

1.5 Origins of 3D graphics, future directions

The beginnings of 3D rendering parallel the evolution of computer graphics itself. Early uses focussed on scientific and technical areas. 3D models of machine parts and architecture helped with design aspects, and later with analysis. Flight simulators were early examples of 3D CG rendered in real time, under user control as they steered. Of course this required very powerful graphics machines, built by companies such as Evans and Sutherland.

The 1970s and 1980s saw an explosion of research in the field, leading to a whole slew of algorithms that are commonly employed today. These include techniques for hidden surface elimination, clipping, texture and reflection mapping, shadow generation, light/surface interaction calculations, ray tracing, radiosity, etc.

In 1981, Silicon Graphics was founded by James Clark, with the goal of creating powerful graphics computers. They were totally successful in their endeavor and launched a new era of visual computing. Their machines featured custom graphics chips called Geometry Engines which accelerated steps in rendering pipelines similar to those described in the previous section. Such hardware acceleration was accessible to programmers via SGI's GL programming library (a set of prebuilt modules). Many common tasks for which computers are used require the machine to mostly pattern match, count, sort and search. These operations do not involve floating point (involving decimals) calculations which are processor intensive. Graphics programs, particularly renderers, do. This rationale for off-loading such heavy calculations to custom-built hardware was recognized early on, so custom-built graphics machines have been part of 3D graphics from the early days.

GL later gave way to OpenGL, a platform independent version of these programming building blocks. Several vendors began manufacturing OpenGL cards that plugged into PCs, for accelerating graphics intensive applications such as video games.

Today we have game platforms such as PlayStation and Xbox which feature incredibly powerful graphics hardware to keep up with the demands of modern game environments.

The evolution of software renderers has its origins in academia. Researchers from the University of Utah pooled their expertise at Pixar to create RenderMan, which has been publicly available since the late 1980s. CG production companies such as Cranston-Csuri, Digital Productions, Omnibus, Abel & Associates, MetroLight Studios, PDI, RezN8 and Rhythm & Hues were formed during the 80s, to generate visual effects for movies and in television, to create commercials, station logos, interstitials, visuals for upcoming programs and other broadcast graphics. In-house software developers in these production companies had to create their own proprietary renderers, since none were available for purchase. Recognizing this need for production quality rendering (and other CG) software, several companies were launched to fill the void. These include Wavefront, Alias, TDI, Softimage, Side Effects, etc. Early versions of RenderMan had been around at Pixar since the mid-80s, and began to be commercially offered as a product in 1989 (much more on RenderMan in the following chapter). Thanks to its stellar image quality, RenderMan quickly found its way into the production pipelines of visual effects houses. Integration of RenderMan with existing software was done by in-house programmers, who wrote custom software to create RIB files out of scene descriptions originating from home-grown modeling and animation software as well as from third-party packages such as Alias.

These days native renderers are built into programs such as Maya, Softimage XSI, Houdini, LightWave, 3D Studio Max, etc. These programs offer integrated modeling, animation, rendering and in some cases, post-production capabilities. Alternatively, renderers such as RenderMan and mental ray are “standalone” in the sense that their scene descriptions come from external sources. Bridging these two extremes are plugins that create on-the-fly scene descriptions out of Maya etc. for consumption by RenderMan, mental ray and other standalones. For instance, MTOR (Maya To RenderMan) and MayaMan are plugins that generate RenderMan RIB descriptions of Maya scenes. Three-dimensional rendering is also increasingly popular on the Internet, e.g. in the form of “talking head” character animation and visually browsable environments (called worlds) built using a specification called VRML (Virtual Reality Modeling Language) and its descendants such as X3D. MPEG-4, another modern specification for video streaming, even has provision for some limited rendering to occur in handheld devices (e.g. for rendering talking head “avatars” or game characters on your cell phone). By the way, MPEG stands for Motion Picture Experts Group and is a standards body responsible for some video formats widely in use today, such as MPEG and DVD-quality MPEG-2. It seems like 3D rendering has escaped its academic and industrial roots, evolving through visual effects CG production into somewhat of a commodity, available in a variety of affordable hardware and software products.

Rendering is an ongoing active graphics research area where advances are constantly being made both in software as well as hardware. Global illumination, which enhances the realism of rendered images by better simulating interactions between surfaces and lights, is one such area of intensive research. Since photorealism is just one target representation of 3D scenes, how about using techniques and styles from the art world for alternate ways to render scenes? Another burgeoning field called painterly rendering or non-photoreal rendering explores this idea in detail. This will open up the possibility of rendering the same input scene in a multitude of styles. On the hardware front, effort is underway to accelerate RenderMan-quality image generation using custom graphics chips. The goal is nothing short of rendering full motion cinematic imagery in real time, on personal computers. This should do wonders for entertainment, education, communication and commerce in the coming years.