Chapter 2
The Graphics Rendering Pipeline

"A chain is no stronger than its weakest link."
—Anonymous

This chapter presents what is considered to be the core component of real-time graphics, namely the *graphics rendering pipeline*, also known simply as the pipeline. The main function of the pipeline is to generate, or *render*, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, shading equations, textures, and more. The rendering pipeline is thus the underlying tool for real-time rendering. The process of using the pipeline is depicted in Figure 2.1. The locations and shapes of the objects in the image are determined by their geometry, the characteristics of the environment, and the placement of the camera in that environment. The appearance of the objects is affected by material properties, light sources, textures, and shading models.

The different stages of the rendering pipeline will now be discussed and explained, with a focus on function and not on implementation. Implementation details are either left for later chapters or are elements over which the programmer has no control. For example, what is important to someone using lines are characteristics such as vertex data formats, colors, and pattern types, and whether, say, depth cueing is available, not whether lines are implemented via Bresenham's line-drawing algorithm [142] or via a symmetric double-step algorithm [1391]. Usually some of these pipeline stages are implemented in non-programmable hardware, which makes it impossible to optimize or improve on the implementation. Details of basic draw and fill algorithms are covered in depth in books such as Rogers [1077]. While we may have little control over some of the underlying hardware, algorithms and coding methods have a significant effect on the speed and quality at which images are produced.

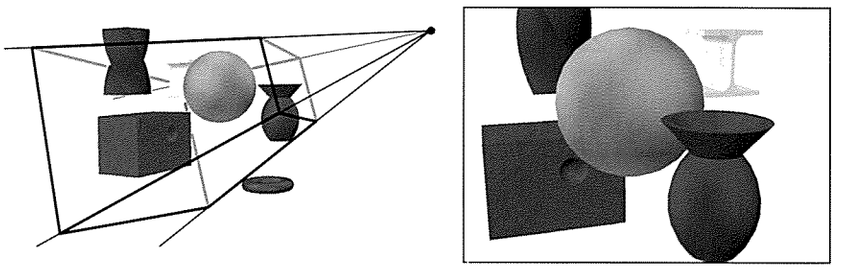

Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid (where four lines converge). Only the primitives inside the view volume are rendered. For an image that is rendered in perspective (as is the case here), the view volume is a frustum, i.e., a truncated pyramid with a rectangular base. The right image shows what the camera "sees." Note that the red donut shape in the left image is not in the rendering to the right because it is located outside the view frustum. Also, the twisted blue prism in the left image is clipped against the top plane of the frustum.

2.1 Architecture

In the physical world, the pipeline concept manifests itself in many different forms, from factory assembly lines to ski lifts. It also applies to graphics rendering.

A pipeline consists of several stages [541]. For example, in an oil pipeline, oil cannot move from the first stage of the pipeline to the second until the oil already in that second stage has moved on to the third stage, and so forth. This implies that the speed of the pipeline is determined by the slowest stage, no matter how fast the other stages may be.

Ideally, a nonpipelined system that is then divided into n pipelined stages could give a speedup of a factor of n. This increase in performance is the main reason to use pipelining. For example, a ski chairlift containing only one chair is inefficient; adding more chairs creates a proportional speedup in the number of skiers brought up the hill. The pipeline stages execute in parallel, but they are stalled until the slowest stage has finished its task. For example, if the steering wheel attachment stage on a car assembly line takes three minutes and every other stage takes two minutes, the best rate that can be achieved is one car made every three minutes; the other stages must be idle for one minute while the steering wheel attachment is completed. For this particular pipeline, the steering wheel stage is the *bottleneck*, since it determines the speed of the entire production.

This kind of pipeline construction is also found in the context of real-time computer graphics. A coarse division of the real-time rendering pipeline into three *conceptual stages*—*application*, *geometry*, and *rasterizer*—is shown in Figure 2.2. This structure is the core—the engine of the rendering

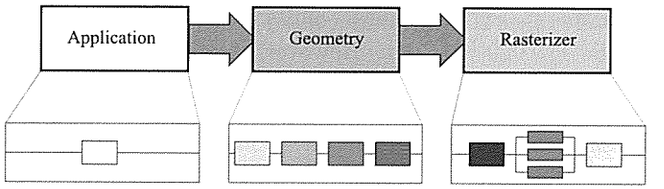

Figure 2.2. The basic construction of the rendering pipeline, consisting of three stages: application, geometry, and the rasterizer. Each of these stages may be a pipeline in itself, as illustrated below the geometry stage, or a stage may be (partly) parallelized, as shown below the rasterizer stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized.

pipeline—which is used in real-time computer graphics applications and is thus an essential base for discussion in subsequent chapters. Each of these stages is usually a pipeline in itself, which means that it consists of several substages. We differentiate between the conceptual stages (application, geometry, and rasterizer), functional stages, and pipeline stages. A functional stage has a certain task to perform but does not specify the way that task is executed in the pipeline. A pipeline stage, on the other hand, is executed simultaneously with all the other pipeline stages. A pipeline stage may also be parallelized in order to meet high performance needs. For example, the geometry stage may be divided into five functional stages, but it is the implementation of a graphics system that determines its division into pipeline stages. A given implementation may combine two functional stages into one pipeline stage, while it divides another, more time-consuming, functional stage into several pipeline stages, or even parallelizes it.

It is the slowest of the pipeline stages that determines the *rendering speed*, the update rate of the images. This speed may be expressed in *frames per second* (fps), that is, the number of images rendered per second. It can also be represented using *Hertz* (Hz), which is simply the notation for $1/seconds$, i.e., the frequency of update. The time used by an application to generate an image usually varies, depending on the complexity of the computations performed during each frame. Frames per second is used to express either the rate for a particular frame, or the average performance over some duration of use. Hertz is used for hardware, such as a display, which is set to a fixed rate. Since we are dealing with a pipeline, it does not suffice to add up the time it takes for all the data we want to render to pass through the entire pipeline. This, of course, is a consequence of the pipeline construction, which allows the stages to execute in parallel. If we could locate the bottleneck, i.e., the slowest stage of the pipeline, and measure how much time it takes data to pass through that stage, then

we could compute the rendering speed. Assume, for example, that the bottleneck stage takes 20 ms (milliseconds) to execute; the rendering speed then would be $1/0.020 = 50$ Hz. However, this is true only if the output device can update at this particular speed; otherwise, the true output rate will be slower. In other pipelining contexts, the term *throughput* is used instead of rendering speed.

EXAMPLE: RENDERING SPEED. Assume that our output device's maximum update frequency is 60 Hz, and that the bottleneck of the rendering pipeline has been found. Timings show that this stage takes 62.5 ms to execute. The rendering speed is then computed as follows. First, ignoring the output device, we get a maximum rendering speed of $1/0.0625 = 16$ fps. Second, adjust this value to the frequency of the output device: 60 Hz implies that rendering speed can be 60 Hz, $60/2 = 30$ Hz, $60/3 = 20$ Hz, $60/4 = 15$ Hz, $60/5 = 12$ Hz, and so forth. This means that we can expect the rendering speed to be 15 Hz, since this is the maximum constant speed the output device can manage that is less than 16 fps. □

As the name implies, the *application stage* is driven by the application and is therefore implemented in software running on general-purpose CPUs. These CPUs commonly include multiple cores that are capable of processing multiple *threads of execution* in parallel. This enables the CPUs to efficiently run the large variety of tasks that are the responsibility of the application stage. Some of the tasks traditionally performed on the CPU include collision detection, global acceleration algorithms, animation, physics simulation, and many others, depending on the type of application. The next step is the *geometry stage*, which deals with transforms, projections, etc. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a graphics processing unit (GPU) that contains many programmable cores as well as fixed-operation hardware. Finally, the *rasterizer stage* draws (renders) an image with use of the data that the previous stage generated, as well as any per-pixel computation desired. The rasterizer stage is processed completely on the GPU. These stages and their internal pipelines will be discussed in the next three sections. More details on how the GPU processes these stages are given in Chapter 3.

2.2 The Application Stage

The developer has full control over what happens in the application stage, since it executes on the CPU. Therefore, the developer can entirely determine the implementation and can later modify it in order to improve

performance. Changes here can also affect the performance of subsequent stages. For example, an application stage algorithm or setting could decrease the number of triangles to be rendered.

At the end of the application stage, the geometry to be rendered is fed to the geometry stage. These are the *rendering primitives*, i.e., points, lines, and triangles, that might eventually end up on the screen (or whatever output device is being used). This is the most important task of the application stage.

A consequence of the software-based implementation of this stage is that it is not divided into substages, as are the geometry and rasterizer stages.[1] However, in order to increase performance, this stage is often executed in parallel on several processor cores. In CPU design, this is called a *superscalar* construction, since it is able to execute several processes at the same time in the same stage. Section 15.5 presents various methods for utilizing multiple processor cores.

One process commonly implemented in this stage is *collision detection*. After a collision is detected between two objects, a response may be generated and sent back to the colliding objects, as well as to a force feedback device. The application stage is also the place to take care of input from other sources, such as the keyboard, the mouse, a head-mounted helmet, etc. Depending on this input, several different kinds of actions may be taken. Other processes implemented in this stage include texture animation, animations via transforms, or any kind of calculations that are not performed in any other stages. Acceleration algorithms, such as hierarchical view frustum culling (see Chapter 14), are also implemented here.

2.3 The Geometry Stage

The geometry stage is responsible for the majority of the per-polygon and per-vertex operations. This stage is further divided into the following functional stages: model and view transform, vertex shading, projection, clipping, and screen mapping (Figure 2.3). Note again that, depending on the implementation, these functional stages may or may not be equivalent to pipeline stages. In some cases, a number of consecutive functional stages form a single pipeline stage (which runs in parallel with the other pipeline stages). In other cases, a functional stage may be subdivided into several smaller pipeline stages.

[1] Since a CPU itself is pipelined on a much smaller scale, you could say that the application stage is further subdivided into several pipeline stages, but this is not relevant here.

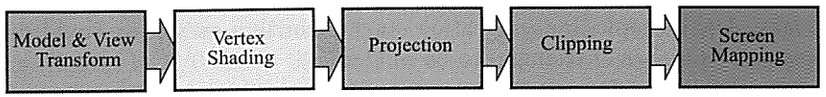

Figure 2.3. The geometry stage subdivided into a pipeline of functional stages.

For example, at one extreme, all stages in the entire rendering pipeline may run in software on a single processor, and then you could say that your entire pipeline consists of one pipeline stage. Certainly this was how all graphics were generated before the advent of separate accelerator chips and boards. At the other extreme, each functional stage could be subdivided into several smaller pipeline stages, and each such pipeline stage could execute on a designated processor core element.

2.3.1 Model and View Transform

On its way to the screen, a model is transformed into several different *spaces* or *coordinate systems*. Originally, a model resides in its own *model space*, which simply means that it has not been transformed at all. Each model can be associated with a *model transform* so that it can be positioned and oriented. It is possible to have several model transforms associated with a single model. This allows several copies (called *instances*) of the same model to have different locations, orientations, and sizes in the same scene, without requiring replication of the basic geometry.

It is the vertices and the normals of the model that are transformed by the model transform. The coordinates of an object are called *model coordinates*, and after the model transform has been applied to these coordinates, the model is said to be located in *world coordinates* or in *world space*. The world space is unique, and after the models have been transformed with their respective model transforms, all models exist in this same space.

As mentioned previously, only the models that the camera (or observer) sees are rendered. The camera has a location in world space and a direction, which are used to place and aim the camera. To facilitate projection and clipping, the camera and all the models are transformed with the *view transform*. The purpose of the view transform is to place the camera at the origin and aim it, to make it look in the direction of the negative z-axis,[2] with the y-axis pointing upwards and the x-axis pointing to the right. The actual position and direction after the view transform has been applied are dependent on the underlying application programming interface (API). The

[2] We will be using the $-z$-axis convention; some texts prefer looking down the $+z$-axis. The difference is mostly semantic, as transform between one and the other is simple.

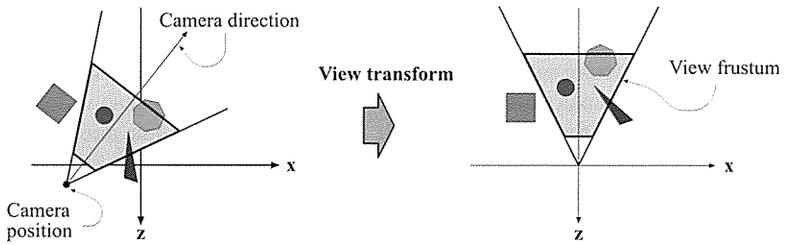

Figure 2.4. In the left illustration, the camera is located and oriented as the user wants it to be. The view transform relocates the camera at the origin, looking along the negative z-axis, as shown on the right. This is done to make the clipping and projection operations simpler and faster. The light gray area is the view volume. Here, perspective viewing is assumed, since the view volume is a frustum. Similar techniques apply to any kind of projection.

space thus delineated is called the *camera space*, or more commonly, the *eye space*. An example of the way in which the view transform affects the camera and the models is shown in Figure 2.4. Both the model transform and the view transform are implemented as 4×4 matrices, which is the topic of Chapter 4.

2.3.2 Vertex Shading

To produce a realistic scene, it is not sufficient to render the shape and position of objects, but their appearance must be modeled as well. This description includes each object's material, as well as the effect of any light sources shining on the object. Materials and lights can be modeled in any number of ways, from simple colors to elaborate representations of physical descriptions.

This operation of determining the effect of a light on a material is known as *shading*. It involves computing a *shading equation* at various points on the object. Typically, some of these computations are performed during the geometry stage on a model's vertices, and others may be performed during per-pixel rasterization. A variety of material data can be stored at each vertex, such as the point's location, a normal, a color, or any other numerical information that is needed to compute the shading equation. Vertex shading results (which can be colors, vectors, texture coordinates, or any other kind of shading data) are then sent to the rasterization stage to be interpolated.

Shading computations are usually considered as happening in world space. In practice, it is sometimes convenient to transform the relevant entities (such as the camera and light sources) to some other space (such

as model or eye space) and perform the computations there. This works because the relative relationships between light sources, the camera, and the models are preserved if all entities that are included in the shading calculations are transformed to the same space.

Shading is discussed in more depth throughout this book, most specifically in Chapters 3 and 5.

2.3.3 Projection

After shading, rendering systems perform *projection*, which transforms the view volume into a unit cube with its extreme points at $(-1, -1, -1)$ and $(1, 1, 1)$.[3] The unit cube is called the *canonical view volume*. There are two commonly used projection methods, namely *orthographic* (also called *parallel*)[4] and *perspective* projection. See Figure 2.5.

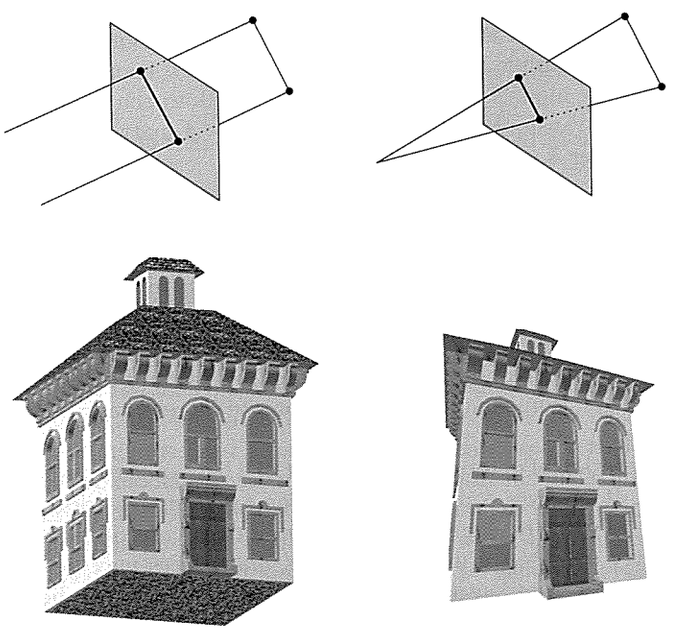

Figure 2.5. On the left is an orthographic, or parallel, projection; on the right is a perspective projection.

[3] Different volumes can be used, for example $0 \leq z \leq 1$. Blinn has an interesting article [102] on using other intervals.

[4] Actually, orthographic is just one type of parallel projection. For example, there is also an oblique parallel projection method [516], which is much less commonly used.

The view volume of orthographic viewing is normally a rectangular box, and the orthographic projection transforms this view volume into the unit cube. The main characteristic of orthographic projection is that parallel lines remain parallel after the transform. This transformation is a combination of a translation and a scaling.

The perspective projection is a bit more complex. In this type of projection, the farther away an object lies from the camera, the smaller it appears after projection. In addition, parallel lines may converge at the horizon. The perspective transform thus mimics the way we perceive objects' size. Geometrically, the view volume, called a *frustum*, is a truncated pyramid with rectangular base. The frustum is transformed into the unit cube as well. Both orthographic and perspective transforms can be constructed with 4×4 matrices (see Chapter 4), and after either transform, the models are said to be in *normalized device coordinates*.

Although these matrices transform one volume into another, they are called projections because after display, the z-coordinate is not stored in the image generated.[5] In this way, the models are projected from three to two dimensions.

2.3.4 Clipping

Only the primitives wholly or partially inside the view volume need to be passed on to the rasterizer stage, which then draws them on the screen. A primitive that lies totally inside the view volume will be passed on to the next stage as is. Primitives entirely outside the view volume are not passed on further, since they are not rendered. It is the primitives that are partially inside the view volume that require *clipping*. For example, a line that has one vertex outside and one inside the view volume should be clipped against the view volume, so that the vertex that is outside is replaced by a new vertex that is located at the intersection between the line and the view volume. The use of a projection matrix means that the transformed primitives are clipped against the unit cube. The advantage of performing the view transformation and projection before clipping is that it makes the clipping problem consistent; primitives are always clipped against the unit cube. The clipping process is depicted in Figure 2.6. In addition to the six clipping planes of the view volume, the user can define additional clipping planes to visibly chop objects. An image showing this type of visualization, called *sectioning*, is shown in Figure 14.1 on page 646. Unlike the previous geometry stages, which are typically performed by programmable processing units, the clipping stage (as well as the subsequent screen mapping stage) is usually processed by fixed-operation hardware.

[5] Rather, the z-coordinate is stored in a Z-buffer. See Section 2.4.

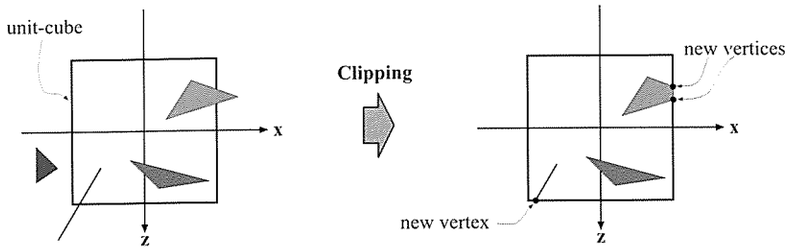

Figure 2.6. After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded and primitives totally inside are kept. Primitives intersecting with the unit cube are clipped against the unit cube, and thus new vertices are generated and old ones are discarded.

2.3.5 Screen Mapping

Only the (clipped) primitives inside the view volume are passed on to the screen mapping stage, and the coordinates are still three dimensional when entering this stage. The x- and y-coordinates of each primitive are transformed to form *screen coordinates*. Screen coordinates together with the z-coordinates are also called *window coordinates*. Assume that the scene should be rendered into a window with the minimum corner at (x_1, y_1) and the maximum corner at (x_2, y_2), where $x_1 < x_2$ and $y_1 < y_2$. Then the screen mapping is a translation followed by a scaling operation. The z-coordinate is not affected by this mapping. The new x- and y-coordinates are said to be screen coordinates. These, along with the z-coordinate ($-1 \leq z \leq 1$), are passed on to the rasterizer stage. The screen mapping process is depicted in Figure 2.7.

A source of confusion is how integer and floating point values relate to pixel (and texture) coordinates. DirectX 9 and its predecessors use a

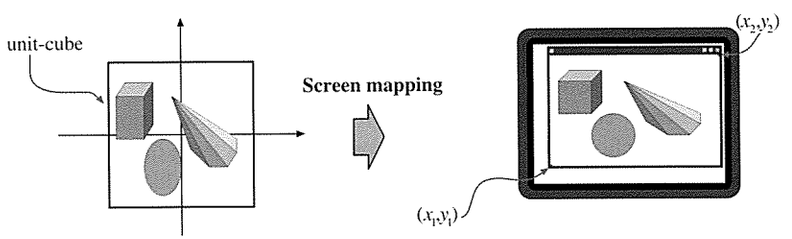

Figure 2.7. The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

coordinate system where 0.0 is the center of the pixel, meaning that a range of pixels $[0, 9]$ cover a span from $[-0.5, 9.5)$. Heckbert [520] presents a more logically consistent scheme. Given a horizontal array of pixels and using Cartesian coordinates, the left edge of the leftmost pixel is 0.0 in floating point coordinates. OpenGL has always used this scheme, and DirectX 10 and its successors use it. The center of this pixel is at 0.5. So a range of pixels $[0, 9]$ cover a span from $[0.0, 10.0)$. The conversions are simply

$$d = \texttt{floor(c)}, \tag{2.1}$$

$$c = d + 0.5, \tag{2.2}$$

where d is the discrete (integer) index of the pixel and c is the continuous (floating point) value within the pixel.

While all APIs have pixel location values that increase going from left to right, the location of zero for the top and bottom edges is inconsistent in some cases between OpenGL and DirectX.[6] OpenGL favors the Cartesian system throughout, treating the lower left corner as the lowest-valued element, while DirectX sometimes defines the upper left corner as this element, depending on the context. There is a logic to each, and no right answer exists where they differ. As an example, $(0, 0)$ is located at the lower left corner of an image in OpenGL, while it is upper left for DirectX. The reasoning for DirectX is that a number of phenomena go from top to bottom on the screen: Microsoft Windows uses this coordinate system, we read in this direction, and many image file formats store their buffers in this way. The key point is that the difference exists and is important to take into account when moving from one API to the other.

2.4 The Rasterizer Stage

Given the transformed and projected vertices with their associated shading data (all from the geometry stage), the goal of the rasterizer stage is to compute and set colors for the pixels[7] covered by the object. This process is called *rasterization* or *scan conversion*, which is thus the conversion from two-dimensional vertices in screen space—each with a z-value (depth-value), and various shading information associated with each vertex—into pixels on the screen.

[6] "Direct3D" is the three-dimensional graphics API component of DirectX. DirectX includes other API elements, such an input and audio control. Rather than differentiate between writing "DirectX" when specifying a particular release and "Direct3D" when discussing this particular API, we follow common usage by writing "DirectX" throughout.

[7] Short for *picture elements*.

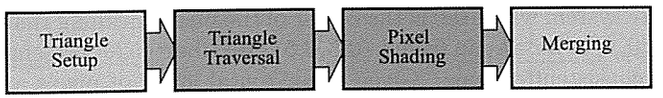

Figure 2.8. The rasterizer stage subdivided into a pipeline of functional stages.

Similar to the geometry stage, this stage is divided into several functional stages: triangle setup, triangle traversal, pixel shading, and merging (Figure 2.8).

2.4.1 Triangle Setup

In this stage the differentials and other data for the triangle's surface are computed. This data is used for scan conversion, as well as for interpolation of the various shading data produced by the geometry stage. This process is performed by fixed-operation hardware dedicated to this task.

2.4.2 Triangle Traversal

Here is where each pixel that has its center (or a sample) covered by the triangle is checked and a *fragment* generated for the part of the pixel that overlaps the triangle. Finding which samples or pixels are inside a triangle is often called *triangle traversal* or *scan conversion*. Each triangle fragment's properties are generated using data interpolated among the three triangle vertices (see Chapter 5). These properties include the fragment's depth, as well as any shading data from the geometry stage. Akeley and Jermoluk [7] and Rogers [1077] offer more information on triangle traversal.

2.4.3 Pixel Shading

Any per-pixel shading computations are performed here, using the interpolated shading data as input. The end result is one or more colors to be passed on to the next stage. Unlike the triangle setup and traversal stages, which are usually performed by dedicated, hardwired silicon, the pixel shading stage is executed by programmable GPU cores. A large variety of techniques can be employed here, one of the most important of which is *texturing*. Texturing is treated in more detail in Chapter 6. Simply put, texturing an object means "gluing" an image onto that object. This process is depicted in Figure 2.9. The image may be one-, two-, or three-dimensional, with two-dimensional images being the most common.

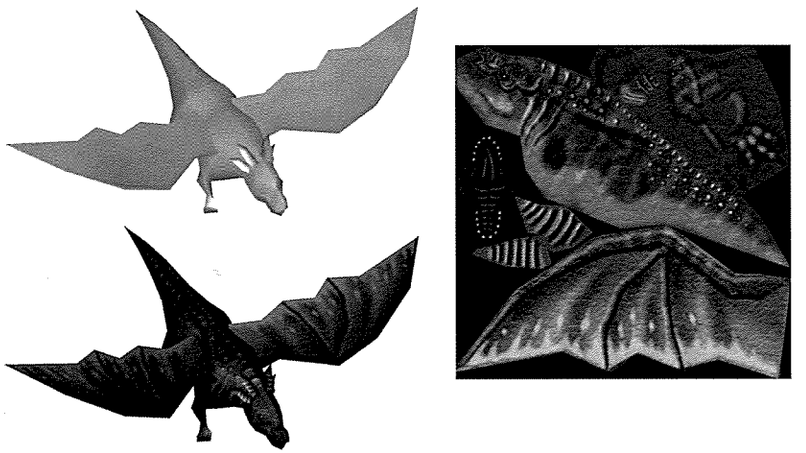

Figure 2.9. A dragon model without textures is shown in the upper left. The pieces in the image texture are "glued" onto the dragon, and the result is shown in the lower left.

2.4.4 Merging

The information for each pixel is stored in the *color buffer*, which is a rectangular array of colors (a red, a green, and a blue component for each color). It is the responsibility of the merging stage to combine the fragment color produced by the shading stage with the color currently stored in the buffer. Unlike the shading stage, the GPU subunit that typically performs this stage is not fully programmable. However, it is highly configurable, enabling various effects.

This stage is also responsible for resolving visibility. This means that when the whole scene has been rendered, the color buffer should contain the colors of the primitives in the scene that are visible from the point of view of the camera. For most graphics hardware, this is done with the Z-buffer (also called *depth buffer*) algorithm [162].[8] A Z-buffer is the same size and shape as the color buffer, and for each pixel it stores the z-value from the camera to the currently closest primitive. This means that when a primitive is being rendered to a certain pixel, the z-value on that primitive at that pixel is being computed and compared to the contents of the Z-buffer at the same pixel. If the new z-value is smaller than the z-value in the Z-buffer, then the primitive that is being rendered is closer to the camera than the primitive that was previously closest to the camera at that pixel. Therefore, the z-value and the color of that pixel are updated

[8] When a Z-buffer is not available, a *BSP tree* can be used to help render a scene in back-to-front order. See Section 14.1.2 for information about BSP trees.

with the z-value and color from the primitive that is being drawn. If the computed z-value is greater than the z-value in the Z-buffer, then the color buffer and the Z-buffer are left untouched. The Z-buffer algorithm is very simple, has $O(n)$ convergence (where n is the number of primitives being rendered), and works for any drawing primitive for which a z-value can be computed for each (relevant) pixel. Also note that this algorithm allows most primitives to be rendered in any order, which is another reason for its popularity. However, partially transparent primitives cannot be rendered in just any order. They must be rendered after all opaque primitives, and in back-to-front order (Section 5.7). This is one of the major weaknesses of the Z-buffer.

We have mentioned that the color buffer is used to store colors and that the Z-buffer stores z-values for each pixel. However, there are other channels and buffers that can be used to filter and capture fragment information. The *alpha channel* is associated with the color buffer and stores a related opacity value for each pixel (Section 5.7). An optional *alpha test* can be performed on an incoming fragment before the depth test is performed.[9] The alpha value of the fragment is compared by some specified test (equals, greater than, etc.) to a reference value. If the fragment fails to pass the test, it is removed from further processing. This test is typically used to ensure that fully transparent fragments do not affect the Z-buffer (see Section 6.6).

The *stencil buffer* is an offscreen buffer used to record the locations of the rendered primitive. It typically contains eight bits per pixel. Primitives can be rendered into the stencil buffer using various functions, and the buffer's contents can then be used to control rendering into the color buffer and Z-buffer. As an example, assume that a filled circle has been drawn into the stencil buffer. This can be combined with an operator that allows rendering of subsequent primitives into the color buffer only where the circle is present. The stencil buffer is a powerful tool for generating special effects. All of these functions at the end of the pipeline are called *raster operations* (ROP) or *blend operations.*

The *frame buffer* generally consists of all the buffers on a system, but it is sometimes used to mean just the color buffer and Z-buffer as a set. In 1990, Haeberli and Akeley [474] presented another complement to the frame buffer, called the *accumulation buffer.* In this buffer, images can be accumulated using a set of operators. For example, a set of images showing an object in motion can be accumulated and averaged in order to generate motion blur. Other effects that can be generated include depth of field, antialiasing, soft shadows, etc.

[9] In DirectX 10, the alpha test is no longer part of this stage, but rather a function of the pixel shader.

When the primitives have reached and passed the rasterizer stage, those that are visible from the point of view of the camera are displayed on screen. The screen displays the contents of the color buffer. To avoid allowing the human viewer to see the primitives as they are being rasterized and sent to the screen, *double buffering* is used. This means that the rendering of a scene takes place off screen, in a *back buffer.* Once the scene has been rendered in the back buffer, the contents of the back buffer are swapped with the contents of the *front buffer* that was previously displayed on the screen. The swapping occurs during *vertical retrace*, a time when it is safe to do so.

For more information on different buffers and buffering methods, see Sections 5.6.2 and 18.1.

2.5 Through the Pipeline

Points, lines, and triangles are the rendering primitives from which a model or an object is built. Imagine that the application is an interactive *computer aided design* (CAD) application, and that the user is examining a design for a cell phone. Here we will follow this model through the entire graphics rendering pipeline, consisting of the three major stages: application, geometry, and the rasterizer. The scene is rendered with perspective into a window on the screen. In this simple example, the cell phone model includes both lines (to show the edges of parts) and triangles (to show the surfaces). Some of the triangles are textured by a two-dimensional image, to represent the keyboard and screen. For this example, shading is computed completely in the geometry stage, except for application of the texture, which occurs in the rasterization stage.

Application

CAD applications allow the user to select and move parts of the model. For example, the user might select the top part of the phone and then move the mouse to flip the phone open. The application stage must translate the mouse move to a corresponding rotation matrix, then see to it that this matrix is properly applied to the lid when it is rendered. Another example: An animation is played that moves the camera along a predefined path to show the cell phone from different views. The camera parameters, such as position and view direction, must then be updated by the application, dependent upon time. For each frame to be rendered, the application stage feeds the camera position, lighting, and primitives of the model to the next major stage in the pipeline—the geometry stage.

Geometry

The view transform was computed in the application stage, along with a model matrix for each object that specifies its location and orientation. For

each object passed to the geometry stage, these two matrices are usually multiplied together into a single matrix. In the geometry stage the vertices and normals of the object are transformed with this concatenated matrix, putting the object into eye space. Then shading at the vertices is computed, using material and light source properties. Projection is then performed, transforming the object into a unit cube's space that represents what the eye sees. All primitives outside the cube are discarded. All primitives intersecting this unit cube are clipped against the cube in order to obtain a set of primitives that lies entirely inside the unit cube. The vertices then are mapped into the window on the screen. After all these per-polygon operations have been performed, the resulting data is passed on to the rasterizer—the final major stage in the pipeline.

Rasterizer

In this stage, all primitives are rasterized, i.e., converted into pixels in the window. Each visible line and triangle in each object enters the rasterizer in screen space, ready to convert. Those triangles that have been associated with a texture are rendered with that texture (image) applied to them. Visibility is resolved via the Z-buffer algorithm, along with optional alpha and stencil tests. Each object is processed in turn, and the final image is then displayed on the screen.

Conclusion

This pipeline resulted from decades of API and graphics hardware evolution targeted to real-time rendering applications. It is important to note that this is not the only possible rendering pipeline; offline rendering pipelines have undergone different evolutionary paths. Rendering for film production is most commonly done with *micropolygon* pipelines [196, 1236]. Academic research and *predictive rendering* applications such as architectural previsualization usually employ *ray tracing* renderers (see Section 9.8.2).

For many years, the only way for application developers to use the process described here was through a *fixed-function pipeline* defined by the graphics API in use. The fixed-function pipeline is so named because the graphics hardware that implements it consists of elements that cannot be programmed in a flexible way. Various parts of the pipeline can be set to different states, e.g., Z-buffer testing can be turned on or off, but there is no ability to write programs to control the order in which functions are applied at various stages. The latest (and probably last) example of a fixed-function machine is Nintendo's Wii. Programmable GPUs make it possible to determine exactly what operations are applied in various substages throughout the pipeline. While studying the fixed-function pipeline provides a reasonable introduction to some basic principles, most new development is aimed at programmable GPUs. This programmability is the default assumption for this third edition of the book, as it is the modern way to take full advantage of the GPU.

Further Reading and Resources

Blinn's book *A Trip Down the Graphics Pipeline* [105] is an older book about writing a software renderer from scratch, but is a good resource for learning about some of the subtleties of implementing a rendering pipeline. For the fixed-function pipeline, the venerable (yet frequently updated) *OpenGL Programming Guide* (a.k.a., the "Red Book") [969] provides a thorough description of the fixed-function pipeline and algorithms related to its use. Our book's website, http://www.realtimerendering.com, gives links to a variety of rendering engine implementations.