Chapter 2
The Graphics Rendering Pipeline

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

This chapter presents 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, 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 (images applied to surfaces), and shading equations.

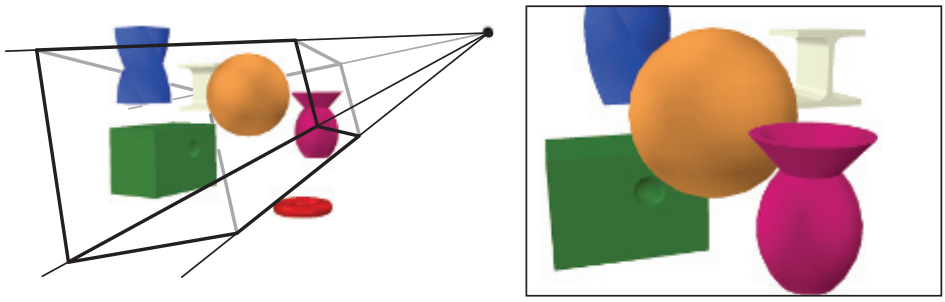

Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid (where four lines converge). Only the primitives inside the view volume are rendered. For an image that is rendered in perspective (as is the case here), the view volume is a *frustum* (plural: *frusta*), i.e., a truncated 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.

We will explain the different stages of the rendering pipeline, with a focus on function rather than implementation. Relevant details for applying these stages will be covered in later chapters.

2.1 Architecture

In the physical world, the pipeline concept manifests itself in many different forms, from factory assembly lines to fast food kitchens. It also applies to graphics rendering. A pipeline consists of several stages [715], each of which performs part of a larger task.

The pipeline stages execute in parallel, with each stage dependent upon the result of the previous stage. 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 large number of sandwiches can be prepared quickly by a series of people—one preparing the bread, another adding meat, another adding toppings. Each passes the result to the next person in line and immediately starts work on the next sandwich. If each person takes twenty seconds to perform their task, a maximum rate of one sandwich every twenty seconds, three a minute, is possible. The pipeline stages execute in parallel, but they are stalled until the slowest stage has finished its task. For example, say the meat addition stage becomes more involved, taking thirty seconds. Now the best rate that can be achieved is two sandwiches a minute. For this particular pipeline, the meat stage is the *bottleneck*, since it determines the speed of the entire production. The toppings stage is said to be *starved* (and the customer, too) during the time it waits for the meat stage to be done.

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 four main stages—*application*, *geometry processing*, *rasterization*, and *pixel processing*—is shown in Figure 2.2. This structure is the core—the engine of the rendering pipeline—which is used in real-time computer graphics applications and is thus an essential base for

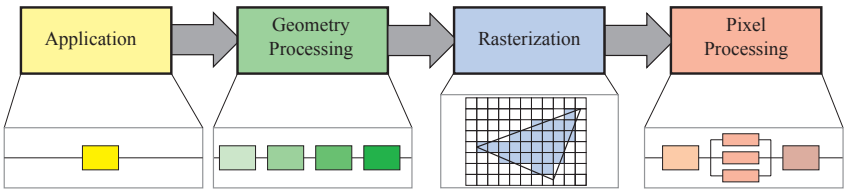

Figure 2.2. The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, as shown below the pixel processing stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized. Note that rasterization finds the pixels inside a primitive, e.g., a triangle.

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 functional stages shown here and the structure of their implementation. A functional stage has a certain task to perform but does not specify the way that task is executed in the pipeline. A given implementation may combine two functional stages into one unit or execute using programmable cores, while it divides another, more time-consuming, functional stage into several hardware units.

The rendering 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. It is also common to just state the time, in milliseconds (ms), that it takes to render an image. The time 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.

As the name implies, the *application* stage is driven by the application and is therefore typically 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 main stage is *geometry processing*, which deals with transforms, projections, and all other types of geometry handling. 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. The *rasterization* stage typically takes as input three vertices, forming a triangle, and finds all pixels that are considered inside that triangle, then forwards these to the next stage. Finally, the *pixel processing* stage executes a program per pixel to determine its color and may perform depth testing to see whether it is visible or not. It may also perform per-pixel operations such as blending the newly computed color with a previous color. The rasterization and pixel processing stages are also processed entirely on the GPU. All these stages and their internal pipelines will be discussed in the next four 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 usually 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.

All this said, some application work can be performed by the GPU, using a separate mode called a *compute shader*. This mode treats the GPU as a highly parallel general processor, ignoring its special functionality meant specifically for rendering graphics.

At the end of the application stage, the geometry to be rendered is fed to the geometry processing 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 processing, rasterization, and pixel processing stages.[1] However, 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 18.5 presents various methods for using 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, or a head-mounted display. Depending on this input, several different kinds of actions may be taken. Acceleration algorithms, such as particular culling algorithms (Chapter 19), are also implemented here, along with whatever else the rest of the pipeline cannot handle.

2.3 Geometry Processing

The geometry processing stage on the GPU is responsible for most of the per-triangle and per-vertex operations. This stage is further divided into the following functional stages: vertex shading, projection, clipping, and screen mapping (Figure 2.3).

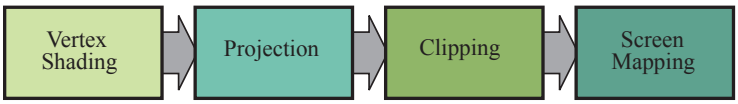

Figure 2.3. The geometry processing stage divided into a pipeline of functional stages.

[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.

2.3.1 Vertex Shading

There are two main tasks of vertex shading, namely, to compute the position for a vertex and to evaluate whatever the programmer may like to have as vertex output data, such as a normal and texture coordinates. Traditionally much of the shade of an object was computed by applying lights to each vertex's location and normal and storing only the resulting color at the vertex. These colors were then interpolated across the triangle. For this reason, this programmable vertex processing unit was named the vertex shader [1049]. With the advent of the modern GPU, along with some or all of the shading taking place per pixel, this vertex shading stage is more general and may not evaluate any shading equations at all, depending on the programmer's intent. The vertex shader is now a more general unit dedicated to setting up the data associated with each vertex. As an example, the vertex shader can animate an object using the methods in Sections 4.4 and 4.5.

We start by describing how the vertex position is computed, a set of coordinates that is always required. 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, with the y-axis pointing upward and the x-axis pointing to the right. We use 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. The actual position and direction after the view transform has been applied are dependent on the underlying application programming interface (API). The space thus delineated is called *camera space*, or more commonly, *view space* or *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 may be implemented as 4×4 matrices, which is the topic of Chapter 4. However, it is important to realize that

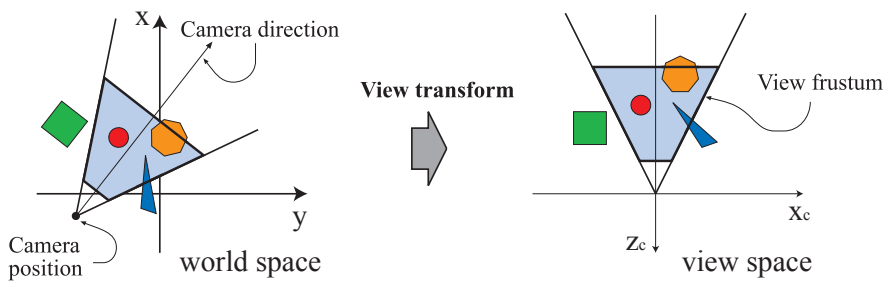

Figure 2.4. In the left illustration, a top-down view shows the camera located and oriented as the user wants it to be, in a world where the $+z$-axis is up. The view transform reorients the world so that the camera is at the origin, looking along its negative z-axis, with the camera's $+y$-axis up, as shown on the right. This is done to make the clipping and projection operations simpler and faster. The light blue 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.

the position and normal of a vertex can be computed in whatever way the programmer prefers.

Next, we describe the second type of output from 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 geometry processing on a model's vertices, and others may be performed during per-pixel processing. 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 evaluate the shading equation. Vertex shading results (which can be colors, vectors, texture coordinates, along with any other kind of shading data) are then sent to the rasterization and pixel processing stages to be interpolated and used to compute the shading of the surface.

Vertex shading in the form of the GPU vertex shader is discussed in more depth throughout this book and most specifically in Chapters 3 and 5.

As part of vertex shading, rendering systems perform *projection* and then clipping, which transforms the view volume into a unit cube with its extreme points at $(-1,-1,-1)$ and $(1,1,1)$. Different ranges defining the same volume can and are used, for example, $0 \leq z \leq 1$. The unit cube is called the *canonical view volume*. Projection is done first, and on the GPU it is done by the vertex shader. There are two commonly used projection methods, namely *orthographic* (also called *parallel*)

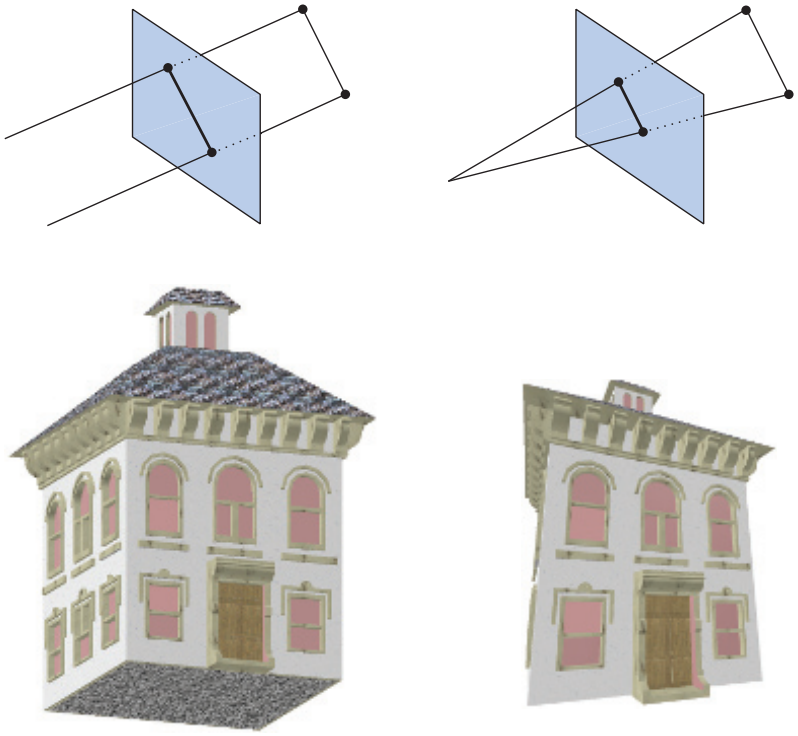

Figure 2.5. On the left is an orthographic, or parallel, projection; on the right is a perspective projection.

and *perspective* projection. See Figure 2.5. In truth, orthographic is just one type of parallel projection. Several others find use, particularly in the field of architecture, such as oblique and axonometric projections. The old arcade game *Zaxxon* is named from the latter.

Note that projection is expressed as a matrix (Section 4.7) and so it may sometimes be concatenated with the rest of the geometry transform.

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 (Chapter 4), and after either transform, the models are said to be in *clip coordinates*. These are in fact homogeneous coordinates, discussed in Chapter 4, and so this occurs before division by w. The GPU's vertex shader must always output coordinates of this type in order for the next functional stage, clipping, to work correctly.

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 but is stored in a z-buffer, described in Section 2.5. In this way, the models are projected from three to two dimensions.

2.3.2 Optional Vertex Processing

Every pipeline has the vertex processing just described. Once this processing is done, there are a few optional stages that can take place on the GPU, in this order: tessellation, geometry shading, and stream output. Their use depends both on the capabilities of the hardware—not all GPUs have them—and the desires of the programmer. They are independent of each other, and in general they are not commonly used. More will be said about each in Chapter 3.

The first optional stage is *tessellation*. Imagine you have a bouncing ball object. If you represent it with a single set of triangles, you can run into problems with quality or performance. Your ball may look good from 5 meters away, but up close the individual triangles, especially along the silhouette, become visible. If you make the ball with more triangles to improve quality, you may waste considerable processing time and memory when the ball is far away and covers only a few pixels on the screen. With tessellation, a curved surface can be generated with an appropriate number of triangles.

We have talked a bit about triangles, but up to this point in the pipeline we have just processed vertices. These could be used to represent points, lines, triangles, or other objects. Vertices can be used to describe a curved surface, such as a ball. Such surfaces can be specified by a set of patches, and each patch is made of a set of vertices. The tessellation stage consists of a series of stages itself—hull shader, tessellator, and domain shader—that converts these sets of patch vertices into (normally) larger sets of vertices that are then used to make new sets of triangles. The camera for the scene can be used to determine how many triangles are generated: many when the patch is close, few when it is far away.

The next optional stage is the *geometry shader*. This shader predates the tessellation shader and so is more commonly found on GPUs. It is like the tessellation shader in that it takes in primitives of various sorts and can produce new vertices. It is a much simpler stage in that this creation is limited in scope and the types of output primitives are much more limited. Geometry shaders have several uses, with one of the most popular being particle generation. Imagine simulating a fireworks explosion.

Each fireball could be represented by a point, a single vertex. The geometry shader can take each point and turn it into a square (made of two triangles) that faces the viewer and covers several pixels, so providing a more convincing primitive for us to shade.

The last optional stage is called *stream output*. This stage lets us use the GPU as a geometry engine. Instead of sending our processed vertices down the rest of the pipeline to be rendered to the screen, at this point we can optionally output these to an array for further processing. These data can be used by the CPU, or the GPU itself, in a later pass. This stage is typically used for particle simulations, such as our fireworks example.

These three stages are performed in this order—tessellation, geometry shading, and stream output—and each is optional. Regardless of which (if any) options are used, if we continue down the pipeline we have a set of vertices with homogeneous coordinates that will be checked for whether the camera views them.

2.3.3 Clipping

Only the primitives wholly or partially inside the view volume need to be passed on to the rasterization stage (and the subsequent pixel processing stage), which then draws them on the screen. A primitive that lies fully 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 19.1 on page 818.

The clipping step uses the 4-value homogeneous coordinates produced by projection to perform clipping. Values do not normally interpolate linearly across a triangle in perspective space. The fourth coordinate is needed so that data are properly interpolated and clipped when a perspective projection is used. Finally, *perspective division* is performed, which places the resulting triangles' positions into three-dimensional *normalized device coordinates*. As mentioned earlier, this view volume ranges from $(-1, -1, -1)$ to $(1, 1, 1)$. The last step in the geometry stage is to convert from this space to window coordinates.

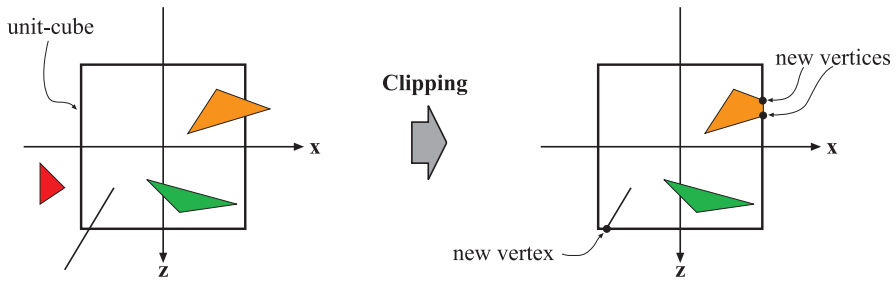

Figure 2.6. After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded, and primitives fully inside are kept. Primitives intersecting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

2.3.4 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 new x- and y-coordinates are said to be screen coordinates. The z-coordinate ($[-1, +1]$ for OpenGL and $[0, 1]$ for DirectX) is also mapped to $[z_1, z_2]$, with $z_1 = 0$ and $z_2 = 1$ as the default values. These can be changed with the API, however. The window coordinates along with this remapped z-value are passed on to the rasterizer stage. The screen mapping process is depicted in Figure 2.7.

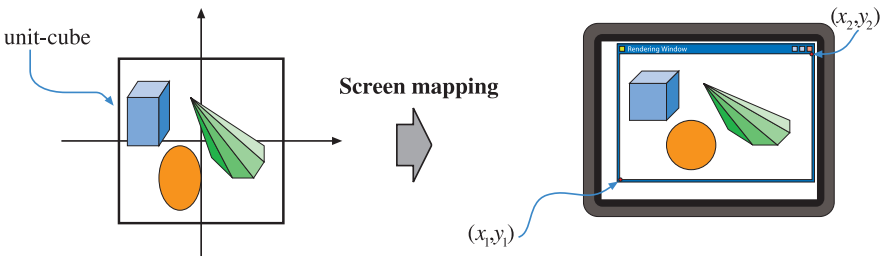

Figure 2.7. The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

Next, we describe how integer and floating point values relate to pixels (and texture coordinates). 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

$$d = \texttt{floor}(\texttt{c}), \tag{2.1}$$
$$c = d + 0.5, \tag{2.2}$$

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.[2] 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. This difference is important to take into account when moving from one API to the other.

2.4 Rasterization

Given the transformed and projected vertices with their associated shading data (all from geometry processing), the goal of the next stage is to find all pixels—short for *picture elements*—that are inside the primitive, e.g., a triangle, being rendered. We call this process *rasterization*, and it is split up into two functional substages: triangle setup (also called primitive assembly) and triangle traversal. These are shown to the left in Figure 2.8. Note that these can handle points and lines as well, but since triangles are most common, the substages have "triangle" in their names. Rasterization, also called *scan conversion*, 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. Rasterization can also be thought of as a synchronization point between geometry processing and pixel processing, since it is here that triangles are formed from three vertices and eventually sent down to pixel processing.

Whether the triangle is considered to overlap the pixel depends on how you have set up the GPU's pipeline. For example, you may use point sampling to determine

[2] "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.

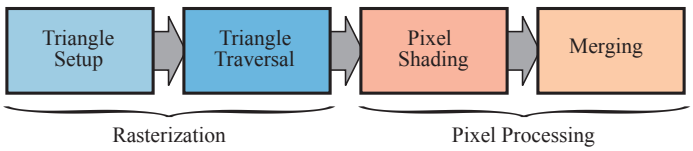

Figure 2.8. Left: rasterization split into two functional stages, called triangle setup and triangle traversal. Right: pixel processing split into two functional stages, namely, pixel processing and merging.

"insideness." The simplest case uses a single point sample in the center of each pixel, and so if that center point is inside the triangle then the corresponding pixel is considered inside the triangle as well. You may also use more than one sample per pixel using supersampling or multisampling antialiasing techniques (Section 5.4.2). Yet another way is to use conservative rasterization, where the definition is that a pixel is "inside" the triangle if at least part of the pixel overlaps with the triangle (Section 23.1.2).

2.4.1 Triangle Setup

In this stage the differentials, edge equations, and other data for the triangle are computed. These data may be used for triangle traversal (Section 2.4.2), as well as for interpolation of the various shading data produced by the geometry stage. Fixed-function hardware is used for 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. More elaborate sampling methods can be found in Section 5.4. Finding which samples or pixels are inside a triangle is often called *triangle traversal*. Each triangle fragment's properties are generated using data interpolated among the three triangle vertices (Chapter 5). These properties include the fragment's depth, as well as any shading data from the geometry stage. McCormack et al. [1162] offer more information on triangle traversal. It is also here that perspective-correct interpolation over the triangles is performed [694] (Section 23.1.1). All pixels or samples that are inside a primitive are then sent to the pixel processing stage, described next.

2.5 Pixel Processing

At this point, all the pixels that are considered inside a triangle or other primitive have been found as a consequence of the combination of all the previous stages. The

pixel processing stage is divided into *pixel shading* and *merging*, shown to the right in Figure 2.8. Pixel processing is the stage where per-pixel or per-sample computations and operations are performed on pixels or samples that are inside a primitive.

2.5.1 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. To that end, the programmer supplies a program for the pixel shader (or fragment shader, as it is known in OpenGL), which can contain any desired computations. 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" one or more images onto that object, for a variety of purposes. A simple example of 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. At its simplest, the end product is a color value for each fragment, and these are passed on to the next substage.

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.

2.5.2 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 pixel shading stage with the color currently stored in the buffer. This stage is also called ROP, standing for "raster operations (pipeline)" or "render output unit," depending on who you ask. Unlike the shading stage, the GPU subunit that performs this stage is typically 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 or even all graphics hardware, this is done with the z-buffer (also called *depth buffer*) algorithm [238]. A z-buffer is the same size and shape as the color buffer, and for each pixel it stores the z-value 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 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 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, the z-buffer stores only a single depth at each point on the screen, so it cannot be used for partially transparent primitives. These must be rendered after all opaque primitives, and in back-to-front order, or using a separate order-independent algorithm (Section 5.5). Transparency is one of the major weaknesses of the basic 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 5.5). In older APIs, the alpha channel was also used to discard pixels selectively via the alpha test feature. Nowadays a discard operation can be inserted into the pixel shader program and any type of computation can be used to trigger a discard. This type of test can be used to ensure that fully transparent fragments do not affect the z-buffer (Section 6.6).

The *stencil buffer* is an offscreen buffer used to record the locations of the rendered primitive. It typically contains 8 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 can be a powerful tool for generating some special effects. All these functions at the end of the pipeline are called *raster operations* (ROP) or *blend operations*. It is possible to mix the color currently in the color buffer with the color of the pixel being processed inside a triangle. This can enable effects such as transparency or the accumulation of color samples. As mentioned, blending is typically configurable using the API and not fully programmable. However, some APIs have support for raster order views, also called pixel shader ordering, which enable programmable blending capabilities.

The *framebuffer* generally consists of all the buffers on a system.

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 often occurs during *vertical retrace*, a time when it is safe to do so.

For more information on different buffers and buffering methods, see Sections 5.4.2, 23.6, and 23.7.

2.6 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 waffle maker. Here we will follow this model through the entire graphics rendering pipeline, consisting of the four major stages: application, geometry, rasterization, and pixel processing. The scene is rendered with perspective into a window on the screen. In this simple example, the waffle maker model includes both lines (to show the edges of parts) and triangles (to show the surfaces). The waffle maker has a lid that can be opened. Some of the triangles are textured by a two-dimensional image with the manufacturer's logo. For this example, surface 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 lid and then move the mouse to open it. 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 waffle maker 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 Processing

For perspective viewing, we assume here that the application has supplied a projection matrix. Also, for each object, the application has computed a matrix that describes both the view transform and the location and orientation of the object in itself. In our example, the waffle maker's base would have one matrix, the lid another. In the geometry stage the vertices and normals of the object are transformed with this matrix, putting the object into view space. Then shading or other calculations at the vertices may be computed, using material and light source properties. Projection is then performed using a separate user-supplied projection matrix, 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-triangle and per-vertex operations have been performed, the resulting data are passed on to the rasterization stage.

Rasterization

All the primitives that survive clipping in the previous stage are then rasterized, which means that all pixels that are inside a primitive are found and sent further down the pipeline to pixel processing.

Pixel Processing

The goal here is to compute the color of each pixel of each visible primitive. Those triangles that have been associated with any textures (images) are rendered with these images applied to them as desired. Visibility is resolved via the z-buffer algorithm, along with optional discard 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 was often done with *micropolygon* pipelines [289, 1734], but ray tracing and path tracing have taken over lately. These

techniques, covered in Section 11.2.2, may also be used in architectural and design previsualization.

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. The last example of a major fixed-function machine is Nintendo's Wii, introduced in 2006. Programmable GPUs, on the other hand, make it possible to determine exactly what operations are applied in various sub-stages throughout the pipeline. For the fourth edition of the book, we assume that all development is done using programmable GPUs.

Further Reading and Resources

Blinn's book *A Trip Down the Graphics Pipeline* [165] is an older book about writing a software renderer from scratch. It is a good resource for learning about some of the subtleties of implementing a rendering pipeline, explaining key algorithms such as clipping and perspective interpolation. The venerable (yet frequently updated) *OpenGL Programming Guide* (a.k.a. the "Red Book") [885] provides a thorough description of the graphics pipeline and algorithms related to its use. Our book's website, realtimerendering.com, gives links to a variety of pipeline diagrams, rendering engine implementations, and more.