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).

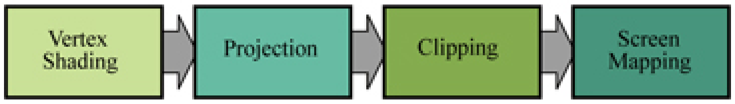

Figure 2.3 The geometry processing stage divided into a pipeline of functional stages.

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

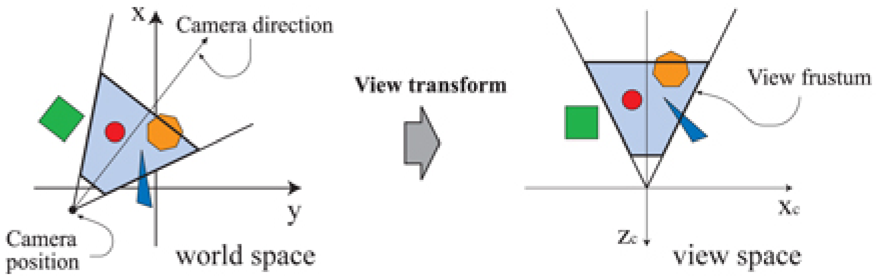

Figure 2.4 In the left illustration, a top-down view shows the camera located and oriented as the user wants it to be, in a world where the $+z$-axis is up. The view transform reorients the world so that the camera is at the origin, looking along its negative z-axis, with the camera's $+y$-axis up, as 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.

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*) 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.

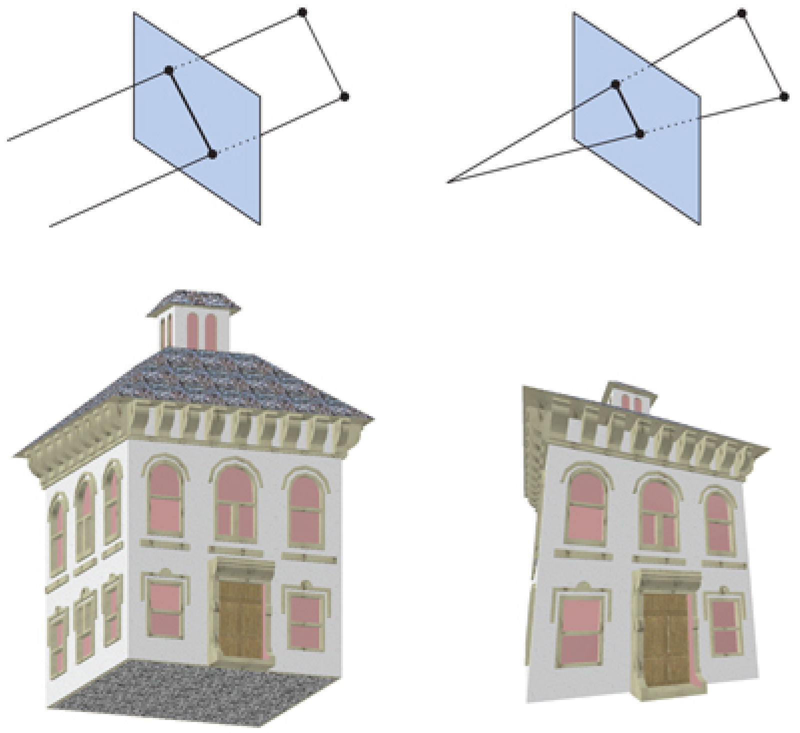
Figure 2.5 On the left is an orthographic, or parallel, projection; on the right is a perspective projection.

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.

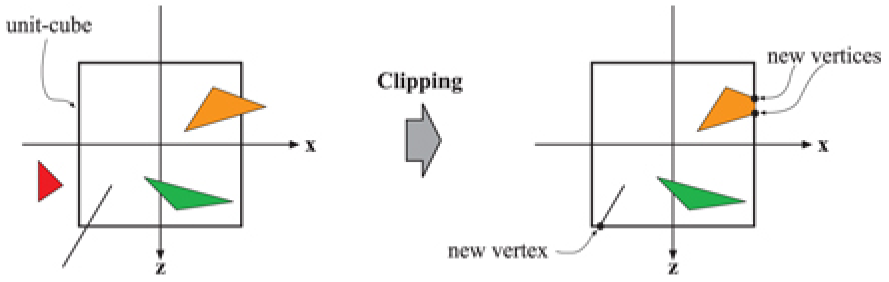

Figure 2.6 After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded, and primitives fully inside are kept. Primitives intersecting 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 $(x1, y1)$ and the maximum corner at $(x2, y2)$, where $x1 < x2$ and $y1 < y2$. 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 $[z1, z2]$, with $z1 = 0$ and $z2 = 1$ 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.

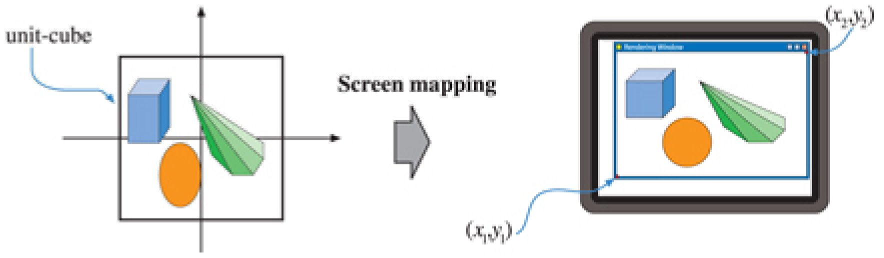

Figure 2.7 The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

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

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

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

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. [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.