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.

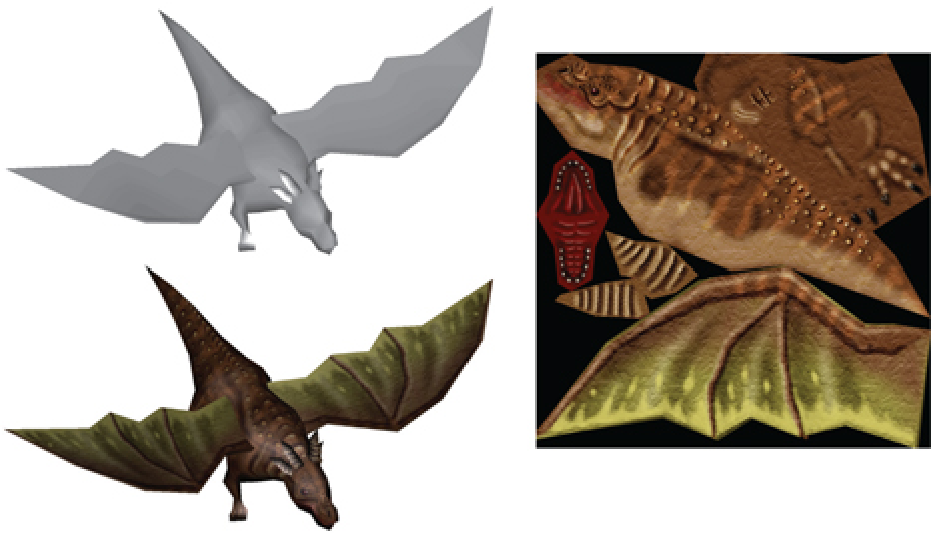

Figure 2.9 A dragon model without textures is shown in the upper left. The pieces in the image texture are “glued” onto the dragon, and the result is shown in the lower left.

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.