CHAPTER 3

The Graphics Pipeline

This chapter describes every step in the execution of shaders, including how they hook into the graphics pipeline.

By graphics pipeline we intend the steps needed to render a 3D scene into a 2D screen. As mentioned in the first chapter, shader execution is composed of many steps. Some of those steps are implemented completely in hardware in the GPU, and they are fixed, while others are programmable through shaders. This is because the modern GPU has evolved around the graphics pipeline.

3D renderers were first implemented completely in software, which was very flexible, but also very slow. Then 3D acceleration appeared, and more and more of the rendering process was implemented in the hardware, which as a consequence made it much more inflexible. To get some of that flexibility back, parts of the pipeline were made programmable with shaders.

This chapter shows how each part of a shader maps to some part of the graphics pipeline.

Why Learn the Basics of Graphics APIs

You might have heard of OpenGL, Metal, Vulkan, and Direct3D.

They are all graphics APIs, and they have emerged on different platforms, and at different times, but they all share the common objective of providing a set of tools to render a 3D scene without starting from scratch, while taking advantage of hardware acceleration.

While it is certainly possible to write a software renderer from scratch, it hasn't been a viable enough (which mainly means fast enough, in this case) option to ship a game for at least a decade. While writing a software renderer is an enormously educational experience, it's not needed to ship games. Nowadays you don't even need to use graphics APIs directly, since the available game engines such as Unity wrap them for you with an ease that no one who isn't wielding a large team of graphics programmer can have.

In general, you don't want to deal with graphics APIs directly, unless you're developing a game engine. But it's very useful, and sometimes even necessary, in shader development, to know what lies beneath. If you aren't aware of the graphics pipeline and the graphics APIs, you are powerless to optimize your shaders and unable to debug the tricky problems that sometimes arise in shader development.

A General Structure of the Graphics Pipeline

The precise way in which the graphics pipeline is implemented, in some specific GPU, can vary a lot. But the general principles can be distilled and simplified into a version that works for our explanatory purposes.

Keep in mind that, while the principles hold in this example, the stage names and how the stages are divided may vary between different graphics APIs. There are some steps that are always going to be needed, but they may be broken down differently, or called with different names, depending on the documentation you're reading. Figure 3-1 shows the stages of a rendering pipeline (which is another name for graphics pipeline), which are then listed and explained.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_3

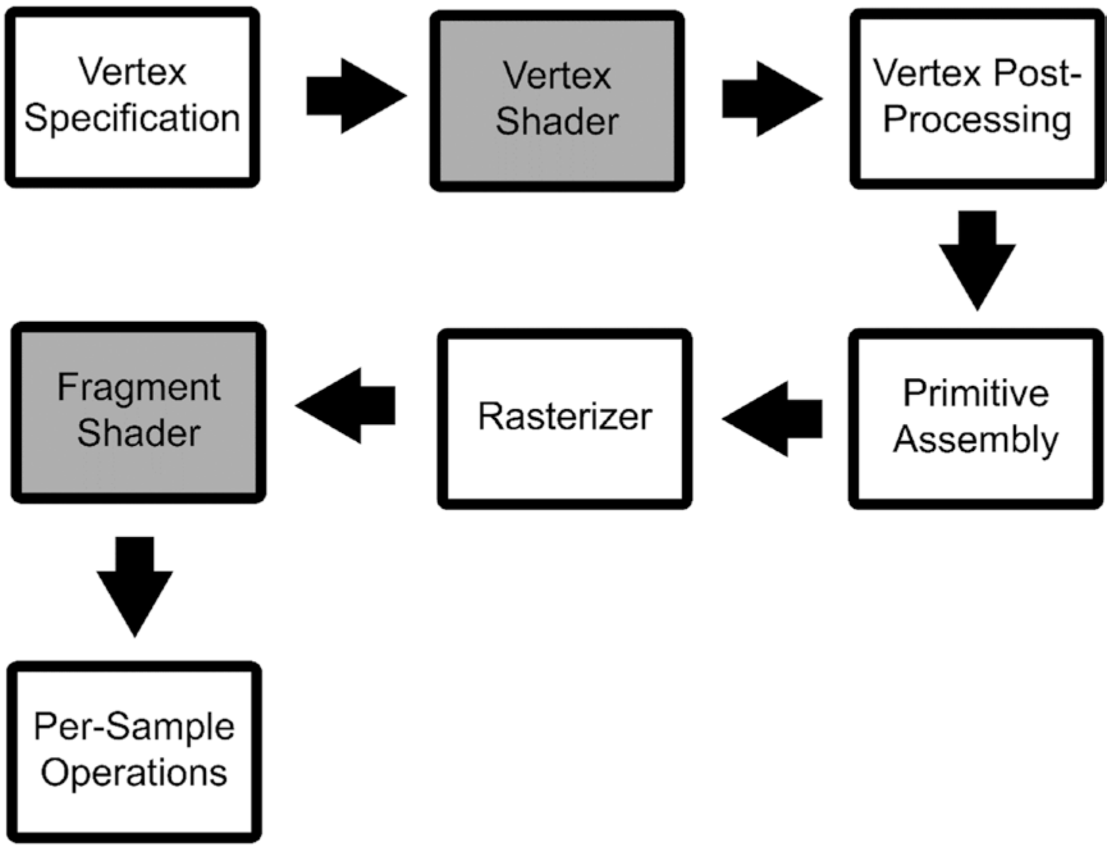

Figure 3-1. *An overview of the graphics pipeline*

The stages of an example graphics pipeline are as follows:

- The input assembly stage gathers data from the scene (meshes, textures, and materials) and organizes it to be used in the pipeline.
- The vertex processing stage gets the vertices and their info from the previous stage and executes the vertex shader on each of them. The main objective of the vertex shader used to be obtaining 2D coordinates out of vertices. In more recent API versions, that is left to a different, later stage.
- The vertex post-processing stage includes transformations between coordinate spaces and the clipping of primitives that are not going to end up on the screen.
- The primitive assembly stage gathers the data output by the vertex processing stages in a primitive and prepares it to be sent to the next stage.
- The rasterizer is not a programmable stage. It takes a triangle (three vertices and their data) and creates potential pixels (fragments) out of it. It also produces an interpolated version of the vertex attributes data for each of the fragments and a depth value.

- The fragment shader stage runs the fragment shader on all the fragments that the rasterizer produces. In order to calculate the color of a pixel, multiple fragments may be necessary (e.g., antialiasing).
- The output merger performs the visibility test that determine whether a fragment will be overwritten by a fragment in front of it. It also does other tests, such as the blending needed for transparency, and more.

This general overview is a mix of many different graphics pipelines, such as OpenGL and Direct3D 11. You might not find the same names in the specific one you want to use, but you'll see very similar patterns.

The Rasterizer

The rasterizer is an important part of rendering pipelines, and one that is normally not much discussed. It's a solved problem, so to speak, and it works quietly without attracting much attention.

It determines which pixels in the final image the triangle covers (see Figure 3-2). It also interpolates the different values that belong to each vertex (such as colors, UVs, or normals) over the pixels the triangle covers.

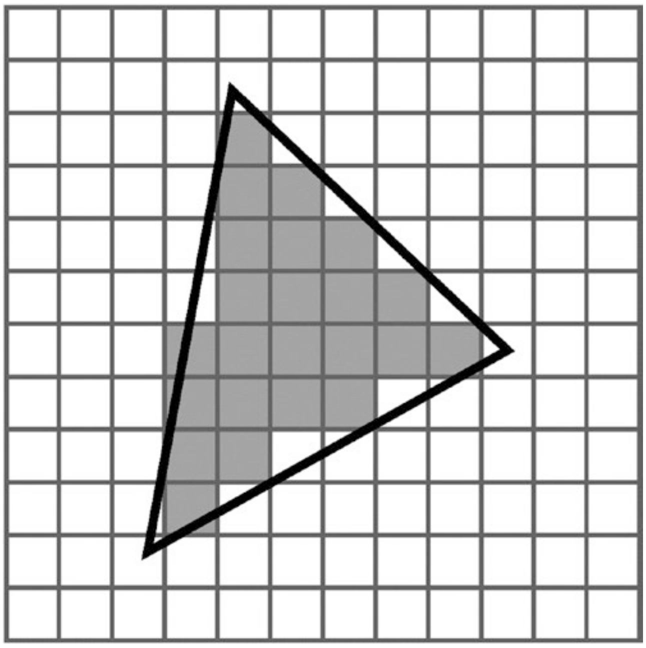

Figure 3-2. *The rasterizer determines which pixels a triangle covers*

The rasterizer is a fairly hidden bit of functionality and is easily overlooked. One of the clearest examples that can be devised to show you how it makes a difference is the interpolation of vertex colors. Imagine that you have a model (a simple quad, or even a triangle, will do) that has different vertex colors attached to its vertices. Figure 3-3 shows how that looks after it passes through the rasterizer.

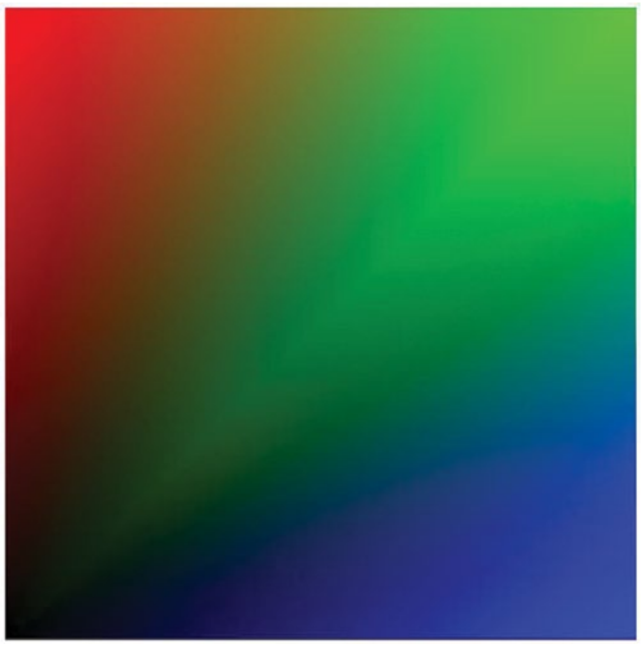

Figure 3-3. *Interpolation of vertex colors by the rasterizer*

Later in this chapter, you'll learn how to write the shader that will render this example.

The Structure of an Unlit Shader

As you saw in the last chapter, an Unlit shader revolves around the two shader functions and the two data structures used to pass information between them. Since their objective is to script parts of the graphics pipeline, they match some steps of the pipeline quite precisely. We're going to detail what they are in this section.

To summarize what you learned about Unlit shaders, Figure 3-4 shows an illustration representing their data flow. The vertex data is gathered into the `appdata` struct, which is passed to the vertex function. The vertex function fills in the members of the `v2f` data structure (which stands for vertex to fragment), which is passed as an argument to the fragment function. The fragment function returns the final color, which is a vertex of the four values—red, green, blue, and alpha.

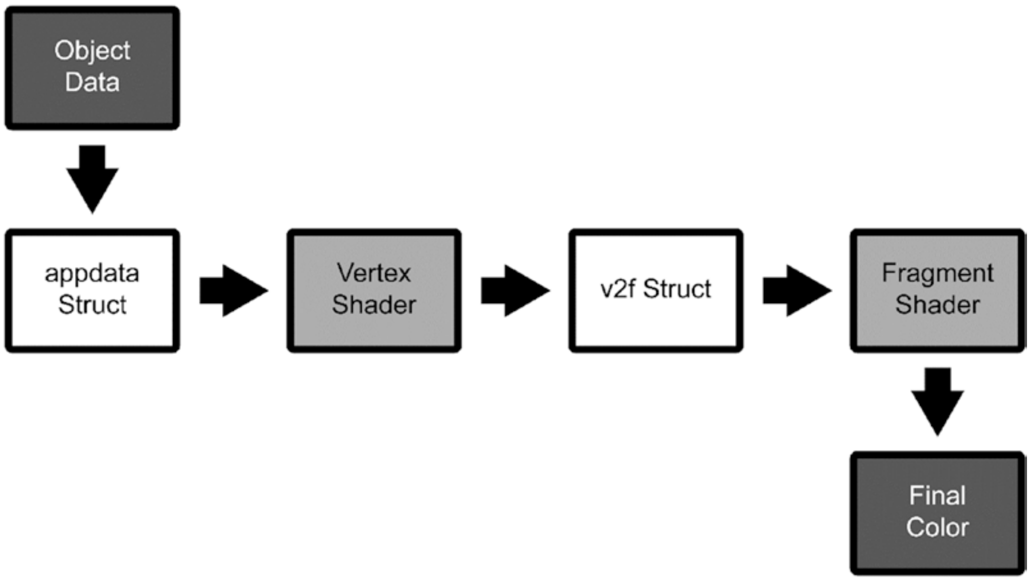

Figure 3-4. *The data flow of an Unlit shader*

In the last chapter, you wrote your first shader. It was very simple; it just rendered a model in a single color, without any lighting. We're going to use that example again to match some parts of the graphics pipeline to shaders, and then we'll also add to it, to better show the usefulness of the rasterizer.

Vertex Data Structure

The first structure was `appdata`, which corresponds to the input assembly stage. To freshen your memory, here is the `appdata` from the last shader:

```
struct appdata
{
    float4 vertex : POSITION;
};
```

`: POSITION` is a *shader semantic,* which is a string attached to a shader data structure that conveys information about the intended use of a member. By adding members with a semantic to `appdata`, we tell the input assembly stage what data we want out of what is available.

In this case, we only asked for the vertex position, which is the bare minimum you're always going to ask for. There is more data you can ask for, such as UV information to deal with texturing, or the vertex colors, should the model have them. The semantic and the data type **must** match. If you were to give it a single float for a `POSITION` semantic, the `float4` value would be truncated silently to `float`.

The data type chooses the "shape" of the variable, but the semantic chooses what goes inside it. Many kinds of semantic can fill a `float4`, but if we were to mistake which semantic we want, the shader would break at runtime, likely in subtle ways. Subtle shader breakage is one of the worst issues to track down once your shaders get complex, so be careful.

This is the only way you can gather data that changes per vertex, but you can pass other data in from properties and global properties that doesn't vary per vertex. OpenGL calls these per-vertex values, quite appropriately, *varying*, and the ones that you can pass globally, or a property in a shader, are called *uniform*.

Vertex Function

The next programmable stage is the vertex processing one, in which the vertex shader function is executed. It takes the appdata data structure (appdata) as an argument and returns the second type of data structure (v2f):

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    return o;
}
```

This is the bare minimum that needs to be done in a vertex shader: transforming the coordinates of the vertex to a coordinate space that can be used by the rasterizer. You do that by using the UnityObjectToClipPos function. For now, don't worry too much about coordinate spaces, as we explain them in detail in the next chapter.

Fragment Data Structure

Again, which members we include in the v2f data structure decides what data we can pass on from the vertex shader. This is the v2f struct from last chapter:

```
struct v2f
{
    float4 vertex : SV_POSITION;
};
```

It's very minimal and it only covers the 2D position of the vertex that was processed. We still need to get the semantic sort of right, but this data structure is less sensitive to mistakes. Keep in mind that semantics must not to be repeated. For example, you can't assign the SV_POSITION semantic to a second member as well.

Fragment Function

The next programmable stage is the fragment shading stage, where the fragment shader is executed on each fragment. In the very simple case from Chapter 2, the fragment shader was only using the position from the vertex.

```
float4 frag (v2f i) : SV_Target
{
    return _Color;
}
```

Actually, if you remember, that float4 wasn't used anywhere in the code. But if you try to get rid of it, you'll find that the shader doesn't render anything at all. That value is used by the graphics pipeline, even if you don't see it reflected in the shader code.

You may notice that the frag function has an output semantic, which we didn't mention in the previous chapter. That is used for specific techniques that we are not going to cover in this book. You should stick with SV_Target, which means that it outputs one fragment color.

This simple shader shows you that the conversion from 3D scene space and 2D target render space works, because we can indeed render a 3D model into a 2D screen. But it doesn't showcase clearly the role of the rasterizer. Let's add vertex colors support to this shader. For this, you'll need to use a mesh that does have vertex colors. One is included in the example source code for this chapter.

Adding Vertex Colors Support

Basically, we are adding one varying, one extra value attached to the vertex, which will need to be passed to the rasterizer, which will interpolate it.

Appdata Additions

When adding a member to appdata that is supposed to be filled with the vertex colors of the mesh (if the mesh has them), we need to pay attention to the name and semantic. The best bet is to use `color` as the name of the member and `COLOR` as semantic. Using only the `COLOR` semantic with a differently named variable might not work, depending on your platform. Here's how the structure should look after adding this vertex color member:

```
struct appdata
{
    float4 vertex : POSITION;
    float4 color : COLOR;
};
```

v2f Additions

In `v2f`, you need to add the exact same member. As before, this data structure can be less nitpicky, but you change this formula at your own risk. Here's how the structure should look after adding this vertex color member:

```
struct v2f
{
    float4 vertex : SV_POSITION;
    float4 color : COLOR;
};
```

Assign the Color in the Vertex Function

Having prepared the structures with the appropriate members, we add one line to the vertex function. It assigns whatever is in `appdata`'s color member to the `v2f`'s color member. The "magic" is here; it makes the vertex color data go through the rasterizer, which is going to interpolate the colors appropriately:

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.color = v.color;
    return o;
}
```

Use the Color in the Fragment Function

As in the previous shader, the fragment function is only preoccupied with returning one color, but in this case we ignore our property (which we can remove completely) and use the interpolated varying that is contained in v2f:

```
fixed4 frag (v2f i) : SV_Target
{
    return i.color;
}
```

Final Result

Figure 3-5 shows the result of our interpolated vertex colors shader.

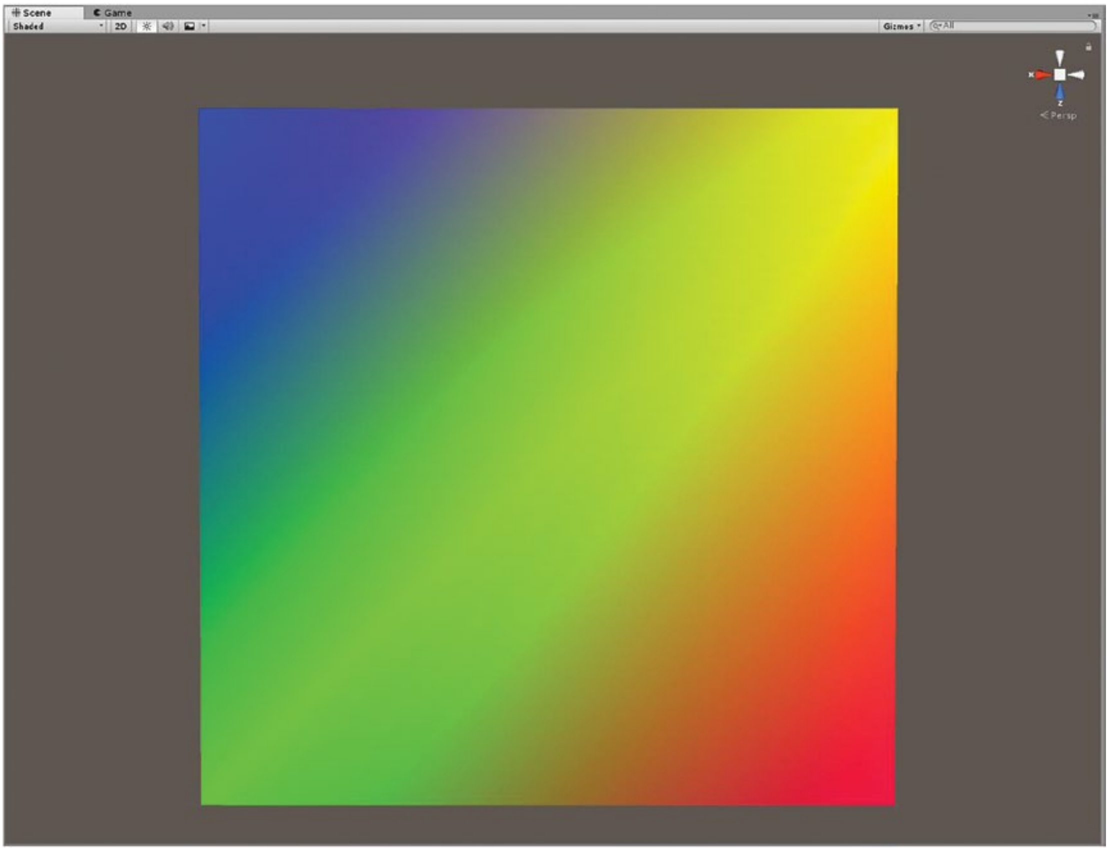

Figure 3-5. *The result of rendering this quad with vertex colors within a Unity scene*

Listing 3-1 shows is the final shader, after removing the color property.

Listing 3-1. The Final Shader That Showcases the Rasterizer's Contribution

```
Shader "Custom/RasterizerTestShader"
{
    SubShader
    {
        Tags { "RenderType"="Opaque" }

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;

            };
            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.color = v.color;
                return o;
            }
            float4 frag (v2f i) : SV_Target
            {
                return i.color;
            }
            ENDCG
        }
    }
}
```

Summary

This chapter explained the stages of the graphics pipeline, specifically which ones you can control through shader code. We demonstrated the otherwise hard to notice and uncelebrated role of the rasterizer with a new shader example.

Next

In the next chapter, we're going to tackle coordinate spaces in detail and connect them to where they are used within the graphics pipeline.