Graphics Pipeline Explanation

When looking at a computer screen, the image displayed comprises a collection of primitive types and vertices. Vertices are points somewhere in space, and primitives are made from these vertices and the faces that join them. We perform the same operation over and over on each vertex, allowing for the use of a pipeline architecture. The graphics pipeline takes vertices and passes them through the pipeline until they become pixels in the frame buffer; once one vertex moves from one step in the pipeline to the next, another vertex can enter and begin the process. The steps of the graphics pipeline consist of a vertex entering the vertex processor, then moving on to the clipper and primitive assembly, followed by the rasterizer, and finally the fragment processor completes the process and produces pixels that the user sees. Each piece of data must go through each step of the pipeline, even if it is simply passed onto the next step without any modifications. The two steps that are programmable are the vertex processor and the fragment processor; the other steps cannot be modified.

Each vertex is processed independently and begins in the vertex processor. This is where different attributes of the vertices are modified, such as coordinate transformations and computing color. Each vertex must be represented in terms of the coordinate system of the display. At a high level, the vertex processer receives vertex data as well as uniforms (constant data with respect to all vertices). After performing some computations to decide the final vertex position, the vertex processor produces varyings (new data for the fragment processor to consume).

The next step of the pipeline is the clipper and primitive assembly. Before clipping, we assemble vertices into primitives such as line segments or polygons because the clipper performs its operations primitive by primitive, not vertex by vertex. The reason we have the clipper is

because no imaging system can see the whole world at once, so we need to “clip” out the parts of the image that aren’t in view. The clipping volume denotes the frame of the image that we can see, and during the clipping process we keep the projections of objects that appear within the clipping volume and clip out the projections outside of the clipping volume. Objects that are on the edge of the clipping volume are partially visible in the image. The output of this process is a set of primitives whose projections can appear in the image.

The next phase of the graphics pipeline is the rasterizer. The purpose of the rasterizer is to convert vertices into pixels in the framebuffer. The output from the rasterizer is a set of fragments for each primitive; a fragment is a potential pixel in the framebuffer that can also carry information about depth with it.

The last element of the graphics pipeline is the fragment processor. Like the vertex processor, the fragment processor processes fragments one at a time. The fragments generated by the rasterizer enter the fragment processor and the output is updated pixels in the framebuffer. Some of the effects produced by the fragment processor include texture or bump mapping and blending color to make a translucent effect. The fragment processor has access to uniform variables that act as a constant for every fragment. These constants can act as lookups for texture, or lookaheads for prefetches, and they can help predict branching. The fragment processor also receives varyings as input. These are read-only values that were produced by the vertex shader to be consumed for our fragments in the fragment processor. In other words, the vertex processor does some sort of data computation that doesn’t get utilized until the fragment processor; an example of a varying would be vertex color. After all pixels have been updated in the frame buffer, we have our image displayed.

A pipeline architecture makes sense in computer graphics as we are continuously performing the same actions on sets of data to convert vertices and primitives into pixels on a screen. The pipeline architecture further offers a more efficient process for rendering images as we do not need to wait for a single set of data to go through the entire pipeline before sending in more data.

Sources

- https://gamedevelopment.tutsplus.com/tutorials/getting-started-in-webgl-part-1-introduction-to-shaders--cms-26208
- https://s3.us-east-1.amazonaws.com/blackboard.learn.xythos.prod/5a319f5d041da/1276323?response-content-disposition=inline%3B%20filename%2A%3DUTF-8%27%27Assignment1Readings.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20180910T174553Z&X-Amz-SignedHeaders=host&X-Amz-Expires=21600&X-Amz-Credential=AKIAIL7WQYDOOHAZJGWQ%2F20180910%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=4428696e860bfae46a69a1926c2c6cf05a43ac00aa335722de8a6ff602549247