Adam Resnick Graphics Pipeline Essay 9/8/18

The graphics pipeline is an optimized model to render discrete 2D pixels on a screen from vertex data that may be in 3 dimensions and contain other “vertex parameters” [1][2].

Early in the history of computer graphics, the graphics pipeline was fixed and all interactions between applications and hardware were abstracted away from developers and designers by massive APIs [2]. This system prevented developers from being able to create applications that used functionality outside the existing API specifications, and each time that new functionality was required, hardware manufacturers would have to add it to their supported API, making complex and customized design difficult [2]. In the late 1980’s to 1990’s a more programmable model of the graphics pipeline was developed with programmable shaders and customized components, allowing developers to run programs more directly on the graphics hardware of a computer [2]. Shaders are small programs that run directly on graphics hardware, which take in graphics information and complete processing (customized by the developer specifically for each application) and output more developed graphics information to the next stage of the graphics pipeline [2].

Overall, the graphics pipeline takes in data as vertices in 3D vector space, along with colors, textures, and any other associated vertex data and outputs pixel values for each location in the framebuffer [3]. First, the vertex shader takes the input vertices and completes all processing necessary to transform “vertices to their final locations on the screen” [2]. This step is required but highly programmable and can include processing to account for projections of images through a perspective, scaling, or rotation [2]. Next the clipper and assembler deletes pixels that will be unseen by the user to prevent unnecessary computation as well as collects vertex data and forms primitives from vertices [1][3]. Primitives are simple structures made up of vertices such as lines, points, or triangles, which make up the structure of an image connecting the individual points that compose the image [3]. Next, the rasterizer converts the primitives that make up the image into a sequence of fragments [4]. Fragments are pieces of data that contain information necessary for the final rendering of pixels [2]. Sometimes a final pixel will depend on multiple fragments containing information about color, depth, opacity, textures, etc [2][4]. These fragments “may or may not wind up” in the final image, and can be thought of as “potential pixels” [1][4]. Next, the information contained in the fragments is run through the fragment shader. The fragment shader processes each fragment individually and applies “per-fragment” operations for values such as color, texture, opacity, among others [4]. The output of the fragment shader is a list of colors for each pixel that are fit into the frame buffer, as well as a depth value for each pixel [3]. The frame buffer is the location in physical memory where pixel values are held [1] The output from the fragment processor, which is the final, processed pixels values are placed in this buffer to be read onto the screen.

The parallel nature of the pipeline is that each step that is completed by a shader or other component can be run immediately when the step before is completed [2]. In this way, data is constantly flowing through the pipeline, transferring data from one component of the pipeline and transferring it as an input to another [2]. Due to the “Unified Shader Model” the same pieces of hardware are now able to act as both vertex and fragment shaders on demand, further allowing cores to be allocated to the most pressing computational step and helping data to move quickly through the system [2].

Sources:

[1] <https://www.pcmag.com/encyclopedia/term/43933/graphics-pipeline>

[2] <https://gamedevelopment.tutsplus.com/tutorials/getting-started-in-webgl-part-1-introduction-to-shaders--cms-26208>

[3] <https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview>

[4] <http://graphics.cs.ucdavis.edu/~joy/ecs175/Lectures/Lecture2.pdf>