

The Low Down on Graphics Pipelines

Phil Monroe

Jan. 17, 2011

Graphics processing units (GPUs) are becoming increasingly important in modern computing. Driven by video games, today's computer software demands higher performance and stunning visuals from GPUs. GPUs start out with several different objects and combine them to form an image in a series of stages called the graphics pipeline. Consider an image where there is a soccer ball sitting on a table inside of a room. This article will describe how a computer creates this scene using the graphics pipeline.

The graphics pipeline begins with the *application* stage where running software interacts with the GPU. The first step of this stage is to build three-dimensional models of each object present in the scene. The 3D models of each object are created using a mesh of triangles. To help visualize this mesh, consider the soccer ball. Real soccer balls are patches of hexagons and pentagons sewn together to form a round ball. So why use triangles instead of hexagons and pentagons to create objects? Well it turns out that complex geometries, such as the hexagons and pentagons, can be broken down into triangles. Triangles also have mathematical properties that make processing the models quicker. Each triangle is defined by the three points that connect the sides, which are known as the vertices. To make these meshes of triangles more realistic looking, textures, which are images that seem to cover the 3D model like a skin, are created and will be applied later in the pipeline.

It is important to note that each 3D model is in its own coordinate space called model space; Therefore, each model of an object has absolutely no information about any of the other objects. Our soccer ball is floating in its own universe where it is the only thing that exists. Now the application needs to transfer all models to the GPU to assemble into the scene. Along with the models, the application passes information on how the models are placed within the scene, the textures they have, where the lights are and at what direction the user is looking at the models.

Now that the GPU knows about every object present, it can start piecing together the scene in what is called the *geometry* stage. The

first step in the geometry stage is to combine all of the models into a common world. This is done by using mathematical transformations to move the objects from model space into what is called world space. After the transformations, all objects lie in the same universe and are spaced properly.

At this point, the GPU can take lighting into consideration. Using the application defined light sources, the GPU can begin to calculate the shadows caused by each of the objects in the world.

It is now time to consider the user's viewpoint, which is called the camera. Much like a real camera, we place the camera in the scene to decide what will be visible to the user. At this point, the GPU can start trimming away triangles outside of the camera's rectangular viewing area in a process called clipping. Clipping also removes rear facing triangles, such as the triangles on the non-visible side of the soccer ball. Removing these triangles reduces the amount of processing required in future steps.

Now we need to start making the scene two dimensional. This is accomplished by taking each object and projecting the 3D mesh of visible triangles onto a 2D plane, which produces 2D objects at various distances, or depth, away from the camera. Visualize this as if we were creating a cartoon out of construction paper. We would first create each of the characters and objects present and form layers to determine depth within the scene.

To perform most of the transformations in this stage, the GPU uses a special *programmable shader* called the vertex shader. A programmable shader is a customizable piece of hardware that modifies data based off a recipe defined by the software developer. The vertex shader focuses on getting the geometry right in the 3D to 2D transformation. This is highly customizable to allow developers to shape the world exactly as they see fit.

Next, consider what our scene will be displayed on: the screen. Screens are two-dimensional grids of colored dots called pixels. Think of them as a sheet of graph paper where you can only color in the individual squares to draw a picture. If the squares are small enough, images can be drawn with very high detail. So how are we going to convert layers of 2D triangles into a bunch of pixels? This conversion process is called *rasterization*. The rasterization process is similar to taking all

the 2D objects in our scene, laying them out on a sheet of graph paper representing the screen, and filling in each object with little squares. Each little square within the blocky rasterized object is called a fragment and has all the information, such as position, initial color and opacity, needed to draw that square on the screen. At the end of the rasterization process, every object in the scene will be converted into boxy rasterized objects at varying depths.

After rasterization, *texture* is applied to the blocky objects to make them seem more realistic. As hinted to earlier, textures are images that are wrapped around objects to provide a skin. While that is a convenient way of thinking of textures, that is not completely accurate. Textures are 2D images that show all sides of an object. Consider a real soccer ball with scuffs and grass stains that we want our digital soccer ball to look like. A texture would be made to appear as if the ball was cut up and laid flat in one piece. The pixels in the texture can now be overlaid onto the rasterized object to give more realistic coloring. During this process, the GPU may warp the texture to give perspective and make the object visually appear 3D.

At this point, developers generally define another custom *programmable shader* called the pixel or fragment shader. The pixel shader is used to define the final color for a fragment and to create special coloring effects.

Now that all of our objects have been rasterized, colored and textured to look realistic, we need to combine the layers in the *composition* stage to form a solid image. The GPU uses a structure called the frame buffer to hold all the final pixels to be displayed on the screen. One by one, each object's fragments are copied over to the frame buffer. If two fragments overlap, then the fragment closest to the viewer will be saved in the frame buffer. For transparency, overlapping fragments will be added together to simulate light passing through.

To display the final image, individual pixels are sent to the screen by reading from the frame buffer. Starting with the top left corner, each pixel on the screen is updated from left to right and top to bottom until the bottom right corner is reached. The GPU then repeats the whole process over to display the next image. For movies and movements within a world, the entire pipeline is run multiple times per second to create the illusion of continuous movement.

References

- **EEC 277 Lectures 1+2 plus slides.**
- **David Luebke and Greg Humphrey - How GPUs Work**
<http://bit.ly/hHt4VH>
- **Jason L. McKesson - Learning Modern 3D Graphics Programming - Graphics and Rendering**
<http://bit.ly/kpxL01>
- **Wikipedia - Graphics Pipeline**
http://en.wikipedia.org/wiki/Graphics_pipeline
- **Wikipedia - Shaders**
<http://en.wikipedia.org/wiki/Shader>
- **Eric Sink - It's All About Triangles**
http://www.ericssink.com/wpf3d/5_Triangles.html