The Graphics Pipeline

Introduction

We have currently been using the following general algorithm for rendering our scene:

```
for (pixel : image)
{
    Ray r = computeRay();
    ShadeRec rc;
    for (object : geometries)
    {
        if (object.intersect(r, rc))
        {
            pixel = shade(rc);
        }
    }
}
```

The question now is the following: presumably we will have a smaller number of primitives in our scene than the total number of pixels in the image. This is true in general, though for *very* dense scenes this may no longer be the case. Since we know before we start the positions of all objects in world space, it would be easier to determine which pixels each object is covering in our image and then shade the pixels in whichever way we want. In other words, we could be looking at an algorithm that resembles this:

```
std::vector<Pixel> pixels;
for (object : scene)
{
    pixels += pixelsCoveredBy(object);
}

for (pixel : pixels)
{
    pixel = shade(pixel);
}
```

In the worst case, we can assume that *every* pixel is covered by an object. In this case, we end up in a similar position as ray tracing, but with the big advantage that not every object covers every pixel.

In essence, what we are doing is determining which "pixels" are covered by which objects, and then shading them. In order to make the distinction between the pixels that each object covers and the final pixels in the image, we will refer to the pixels covered by objects as *fragments*. Once we have the list of fragments per-object, we just assign the corresponding pixels in the image to the colours and we are done.

This process is called *rasterization*, since we are simply turning objects into a set of fragments. This process is the foundation of how real-time graphics systems work, and they are also the basis for the *graphics pipeline*.

Definitions

Before we define what the graphics pipeline is, we first need to define what a pipeline is. In short, a *pipeline* is a sequence of events where the output of one stage becomes the input of the next stage. The graphics pipeline is then a sequence of stages that takes as its initial input a series of objects and renders them into the final image. One of the main characteristics of the pipeline is that each stage is able to process its inputs in *parallel*. In other words, if a stage receives n inputs, then it can do all of them at once. This is important to keep in mind as we move forward in our discussions on the pipeline.

The Graphics Pipeline

The graphics pipeline is formally defined into the following stages:

1. Application,
2. Geometry processing,
3. Rasterization,
4. Pixel processing.

Application

The application is the one that ultimately drives the entire pipeline and is in charge of preparing and setting data for the rest of the pipeline. For us, the application stage is our C++ code. This stage also comprises things like physics engines, collision detection, loading and unloading of assets, etc.

The application is the only portion of the pipeline where there are no "inputs" (or at least not formally). The application stage can generate the inputs for the geometry processing stage if need be, or it can receive the inputs from files (such as mesh files, textures, etc).

Geometry Processing

The input of the geometry processing stage is, as its name implies, *geometry*. The important part is understanding *how* the geometry is represented. In our ray tracers, we could define our geometries through implicit functions. The notable exception were triangles, where they were defined as a series of three vertices. Due to some limitations caused by *where* the pipeline exists, implicit functions cannot be used (at least not directly). This means that the geometry stage will require the geometries in a specific format. In particular, the geometries have to be expressed as a stream of vertex data (that can contain things like position, normals, etc) and the type of geometry that the vertices represent. In general, all the types of primitives are ultimately triangles or compositions of triangles. A partial list of geometries is:

- **Points:** every vertex is a point,
- **Lines:** every pair of vertices represents a line.
- **Triangles:** every triplet of vertices represents a triangle.
- **Triangle Fan:** a series of triangles that all share one vertex.
- **Triangle Strip:** a series of triangles that all share one edge (or two vertices).

Therefore, the application stage will provide a list of vertices and the type of primitive that they represent as the input to the geometry processing stage. This stage is itself comprised of 3 distinct steps:

1. Vertex shaders,
2. Clipping,
3. Screen mapping.

The vertex shader will receive the stream of vertices. The first problem is that we need to somehow convert our primitives into the appropriate space. Recall that the vertices represent geometries, and are therefore defined in model space. In order to figure out which pixels these geometries cover, we first need to take them into *screen space* (or *screen coordinates*). Intuitively, these are the *2D* coordinates of the objects with respect to the screen. So, the question becomes: how do we get from model space into screen space?

The MVP Matrix When we discussed transformations, we mentioned that matrices were also a way of mapping one coordinate space into another. We saw the example of the *model matrix*, that could take us from model space into world space. We will now refer to this matrix as M. Once we are in world space, we now need to take into account our camera. Suppose we are using a pinhole camera as previously defined. Before, we computed the orthonormal basis and then used it to compute the rays. What we now have are vertices in world space that need to be converted into camera space. What we will use is a matrix called the *view matrix* (or *camera matrix*) to take us there. This matrix is computed based on the orthonormal basis, as we will see later. Let this matrix be V.

The final step is to handle the type of projection that we are using. In ray tracing, this was handled automatically by the camera, but recall that both perspective and orthographic projections define a *viewing frustrum*, which ultimately defines all that is visible in the scene. Furthermore, this volume is now also conformed by the *near* and *far* planes that we discussed previously. Likewise to the camera matrix, we will use another matrix called the *projection matrix* to take us from camera space into *clipping space*. Let this matrix be P.

This means that in order to get a vertex v from model space into clipping space, we need to:

- Convert into world space with M,
- Convert into camera space with V, and
- Convert into clipping space with P.

These three matrices together are referred to as the MVP matrix.

For the time being, we will let the vertex shaders handle the transformation from model space into clipping space. Once we have the vertices in this space, they can now be forwarded to the clipping stage.

Clipping At this point, we know which primitives are inside, outside, and partway in the viewing frustrum. Therefore, what we will now do is *cull* (or remove) all primitives and portions thereof that are *not* in the viewing volume. If a primitive is sliced by the viewing volume, we will cut it and add new vertices as needed. Once this is done, we can forward into the final stage: screen mapping. It is worth noting that this is the final stage that is in 3D space.

Screen Mapping Screen mapping is in charge of projecting the points onto the screen (or viewing plane from before). In the process, the z coordinate of all vertices will be clamped in the range $[0, 1]$ and will be stored along with the points. This is the last stage of the geometry processing block and the data is now forwarded into the rasterizer.

Rasterization

This stage has only one job: take the triangles in screen space and determine which pixels they cover. This is achieved by sampling the triangles and is divided into two stages:

- Triangle setup: in charge of setting up all the data structures and information required so we can traverse the triangles in an efficient manner.
- Triangle traversal: walk through every triangle and check to see if a pixel is covered by it. If it is, mark it as covered and save it to a list for the next stage.

The rasterization stage involves complex algorithms that determine in a *very* efficient way whether a given pixel is covered by them. As most of this is handled externally, we will not deal with them beyond knowing that they are there. Once this stage is done, we are left with a list of fragments that are covered by the geometries. Along with this, we also know *which* geometries cover the fragments and the z value that we computed at the screen mapping stage. Any other data that was forwarded in (such as normals) will be *interpolated* at this stage.

Pixel Processing

The final stage of the pipeline, which is ultimately in charge of computing the final colour of the pixels on the screen. It is divided into the following steps:

- Fragment shading: in charge of computing the colour of each fragment via whatever equations we require. Note that since we have all the interpolated data at this point, we can use our equations as in ray tracing.
- Merging: takes the output of the fragment shader and merges it to determine the final colour of the pixel. It is this stage that uses the z coordinate that has been passed down the pipeline in a process called *z-buffer*, which we will discuss later.

Once everything is done, we are left with an image that can be displayed to the screen.

As you might expect, the entire pipeline involves many complex algorithms, all of which need to be optimized in order to run as fast as possible so that users see the results in real-time. Fortunately for us, every stage of the pipeline (with the exception of the application stage) is controlled by the GPU.