Shaders Notes

Introduction

Shaders = algorithms running in the programmable stages of the pipeline. There are different types of pipelines, we will focus on **graphic pipelines**, which are meant to provide rendering of 3D meshes, or in other words they help rendering an image from a set of 3D primitives.

Shaders

When creating a pipeline, the user has to:

- Specify parameters to control the fixed functions.
- Provide shaders for the programmable stages.

All shaders have very similar interfaces, and are connected in a common way. They are written mostly in GLSL (openGL Shading Language). We'll work with two types of shaders:

- `.vert`: vertex shaders
- `.frag`: fragment shaders

How do they work

Vertex shaders Vertex shaders are then executed to perform operations on each vertex. Such operations, for example, transform local coordinates to clipping coordinates by multiplying vertex positions with the corresponding WVP matrix, or compute colors and other values associated to vertices, which will be used in later stages of the process.

Fragment shaders The final color of each fragment is determined by a user defined function contained in the Fragment shader. This section will use either physically based models, or other artistic technique to produce either realistic or effective images.

Fixed functions

Several important actions occurs in the final fixed sections of the pipeline. We will enter in detail in the following functions:

- Primitives clipping
- Back-face culling
- Depth testing (z-buffer)
- Stencil

Clipping The triangles of a mesh can intersect the screen boundaries and can be only partially shown. As we have briefly introduced, clipping is the process

of truncating and eliminating parts of graphics primitives to make them entirely contained on the screen.

Clipping is usually performed after the projection transform, but before normalization (division by w). For this reason, the space in which it is performed is called **Clipping Coordinates**.

GLSL

Program structure Shaders follow the classical convention, having global variables and functions. The entry point of the shader can be user defined in the code calling it, however it is generally called `main()`. It's syntax is very C-like, for example the following is a `.vert` shader:

```
// required language version is 4.5
#version 450

/* those layout(...) are quantifiers
required to interface with the pipeline */
/* communication between the Shaders and the
application occurs using Uniform Variables Blocks */
layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

/* communication between the Shaders and the Pipeline
occurs through global variables, such as those: */
layout(location = 0) in vec3 inPosition;

/* in and out variables are used to interface with the
programmable or configurable part of the pipeline. */
layout(location = 0) out float real;
layout(location = 1) out float img;

void main() {
  /* gl_Position    is a vec4 variable that must be filled with
  the clipping coordinates of the corresponding vertex */
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```