Chapter 7 OpenGL Part II

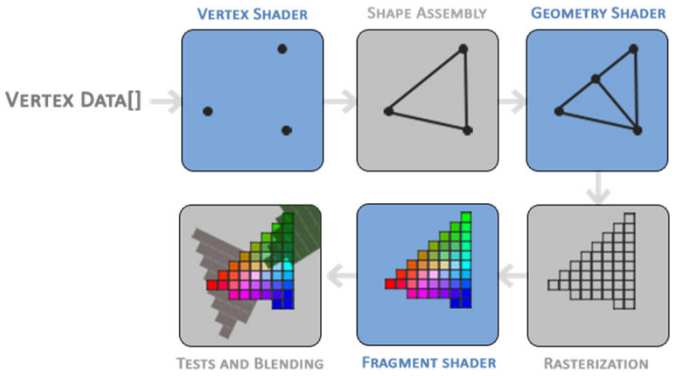
- Hello Triangle
- Shaders
- Transformations
- Coordinate Systems
- Camera
- Summery

Hello Triangle

- In OpenGL everything is in **3D space**, but the screen or window is a **2D array of pixels** so a large part of OpenGL's work is about transforming all 3D coordinates to 2D pixels that fit on your screen.
- The process of transforming 3D coordinates to 2D pixels is managed by the **graphics pipeline** of OpenGL.

Hello Triangle-cont.

- Note that the **blue** sections represent sections where we can inject our own shaders.



124

Shaders

- **Shaders** are little programs that rest on the **GPU**. These programs are run for each specific section of the graphics pipeline.
- In a basic sense, shaders are nothing more than programs transforming inputs to outputs.
- Shaders are also very **isolated** programs in that they're not allowed to communicate with each other; the only communication they have is via their inputs and outputs.

125

GLSL

- Shaders are written in the **C-like** language **GLSL**. GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.
- Shaders always begin with a **version declaration**, followed by a list of **input** and **output** variables, uniforms and its main function.
- Each shader's entry point is at its **main** function where we process any input variables and output the results in its output variables. Don't worry if you don't know what uniforms are, we'll get to those shortly.

126

Vertex shader

- A **vertex shader** is a graphics processing function used to add special effects to objects in a 3D environment by performing mathematical operations on the objects' vertex data. Each vertex can be defined by many different variables.

```
#version 330 core
layout (location = 0) in vec3 aPos; // position has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0); // we give a vec3 to vec4's constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // output variable to dark-red
}
```

127

Fragment shader

- A **Fragment Shader** is the Shader stage that will process a Fragment generated by the Rasterization into a set of colors and a single depth value.

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // input variable from vs (same name and type)

void main()
{
    FragColor = vertexColor;
}
```

128

Transformations

- Luckily, there is an easy-to-use and tailored-for-OpenGL mathematics library called **GLM**.
- Most of GLM's functionality that we need can be found in 3 headers files that we'll include as follows:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

129

Transformations-cont.

- We add the following lines to the vertex shader code

```
uniform mat4 transform;
```

```
gl_Position = transform * vec4(aPos, 1.0f);
```

- Here is how we use **glm** in action to rotate and scale

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

130

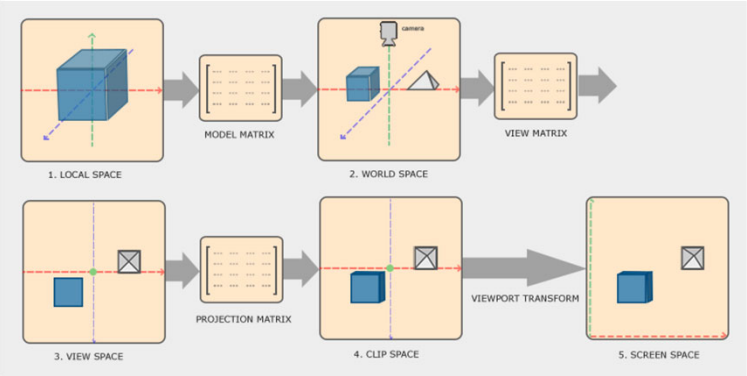
Coordinate Systems

- There are a total of 5 different coordinate systems that are of importance to us:
  1. Local space (or Object space)
  2. World space
  3. View space (or Eye space)
  4. Clip space
  5. Screen space
- Those are all a different state at which our **vertices** will be transformed in before finally ending up as **fragments**.

131

Coordinate Systems-cont.

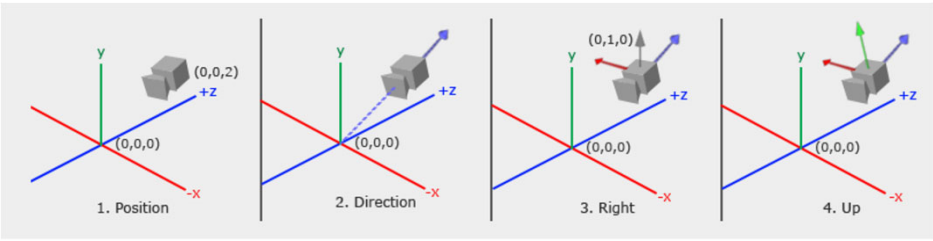
- To transform the coordinates from one space to the next coordinate space we'll use several transformation matrices of which the most important are the **model**, **view** and **projection** matrix.



132

Camera

- **OpenGL** by itself is not familiar with the concept of a **camera**, but we can try to simulate one by moving all objects in the scene in the **reverse** direction, giving the illusion that we are moving.
- To define a camera we need its **position** in world space, the **direction** it's looking at, a vector pointing to the **right** and a vector pointing **upwards** from the camera



133

Summery

- **OpenGL**: a formal specification of a graphics API that defines the layout and output of each function.
- **Viewport**: the 2D window region where we render to.
- **Graphics Pipeline**: the entire process vertices have to walk through before ending up as one or more pixels on the screen.
- **Shader**: a small program that runs on the graphics card. Several stages of the graphics pipeline can use user-made shaders to replace existing functionality.
- **Vertex**: a collection of data that represent a single point.
- **Vector**: a mathematical entity that defines directions and/or positions in any dimension.
- **Matrix**: a rectangular array of mathematical expressions with useful transformation properties.

134

Summery-cont.

- **GLM**: a mathematics library tailored for OpenGL.
- **Local Space**: the space an object begins in. All coordinates relative to an object's origin.
- **World Space**: all coordinates relative to a global origin.
- **View Space**: all coordinates as viewed from a camera's perspective.
- **Clip Space**: all coordinates as viewed from the camera's perspective but with projection applied. This is the space the vertex coordinates should end up in, as output of the vertex shader. OpenGL does the rest (clipping/perspective division).
- **Screen Space**: all coordinates as viewed from the screen. Coordinates range from 0 to screen width/height.
- **LookAt**: a special type of view matrix that creates a coordinate system where all coordinates are rotated and translated in such a way that the user is looking at a given target from a given position.

135