IMAT2908 CW No 1

|  |  |
| --- | --- |
| **Full Name: Tia Mulholland** | **P\_Number: P2433472** |
| **GitHub ID: Asia292** | **Word Count: 510** |

## Abstract

The main aim of this piece of work is to use knowledge of the rendering pipeline to create an interpolation shader of an unfolded coloured cube and kite. Alongside this I created other shapes and created a function that allows switching between them.

## Introduction

OpenGL uses the rendering pipeline whenever it renders an object. There’s a total of seven steps that it goes through in order to achieve a fully rendered object – vertex specification, vertex processing, vertex post-processing, primitive assembly, rasterization, fragment processing and pre-sample operations.

Vertex specification handles the vertices that will be passed to the pipeline itself. It also deals with the VAO and VBO.

Vertex, tessellation and geometry shaders are all be defined within vertex processing. All stages here are programable; each one representing different shaders.

Vertex post-processing puts the outputs of the previous stage through operations including transform feedback – allows the user to transform the data via buffer objects – and clipping – ensures that no primitives lie on the edge of the viewing plane.

The type of primitive used alters the way primitive assembly works, but it still composes the data from previous stages into primitives. The pipeline can be stopped at this stage to allow the use of transform feedback.

Primitives are then rasterized to create fragments. These fragments are used to create the pixel data.

Fragment processing involves using a fragment shader to process the output from rasterization. The result from this process is a list of colours, a depth value and a stencil value.

Pre-sample operations consists of several steps. The first is a set of tests that are activated by the user – pixel ownership, scissor, stencil and depth tests. Colour blending then takes place, combining the colour output from the fragment shader with that in the colour buffer. Finally, the fragment data is written to the frame buffer.

## Method

Each shape follows the same setup with the only difference being the coordinates and colours passed in. The vertex shader is defined first; data is loaded from a file which is then passed to the created vertex shader and then compiled. A similar method is used for the fragment shader but with its own source code. After this the two shaders are linked to create a usable output. The points and colours are defined within arrays along with the final drawing order. This final array is used to set up the VAO and buffers before the outcome is rendered.

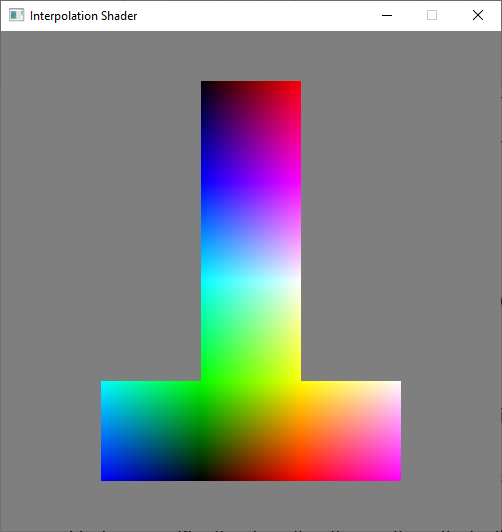
In order to switch scenes, the programme listens for certain key releases. If one is released that it is listening for, it creates a new scene of its target shape and initialises it. To achieve this each shape has its own class and creates its own VAO and VBO.

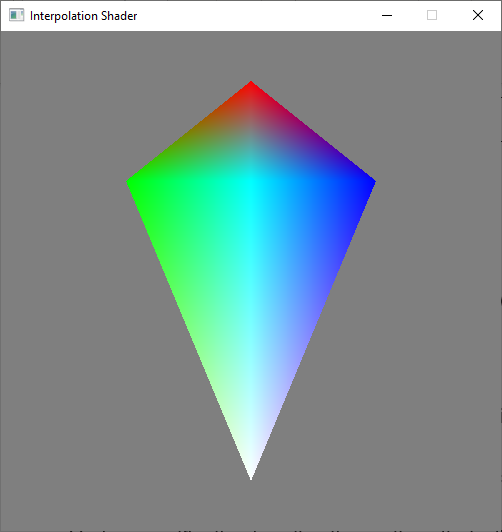
## Evaluation

Overall, I achieved everything I had planned to. However, the scene switching could have been done in a better way. In the current format, each shape has its own class, and, calling creates a new scene. This is fine for a small project but on a larger scale this would be a big load on memory and performance.

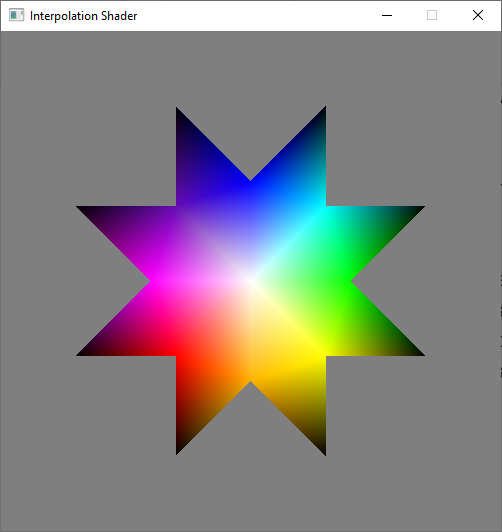
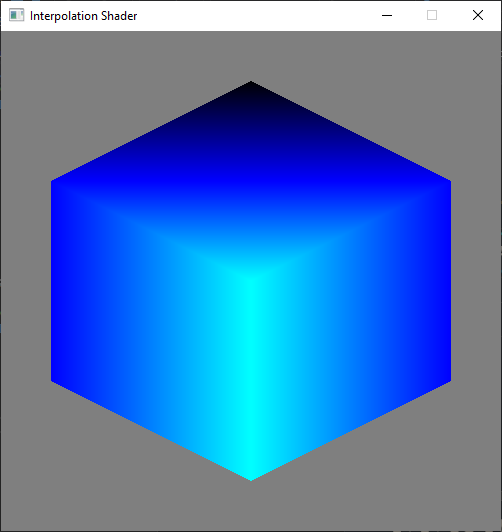
## References

OpenGL Wiki. (8 April 2019). *Rendering Pipeline Overview* [Online]. Available: https://www.khronos.org/opengl/wiki/Rendering\_Pipeline\_Overview

RGBCubeScene RGBKiteScene



Shape cube3D



Switch shape function

