***Assignment 1 Written Report***

Transforms

How they work

In their simplest form, transforms are a way of storing an entities location, rotation and scale in a graphical world space. They are stored in a 4x4 matrix, within which each value correlates to an aspect of and objects transform. Such examples include an objects x axis position, x axis rotation and x axis scale. With a complete matrix a program can know and key basics of an object’s information. By combining and calculating an objects transform with other matrices, such as translation and rotation, we are able to move, rotate and scale objects to suit our needs.

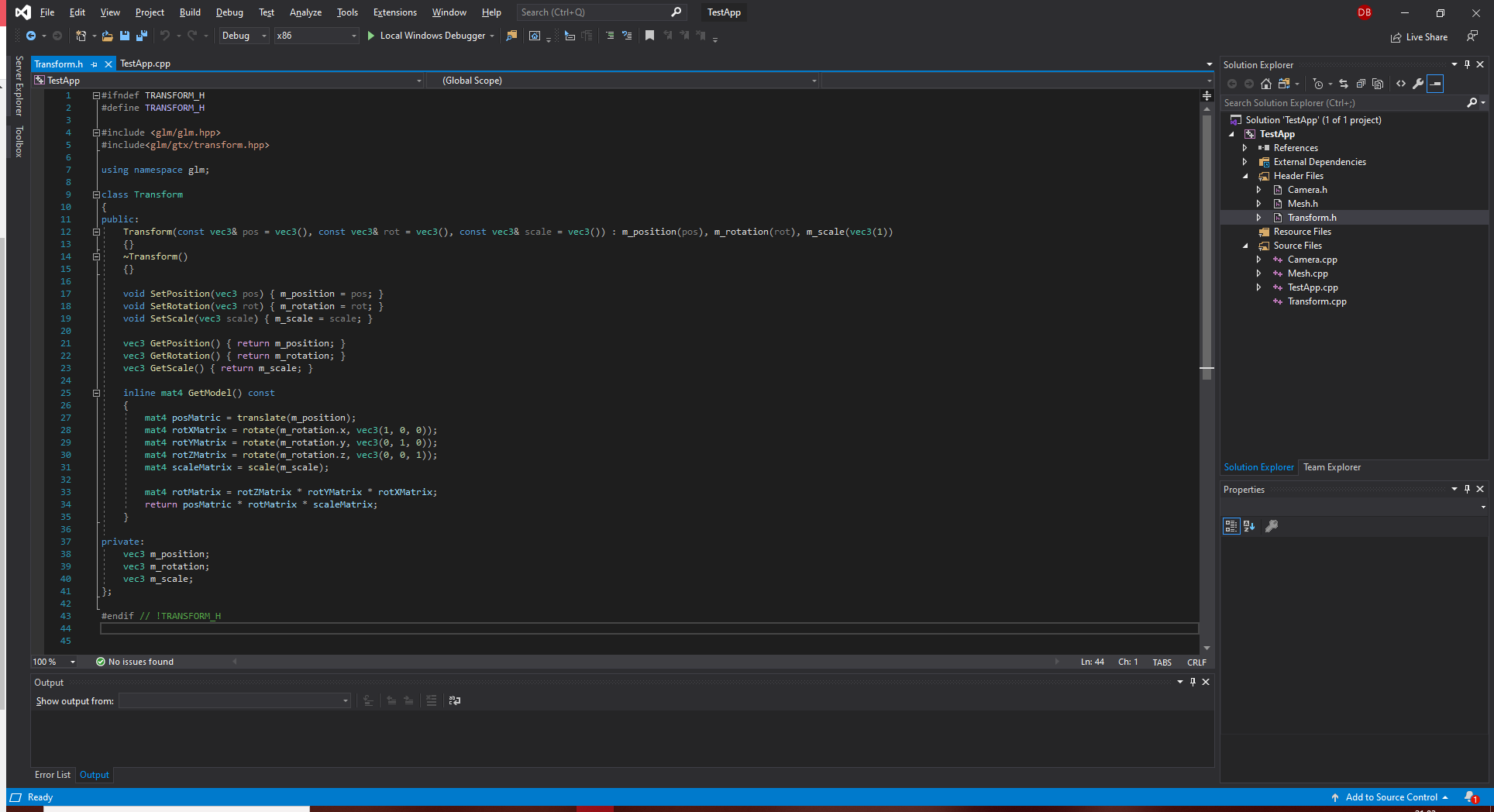
Manipulating a transform with what is known as a “translation matrix” allows for movement of points while retaining shape, size and rotation of the points created. This is calculated as the objects current x, y and z co-ordinates added to the matrix’s x, y and z co-ordinates respectively for each point in the object. The ability to manipulate an objects transform in this way allows for the object to be accurately moved around the world space, for purposes such as a player’s movement within a game.

Alternatively, by manipulating a transform with what is known as a “rotation matrix” such as Real-Time Rendering’s example (See Equation 4.3, 4.4 and 4.5) (Moller T, 2008, p. 57) allows for rotation of shape points the vertices, either around the objects centre or around a specific world location. When dealing with rotation matrices it is important to make sure that the order in which the axis rotation is multiplied is the reverse of the expected x, y, z layout. Without this knowledge the rotation will not occur correctly and provide unexpected results. This becomes useful when objects in games need to turn or otherwise face in a new direction. Similarly, to a translation matrix a rotation matrix will maintain the distance between an object’s points, Real Time Rendering refers to transforms which do this as a “rigid-body transform” (Moller T, 2008, p. 56). When used in conjunction with a translation matrix, rotation matrixes allow for movement and re-orientation of an object as it is needed in a program

Furthermore, a “scale matrix” allow for entities to increase or decrease in size. Each axis can be scaled independently of the others and thus scaling is not always identical across each axis, however each point is still multiplied by a single scale matrix. As stated in Real Time Rendering the “Larger the scale on an axis, and larger and scaled entity gets in that direction.” (Moller T, 2008, p. 58) This means that objects will become visibly larger or smaller along a set axis. However, instances of an object can be individually scaled to allow some to be larger than others in world space. Multiplying a transform by a negative scale matrix will in fact invert the points position on that axis, leading to a “mirror matrix” (Moller T, 2008, p. 59) which is a useful tool for quickly inverting objects.

Code snippets

Figure 1: Showing implementation of position, rotation and scale matrices and order of combination for axis matrix to rotation matrix.



Other transforms

As mentioned previously Real Time Rendering refers to a specific type of transform known as “Rigid-body transforms” (Moller T, 2008, p. 62). These are transforms which only allow for translation and rotation matrix calculations which maintain the same distance between all an object’s points. These transforms are extremely useful for ensuring an object maintains its shape on rotation or movement. Real Time Rendering defines them as “a transform, consisting of concatenations of only translations the rotations […] and has the characteristics of preserving lengths, angles the handedness.” (Moller T, 2008, p. 62) Following this definition, the transforms used in my graphic program are rigid-body transforms as they maintain their shape and size despite any rotations.

Real Time Rendering also refers to another transform type called “normal transforms”. normal transforms “can be used to consistently transform points, lines, polygons the other geometry.” (Moller T, 2008, p. 63) Which means that no matter an objects rotation, applying a “normal transform” matrix will always translate in the same relative direction. These are useful when defining an objects relative version of a direction, such as up or right because it allows for any movement of points and lines to be consistent with the direction the object would have with its offset.

Shaders

How they work

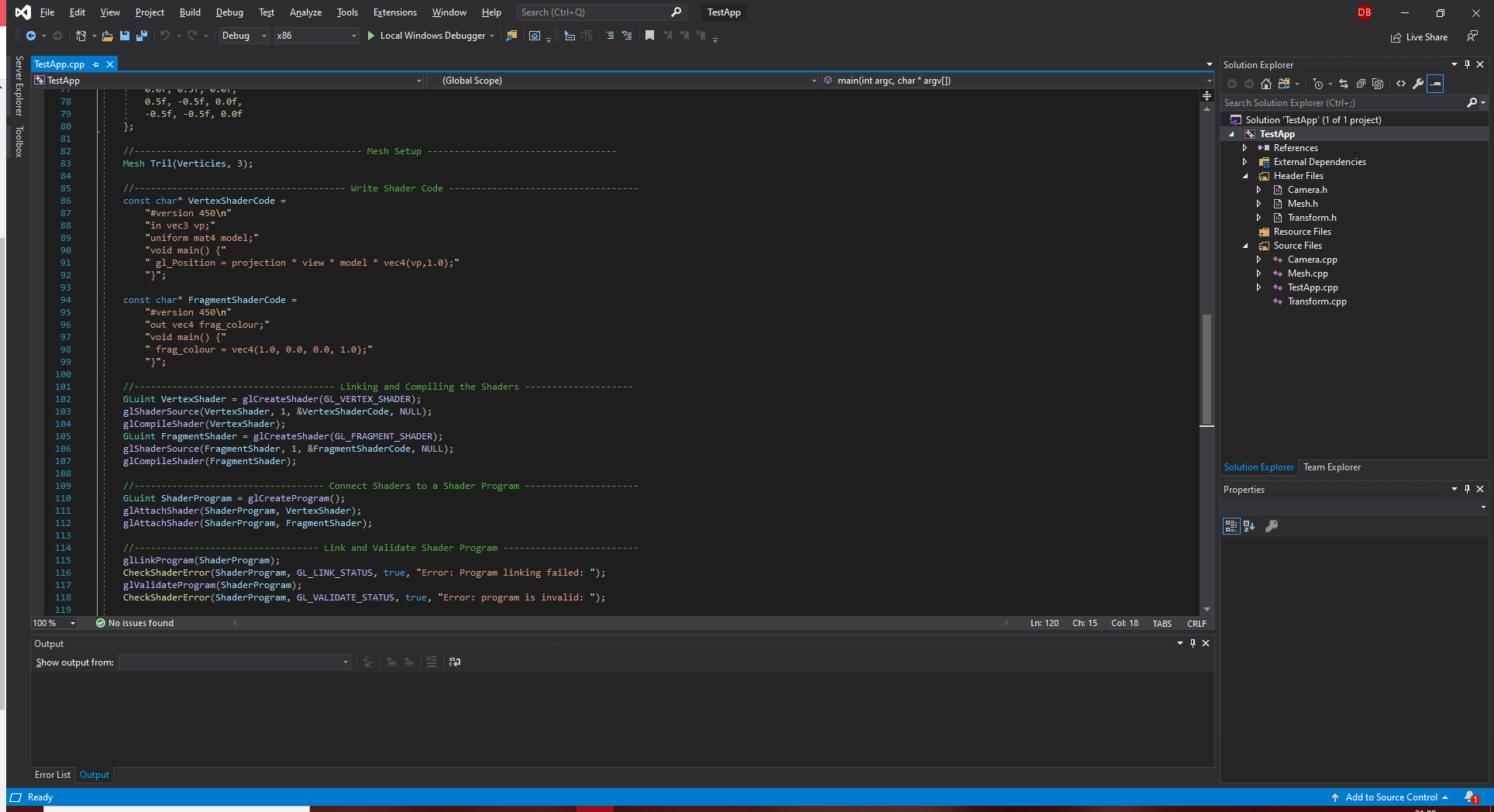
“The vertex shader functionality was first introduced with DirectX 8 in 2001.” (Moller T, 2008, p. 39) Due to the shader being the “first stage in a functional pipeline” and because it was “Invoked infrequently” (Moller T, 2008, p. 39), the shader could be built on the GPU or CPU, “which would then send on the results to and GPU for rasterization.” (Moller T, 2008, p. 39) However as Real Time Rendering states “Some data manipulation happens before this point” (Moller T, 2008, p. 38) but the vertex shader is the first true stage in the graphics pipeline.

“Vertex shaders cannot create or destroy vertices in an object and values cannot be passed between vertices.” (Moller T, 2008, p. 39) As Real Time Rendering states “Since each vertex is treated independently, any number of any number of shader processors on the GPU can be applied in parallel to the incoming stream of vertices.” (Moller T, 2008, p. 39) In addition, the vertex shader “deals exclusively with incoming vertices” (Moller T, 2008, p. 39) which means that it has no access to any information about and triangles and vertices create and only deals with and vertices directly.

A vertex shader is a useful tool for creating the same object with small changes such as colour. These objects are referred to as “instances” and can use “the same array of positions and a different array of colours for its representation.” Instancing an object allows for an object to be drawn any amount of times with some varying data per instance such as colour, however the core of the object will be the same throughout.

Code snippets

Figure 2: Implementation of linking and compiling shaders and connecting shaders to program.



How I can expand on it?

To add extra features and complexity to my code I could integrate a geometry shader. Geometry shaders were added to and graphics pipeline with the release of DirectX 10 in late 2006 and comes “immediately after the vertex shader in the graphics pipeline.” (Moller T, 2008, p. 40/41) It is an entirely optional addition to and pipeline but is required for use of “Shader Model 4.0”.

The geometry shader takes “a single object the its associated vertices and outputs zero or more primitives.” (Moller T, 2008, p. 41) This output is in the form of points, polylines and triangle strips and sometimes no output at all is generated. Despite the input being an object with its vertices, vertices can be selected to be omitted or expanded as required for the shader. In addition, “the output from the shader is guaranteed to be in the same order as they are input” (Moller T, 2008, p. 41) which causes a decrease in performance because “if a number of shader units run in parallel, results must be saved and ordered.” (Moller T, 2008, p. 41)

# Bibliography

Moller T, H. E. H. N., 2008. *Real-Time Rendering.* 3rd ed. New York: A K Peters.