Assignment 2

TDT4195 Visual Computing Fundamentals

Sigurd Totland — MTTK

September 20, 2019

Task 1 Repetition

a)

We create two VBOs. One for vertices and one for the color of said vertices.

b)

To show off our vertex coloring, we create six triangles that are for now non-overlapping. We define our triangles and colors as shown in listing 3. The index set is simply generated and contains values like $1, 2, 3, \ldots$, i.e. our triangle vertices are specified in the order that they are rendered, and are thus specified in such an order that their triangles wind correctly. We pass the color values through the vertex shader, and use them in the fragment shader. The result is shown in figure 1 below.

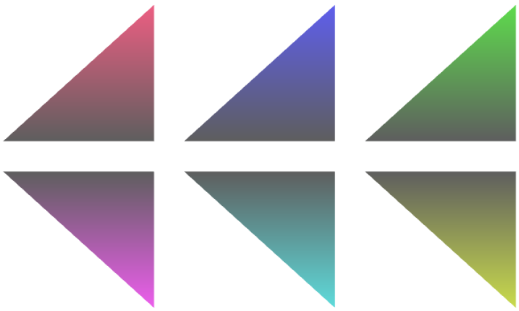

Figure 1: Six gradient-colored triangles.

Task 2 Blending and Depth

a)

We turn on alpha blending and create three transparent, overlapping triangles overlapping at $z = 0.0$, $z = 0.1$ and $z = 0.2$ with different colors. We make sure that they are draw back-to-front. The result when using blending function `GL_ONE_MINUS_SOURCE_ALPHA` is shown in figure 2.

Figure 2: Overlapping triangles.

b)

b.a) i)

Changing the front triangle to green, shown in figure 3 makes the blended colors different.

Figure 3: Overlapping triangles, green in front.

This becomes especially clear if we look at the center triangles separately. These are shown in figure 4.

Figure 4: Overlapping triangle centers.

Clearly, the one with the green triangle in front (left triangle) is much more green than the other. We can analyze the resulting colors. As can be seen in figure 5, the triangles contain equal amounts of

red, but the green and blue colors switch place, exactly as expected.

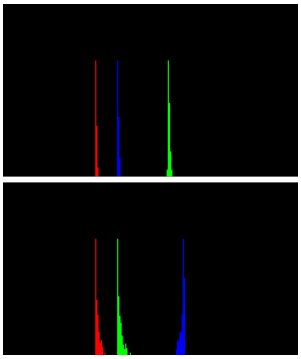

Figure 5: Color intensities. Top: green tri in front. Bottom: blue tri in front.

b.b) ii)

We now try to exchange the order triangle order through the z-coordinate. Placing the front triangle (the blue triangle, which we recall is *drawn last* of the three) inbetween the red and the green, we obtain the (perhaps unexpected) result in figure 6

Figure 6: Depth and draw order mismatched.

This result can be understood by understanding how we emulate transparency in OpenGL and how it relates to the z-buffer. Before painting the fragments of any triangle, transparent or not, we look at at the index buffer for the current fragment. When using `GL_LESS`, if the buffer contains a smaller value than the z-coordinate of the current fragment, it means that some other shape has already been drawn in front and the fragment is hence not painted. In our case, the green triangle is in front of the blue, so upon drawing the blue (which happens after the green) OpenGL does not paint any of the fragments that overlap with the green. For an example of the intended behaviour, we can look at the red and green triangles. The red is really behind the green, but it gets drawn first. So when painting it, every fragment passes the depth test, and are subsequently painted. Afterwards, when the green triangle gets drawn, it (also passing the depth test entirely) gets drawn fully, but because we have enabled alpha blending, the colors drawn get calculated from the blending function. To summarize, the three triangle draws that OpenGL does is shown (from left to right) in figure 7.

Figure 7: Draw steps.

Task 3

b)

We insert premultiply the position with the following matrix in the vertex shader:

Listing 1: GLSL transition matrix

```
mat4 t_mat = mat4(a,   d,   0.0, 0.0,
                  b,   e,   0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  c,   f,   0.0, 1.0);
```

This corresponds to the matrix given in equation two. Note, how it is written down as the transpose of the original matrix, since GLSL has column major matrices. Starting from the previous setup and Manipulating the different variables one at a time yields the setup below.

Figure 8: *a*: Scale in x.

Figure 9: *b*: Skew in x.

Figure 10: *c*: Translation in x.

Figure 11: *d*: Scale in y.

Figure 12: *e*: Skew in y.

Figure 13: *f*: Translation in y.

Lastly, rotation is obtained when the left-upper block matrix of the transformation matrix is in the group $SO(3)$, an example of which is for instance a 90° rotation about the z axis:

$$\mathbf{R}_z = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Task 4

b)

An example of the persective projection and two simple rotations are shown in figure 14 below.

Figure 14: Perspective and rotation

c)

c.a)

We define the following camera struct.

Listing 2: Camera struct

```
typedef struct Camera {
    float x;
    float y;
    float z;
    float phi;
    float theta;
    float psi;
} Camera
```

c.b)

The following table shows the implemented key-binding.

w	forward
a	left
s	backward
d	right
space	up
shift	down
reset	enter

In addition, the camera can be rotated according with the following keybindings

h-l	left-right
k-j	up-down
r-t	roll (ccw-cw)

c.c)

To mimic the movement of a tripod, we only need rotation and translation as well as the perspective transform. The camera can rotate along all 3 axes, along x and y as shown in the assignment, but also the *roll* along the z axis.

c.d)

We multiply the rotation matrices in $x-y-z$ order, then translate it, and finally apply the perspective transform. Obtaining the final transformation matrix

$$T_{\text{final}} = T_{\text{proj}} R_x R_y R_z T_{\text{tran}}. \tag{2}$$

This result is then sent into the vertex shader.

Task 5

a)

For this task, we want to implement a more natural camera motion, akin to mvement in a first-person video game. We here opted for the minecraft-creative-mode type of motion, i.e. when moving in the x-z plane (`wasd`), the camera moves along the x and z axes of the camera (i.e. taking rotation into account), but not in the y direction. This means the camera always stays at the same height until `space` or `shift` is pressed. We do this by simply scaling the movement step with the correct sines and cosines of ϕ for each direction. This implementation is shown in listing 4.

d)

d.a) i)

Possible. This is just a simple scale, shear and translation.

d.b) ii)

Impossible. This changes the shape of the object entirely.

d.c) iii)

Impossible. Parallel lines should be preserved in an affine transformation.

d.d) iv)

Impossible. Ratios of parallel lines are not preserved here. The square lengths are the same as before, whereas the space between them are doubled. An affine transformation cannot do this.

d.e) v)

Impossible. Parrallel lines are again not preserved.

If we can set the z-coordinate arbitrarily, not much changes, as we are still not allowed to do a perspective transform, since it is not affine. iv) is possible if we assume the squares we see are actually 1x1x1 cubes spaced 2 units apart in the z-direction. The transformation shown can then be accomplished with an affine transformation by simply rotating and translating the boxes. This is really cheating though, as we are assuming something about the backside of the object. We could really do the same thing for all the objects.

A C++ snippets

Listing 3: Vertices for 6 triangles

```
std::vector<float> triangleCoords {
   -0.4, 0.05, 0.0,
    -0.4, 0.5, 0.0,
    -0.9, 0.05, 0.0,

    0.8, 0.05, 0.0,
    0.8, 0.5, 0.0,
    0.3, 0.05, 0.0,

    0.2, 0.05, 0.0,
    0.2, 0.5, 0.0,
    -0.3, 0.05, 0.0,

    -0.4, -0.5, 0.0,
    -0.4, -0.05, 0.0,
    -0.9, -0.05, 0.0,

    0.8, -0.5, 0.0,
    0.8, -0.05, 0.0,
    0.3, -0.05, 0.0,

    0.2, -0.5, 0.0,
    0.2, -0.05, 0.0,
    -0.3, -0.05, 0.0,
}


```

```
std::vector<float> triangleColors {
    0.1, 0.1, 0.1, 0.7, // Black
    0.9, 0.1, 0.3, 0.7, // Red
    0.1, 0.1, 0.1, 0.7, // Black

    0.1, 0.1, 0.1, 0.7, // Black
    0.1, 0.8, 0.0, 0.7, // Green
    0.1, 0.1, 0.1, 0.7, // Black

    0.1, 0.1, 0.1, 0.7, // Black
    0.1, 0.1, 0.9, 0.7, // Blue
    0.1, 0.1, 0.1, 0.7, // Black

    0.9, 0.1, 0.9, 0.7, // Magenta
    0.1, 0.1, 0.1, 0.7, // Black
    0.1, 0.1, 0.1, 0.7, // Black

    0.7, 0.8, 0.0, 0.7, // Yellow
    0.1, 0.1, 0.1, 0.7, // Black
    0.1, 0.1, 0.1, 0.7, // Black

    0.1, 0.8, 0.8, 0.7, // Cyan
    0.1, 0.1, 0.1, 0.7, // Black
    0.1, 0.1, 0.1, 0.7, // Black
};
```

Listing 4: Movement keys accounting for camera heading

```
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
{
    cam.x += std::cos(cam.phi) * TRANS_SPEED;
    cam.z += std::sin(cam.phi) * TRANS_SPEED;
}

if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
{
    cam.x -= std::cos(cam.phi) * TRANS_SPEED;
    cam.z -= std::sin(cam.phi) * TRANS_SPEED;
}

if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
{
    cam.x -= std::sin(cam.phi) * TRANS_SPEED;
    cam.z += std::cos(cam.phi) * TRANS_SPEED;
}

if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
{
    cam.x += std::sin(cam.phi) * TRANS_SPEED;
    cam.z -= std::cos(cam.phi) * TRANS_SPEED;
}
```