Individual Assignment 3

Introduction to Computer Graphics

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

last revision February 28, 2014

Abstract

We will discuss the concept of projections from world coordinates onto a surface, for our purposes to produce a projection onto a computer screen. We begin by defining eye coordinates, a projection plane and a look-through window. With these definitions we proceed by looking at how we can produce a parallel projection of objects whose points are given in an arbitrary world coordinate system. Implementation of the parallel and perspective projections are shown and discussed briefly.

The reader is not expected to have any prior knowledge of the material presented. For the code the reader is expected to have some basic understanding of linear algebra and the C++ language.

Contents

1 Introductory

We want to visualize objects whose coordinates are defined by the *World Coordinate System* as though through a camera. That is, we want some amount of control over the view into this world. Camera's are much like objects in and of themselves (i.e. they have position and rotation), but they also have some properties that defines the lense, which affects the image a camera produces. These are the properties we will be discussing, and specifically we will be looking to make use of these properties to transform an arbitrary view volume into the canonical parallel view volume.

1.1 Eye Coordinate System

Let us define what is known as an *Eye Coordinate System*. Firstly, an eye has a position, this is where our *"camera"* will be looking from. Also, we must establish some form of restriction on what the camera can *"see"*, for this we will introduce the notion of a projection plane, with which lines from object coordinates to the viewer eye form points intersecting with the projection plane. This will be our *"window"* into the world.

We define the projection plane in the *World Coordinate System* by a *View Reference Point* (**VRP**, which is a point on the projection plane. The point alone gives us nothing more than a position. Adding a *View Plane Normal* (**VPN**) gives the plane its orientation. Likewise, the camera viewing the world through this projection plane must also establish some orientation, this is achieved by the *View Up Vector* (**VUP**).

The *Eye Coordinate System* $Eye(u, v, n)^T$ is then defined as such

$$n = \frac{VPN}{\|VPN\|} \qquad u = \frac{VUP \times VNP}{\|VUP \times VNP\|} \qquad v = \frac{VPN \times (VUP \times VPN)}{\|VPN \times (VUP \times VPN)\|} \tag{1}$$

All of these vectors are divided by their length in order to ensure that these become unit vectors (i.e. vectors whose magnitude is 1). The n vector points in the direction of the **VPN**. The v vector is perpendicular to vectors **VPN** and **VUP**, making it point along the projection plane. The u vector runs along the projection plane perpendicular to the v vector, which effectively defines the opposing axis.

1.2 The Window

Now that we have the *Eye Space Coordinates* and a projection plane, we can define what will be the window on the projection plane through which our camera will be looking. Since the window will be lying on the projection plane we simply need to delimit an area within this plane. We do so by defining two points in the plane; (u_{min}, v_{min}) denoting the lower-left, and (u_{max}, v_{max}) denoting the upper-right corners of the window. Taking the average of the individual components yields the window center (**CW**[1]). That is $CW = (\frac{u_{min}+u_{max}}{2}, \frac{v_{min}+v_{max}}{2})$.

1.3 Projection

When we define the projection, it is necessary to decide what kind of projection; orthographic, parallel, perspective, etc. For perspective projections a fourth point is needed, namely the

[1]Note that the points **CW** and **VRP** need not coincide

Projection Reference Point (**PRP**), which in relation to the **VRP** defines the *Direction Of Projection* (**DOP**). We can compute this by $DOP = PRP - CW$.

The **DOP** defines the kind of projection we are performing; the projection is called a perspective projection iff. **PRP** is a finite point. Contrarily, the projection is called a parallel projection iff. **PRP** is at infinity. In the case of orthographic projection, obviously points are projected from the world coordinate system along **VPN** onto the projection plane — in order to transform world coordinates into the canonical orthographic view volume this is simply a matter of removing the depth coordinate z.

1.4 View Volumes

A view volume V is a bounded box within which objects can be viewed, if not obscurred by other objects in front. As such $V \subset \mathbb{R}^3$. In example, for a perspective projection the view volume would be a pyramid-shaped boundary box. It is this view volume we wish to transform into the canonical parallel view volume. It should be noted that for view volumes we usually also declare a delimiters for near and far planes, denoting how far and close our camera can see objects.

2 Canonical Parallel View Volume

Having defined the principles on which we have built the notion of a camera thus far, we will now use these to transform world coordinates into what is known as the *Canonical Parallel View Volume*, which produces a parallel projection matrix.

2.1 Translate to the origin

First, we translate the **VRP** to the origin. This is quite simple, we multiply by a translation matrix containing the negated coordinates of **VRP**.

$$T(-VRP) = \begin{bmatrix} 1 & 0 & 0 & -vrp_x \\ 0 & 1 & 0 & -vrp_y \\ 0 & 0 & 1 & -vrp_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

2.2 Rotate to align

Having the **VRP** at the origin, we must now ensure that the eye coordinate axis align with the world. We do this by calculating the unit vectors u, v and n (see section 1.1 on page 2), and then rotate them to line them up with the world axis. This is done by the following rotation matrix.

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix} \tag{3}$$

Where the rotation row vectors are given, as mentioned, by the unit vectors u, v and n. That is,

$$R_z^T = (r_{1z}, r_{2z}, r_{3z}) = \frac{VPN}{\|VPN\|} \tag{4}$$

$$R_x^T = (r_{1x}, r_{2x}, r_{3x}) = \frac{VUP \times R_z}{\|VUP \times R_z\|} \tag{5}$$

$$R_y^T = (r_{1y}, r_{2y}, r_{3y}) = \frac{R_z \times R_x}{\|R_z \times R_x\|} \tag{6}$$

2.3 Shear to parallel projection

Being at the origin and having oriented ourselves correctly, we can now start deforming the world by shearing, such that the direction of projection becomes parallel to the Z-axis.

To do this, we must first do a bit of arithmetic. We need to solve for sh_x and sh_y in the shear matrix.

$$Sh_{par} = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}$$

We know that $DOP = PRP - CW = (dop_u, dop_v, dop_n, 0)^T$, and we want the above shear transformation matrix to produce $DOP' = Sh_{par}DOP = (0, 0, dop_n, 0)^T$.

If we write up the equation, this is what we get

$$\begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} dop_u \\ dop_v \\ dop_n \\ 0 \end{bmatrix} = \begin{bmatrix} dop_u + sh_x dop_n \\ dop_v + sh_y dop_n \\ dop_n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ dop_n \\ 0 \end{bmatrix} \tag{8}$$

the solution to which is

$$sh_x = -\frac{dop_u}{dop_n} \tag{9}$$

$$sh_y = -\frac{dop_v}{dop_n} \tag{10}$$

Substituting this into our shear transformation matrix, we get that

$$Sh_{par} = \begin{bmatrix} 1 & 0 & -\frac{dop_u}{dop_n} & 0 \\ 0 & 1 & -\frac{dop_v}{dop_n} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

with which we can multiply onto our transformations, effectively shearing such that the direction of projection becomes parallel to the Z-axis.

2.4 Final translation and scale

Finally, we translate yet again in order to bring near clipping plane to line up with the XY-plane, and place **CW** at the origin. In order to bring the near clipping plane into alignment with the XY-plane we simply need to translate by $-near$ on the z-axis, and likewise to put **CW** at the origin we simply need to translate by $-cw_u$ on the X-axis and $-cw_v$ on the Y-axis. This yields the following translation matrix.

$$T_{par} = \begin{bmatrix} 1 & 0 & 0 & -cw_u \\ 0 & 1 & 0 & -cw_v \\ 0 & 0 & 1 & -near \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -\frac{u_{min}+u_{max}}{2} \\ 0 & 1 & 0 & -\frac{v_{min}+v_{max}}{2} \\ 0 & 0 & 1 & -near \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

Note that the right-hand side has been expanded according to the definition of **CW** discussed earlier (see section 1.2 on page 2).

Following this translation we scale the current condition of the view volume such that it gets the proportions $[-1,1]_X \times [-1,1]_Y \times [0,-1]_Z$. We do this by multiplying with the following scale transformation matrix.

$$S_{par} = \begin{bmatrix} \frac{2}{u_{max}-u_{min}} & 0 & 0 & 0 \\ 0 & \frac{2}{v_{max}-v_{min}} & 0 & 0 \\ 0 & 0 & \frac{1}{F-B} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

Which effectively brings us to the final matrix and the transformation is completed, the result of which is called *The Canonical Parallel View Volume*.

$$N_{par} = S_{par} \cdot T_{par} \cdot Sh_{par} \cdot R \cdot T(-VRP) \tag{14}$$

The entirety of the transformation is given by the equation above.

3 Implementation

In my implementation of the projection transformation I have a `Camera` class the definition of which is shown below, containing all of the points of vectors defined for the eye coordinate system (see section 1.1 on page 2) on lines 25–27, as well as a projection reference point (see section 1.3 on page 2) on line 28. It also contains the window boundaries (see section 1.2 on page 2) on lines 30–31, as well as the near and far view volume boundaries (see section 1.4 on page 3) on line 33.

```
1  class Camera : public Object
2  {
3  public:
4    Camera(const glm::vec3 &position = glm::vec3(0,0,0));
5    virtual ~Camera();
6
7      static Camera * getActiveCamera();
8      static void setActiveCamera(Camera * camera);
9
10     virtual glm::mat4 getTransform();
11
12     glm::mat4 getParallelTransform();
13     glm::mat4 getPerspectiveTransform();
14
15     glm::vec2 getWindowCenter();
16     glm::vec2 getViewport();
17     glm::vec3 getDirectionOfProjection();
18
19     int test();
20
21 protected:
22 private:
23     static Camera * active;
24
25     glm::vec3 vrp; // view reference point
26     glm::vec3 vpn; // view plane normal
27     glm::vec3 vup; // view up vector
28     glm::vec3 prp; // projection reference point
29
30     glm::vec2 min; // lower-left
31     glm::vec2 max; // upper-right
32
33     float front, back; // delimiter planes
34 };
```

Figure 1: Code excerpt showing the `Camera` class definition.

Many of the class methods are implementation specific, not directly related to performing view volume transformations. These will not be covered. There are, however, a few class methods which needs to be covered before proceeding, as they will be used for convenience later.

3.1 Convenience Methods

First, and foremost the `getWindowCenter` performs the equivalent operation of how **CW** was defined (see section 1.2 on page 2).

```
1 glm::vec2 Camera::getWindowCenter()
2 {
3     return glm::vec2( (min.x + max.x) / 2.0f, (min.y + max.y) / 2.0f );
4 }
```

Figure 2: Code excerpt showing the `Camera::getWindowCenter()` class method.

Another convient class method used is the `getDirectionOfProjection`, which performs the equivalent of how **DOP** was defined (see section 1.3 on page 2).

```
1 glm::vec3 Camera::getDirectionOfProjection()
2 {
3     glm::vec2 cw = getWindowCenter();
4     return glm::vec3(prp.x - cw.x, prp.y - cw.y, prp.z);
5 }
```

Figure 3: Code excerpt showing the `Camera::getDirectionOfProjection()` class method.

Already in this method we see that `getWindowCenter` is being put to good use.

3.2 Parallel Transformation

Below is an excerpt of the implementation of parallel projection. As can be seen, lines 4–9 defines the initial translation (see section 2.1 on page 3). Lines 12–31 builds the rotation matrix (see section 2.2 on page 3). Lines 34–42 builds the shear matrix (see section 2.3 on page 4). Lines 45–59 builds the final translation and scaling (see section 2.4 on page 5). The last line 61 multiplies the matrices, producing the final projection matrix.

```
1  glm::mat4 Camera::getParallelTransform()
2  {
3      // translation
4      glm::mat4 T_vrp({
5          1.0f, 0.0f, 0.0f, 0.0f,
6          0.0f, 1.0f, 0.0f, 0.0f,
7          0.0f, 0.0f, 1.0f, 0.0f,
8          -vrp.x, -vrp.y, -vrp.z, 1.0f
9      });
10
11     // z-rotation
12     float l = glm::length(vpn);
13     glm::vec3 rz = vpn / l;
14
15     // x-rotation
16     glm::vec3 vup_x_rz = glm::cross(vup, rz);
17     l = glm::length(vup_x_rz);
18     glm::vec3 rx = vup_x_rz / l;
```

```
20      // y-rotation
21      glm::vec3 rz_x_rx = glm::cross(rz, rx);
22      l = glm::length(rz_x_rx);
23      glm::vec3 ry = rz_x_rx / l;
24
25      // rotation
26      glm::mat4 R({
27          rx.x, ry.x, rz.x, 0.0f,
28          rx.y, ry.y, rz.y, 0.0f,
29          rx.z, ry.z, rz.z, 0.0f,
30          0.0f, 0.0f, 0.0f, 1.0f
31      });
32
33      // shear - Sh
34      glm::vec3 dop = getDirectionOfProjection();
35      float sh_x = -(dop.x / dop.z);
36      float sh_y = -(dop.y / dop.z);
37      glm::mat4 Sh({
38          1.0f, 0.0f, 0.0f, 0.0f,
39          0.0f, 1.0f, 0.0f, 0.0f,
40          sh_x, sh_y, 1.0f, 0.0f,
41          0.0f, 0.0f, 0.0f, 1.0f
42      });
43
44      // translate
45      glm::vec2 cw = getWindowCenter();
46      glm::mat4 T_par({
47          1.0f, 0.0f, 0.0f, 0.0f,
48          0.0f, 1.0f, 0.0f, 0.0f,
49          0.0f, 0.0f, 1.0f, 0.0f,
50          -cw.x, -cw.y, -front, 1.0f
51      });
52
53      // scale
54      glm::mat4 S_par({
55          2.0f / (max.x - min.x), 0.0f, 0.0f, 0.0f,
56          0.0f, 2.0f / (max.y - min.y), 0.0f, 0.0f,
57          0.0f, 0.0f, 1.0f / (front - back), 0.0f,
58          0.0f, 0.0f, 0.0f, 1.0f
59      });
60
61      return S_par * T_par * Sh * R * T_vrp * Object::getTransform();
62  }
```

Figure 4: Code excerpt showing the `Camera::getParallelTransform()` class method.

3.3 Perspective Transformation

Many things translates directly from the parallel projection over into the perspective projection matrix. On lines 4–9 we perform the exact same translation, and also lines 12–31 produces the axis alignment rotation. From here on out, things begin to differ. These differences are, as far as I understand, not supposed to be covered by the assignment scope, and an explanation of these would require more theory than provided in earlier sections. So I will leave the code excerpt as is, with no further explanation.

```
glm::mat4 Camera::getPerspectiveTransform()
{
    // translation - T(-VRP)
    glm::mat4 T_vrp({
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        -vrp.x, -vrp.y, -vrp.z, 1.0f
    });

    // z-rotation
    float l = glm::length(vpn);
    glm::vec3 rz = vpn / l;

    // x-rotation
    glm::vec3 vup_x_rz = glm::cross(vup, rz);
    l = glm::length(vup_x_rz);
    glm::vec3 rx = vup_x_rz / l;

    // y-rotation
    glm::vec3 rz_x_rx = glm::cross(rz, rx);
    l = glm::length(rz_x_rx);
    glm::vec3 ry = rz_x_rx / l;

    // rotation - R
    glm::mat4 R({
        rx.x, ry.x, rz.x, 0.0f,
        rx.y, ry.y, rz.y, 0.0f,
        rx.z, ry.z, rz.z, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f
    });

    // translation - T(-PRP)
    glm::mat4 T_prp({
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f, 0.0f,
        -prp.x, -prp.y, -prp.z, 1.0f
    });

    // shear - Sh
    glm::vec3 dop = getDirectionOfProjection();
    float sh_x = -(dop.x / dop.z);
    float sh_y = -(dop.y / dop.z);
    glm::mat4 Sh({
        1.0f, 0.0f, 0.0f, 0.0f,
        0.0f, 1.0f, 0.0f, 0.0f,
        sh_x, sh_y, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 1.0f
    });
```

```
52      glm::mat4 S({
53          (-2.0f * prp.z) / ( (max.x - min.x) * (back - prp.z)), 0.0f, 0.0f, 0.0
                f,
54          0.0f, (-2.0f * prp.z) / ( (max.y - min.y) * (back - prp.z)), 0.0f, 0.0
                f,
55          0.0f, 0.0f, -1.0f / (back - prp.z), 0.0f,
56          0.0f, 0.0f, 0.0f, 1.0f,
57      });
58
59      glm::mat4 P({
60          1.0f, 0.0f, 0.0f, 0.0f,
61          0.0f, 1.0f, 0.0f, 0.0f,
62          0.0f, 0.0f, 1.0f, 1.0f / (prp.z / (back - prp.z) ),
63          0.0f, 0.0f, 0.0f, 0.0f,
64      });
65
66      return P * S * Sh * T_prp * R * T_vrp * Object::getTransform();
67  }
```

Figure 5: Code excerpt showing the `Camera::getPerspectiveTransform()` class method.

I should note that the first camera property test shows nothing on screen in the current program version — this might be a bug in the implementation, I failed to ascertain the error, but I am aware that it is there.

3.4 Tests

I have arranged the program code provided with this assignment to automatically run through each camera property tests asked to test by the assignment text. A few screenshots of which are shown below.

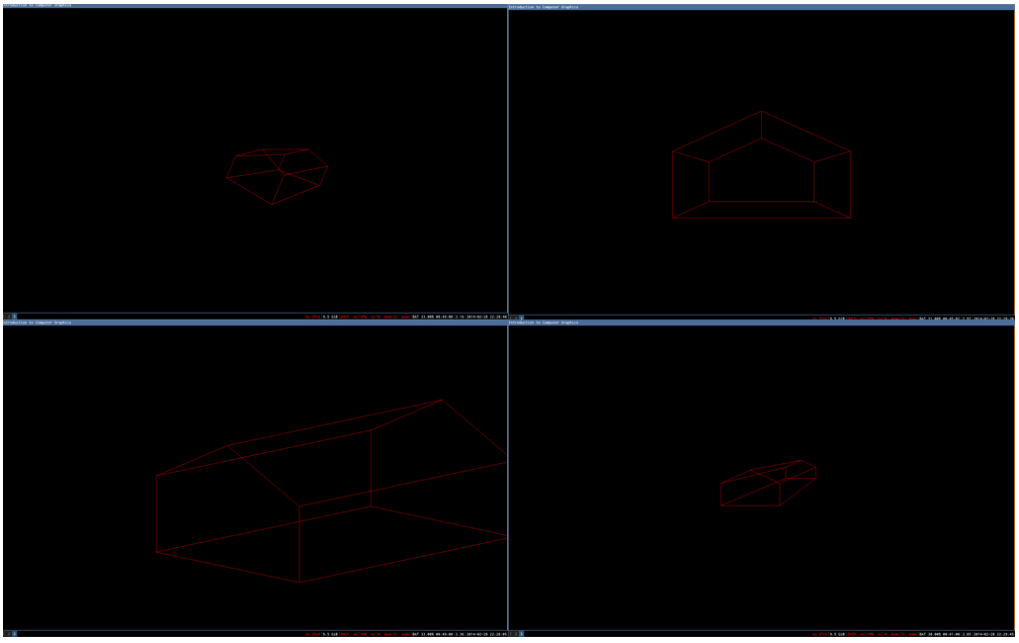

Figure 6: Screenshots from the test program