Lecture 3

View/eye/camera coordinates

In order to define an image of a scene, we need to specify the 3D position of the viewer and we also need to specify the orientation of the viewer. The way this is traditionally done is to specify a 3D "eye" point (viewer/camera position), and to specify a point in the scene where the eye is looking at – called the "look at" point. This specifies the viewer's z axis. We also need to orient the viewer to specify where the viewer's x and y axes point. This involves spinning the viewer around the viewer's z axis. I'll explain how this is done next. I will use the term "viewer", "eye", "camera" interchangably.

Let $\mathbf{p}_c$ be the position of the viewer in world coordinates. Let $\mathbf{p}_{lookat}$ be a position that the viewer is looking at. This defines a vector $\mathbf{p}_c - \mathbf{p}_{lookat}$ from the look at point to the viewer. Note that this vector points *towards the viewer*, rather than toward the look-at point. The camera's z axis is defined to be the normalized vector

$$\hat{\mathbf{z}}_c = \frac{\mathbf{p}_c - \mathbf{p}_{lookat}}{\parallel \mathbf{p}_c - \mathbf{p}_{lookat} \parallel}.$$

[ASIDE: in the lecture itself, I used the traditional notation and defined $\mathbf{N} = \mathbf{p}_c - \mathbf{p}_{lookat}$ and $\mathbf{n}$ to be the normalized version. But afterwards I changed my mind and decided this extra notation was not necessary. I mention it now only in case you are looking at the video of the lecture and wondering what the heck happened to the $\mathbf{N}$ and $\mathbf{n}$.]

We have specified the position of the camera and the direction where the camera is looking. We next need to orient the camera and specify which direction is "up", that is, where the y axis of the camera is pointing in world coordinates. The x axis is then specified at the same time since it must perpendicular to the camera's y and z axes.

We use the following notation. We specify the camera's x, y and z axes with the vectors $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$. These vectors are typically different from the world coordinate x, y, z axes which are written $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ without the c subscript.

To construct the camera coordinate axes, recall Examples 1 and 2 from last lecture, when we specified one row of a rotation matrix and we needed to find two other rows. Here we face a similar task in defining the camera coordinates axes. Essentially we need to define a 3×3 rotation matrix whose rows are the vectors $\hat{\mathbf{x}}_c, \hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$.

We will specify the camera's x and y axes as follows. We choose a vector – called the *view up* $\mathbf{V}_{up}$ vector – that is *not* parallel to $\hat{\mathbf{z}}_c$. It is common to define $\mathbf{V}_{up}$ to be the world $\hat{\mathbf{y}}$ vector, i.e. $(0, 1, 0)$. This works provided that the viewer is not looking in the $\mathbf{y}$ direction. Notice that the $\mathbf{V}_{up}$ vector does not need to be perpendicular to $\mathbf{z}_c$. Rather the constraint is much weaker: it only needs to be not parallel to $\mathbf{z}_c$.

In practice, the user specifies $\mathbf{V}_{up}$, $\mathbf{p}_c$, and $\mathbf{p}_{lookat}$. Then $\hat{\mathbf{x}}_c$ and $\hat{\mathbf{y}}_c$ are computed:

$$\hat{\mathbf{x}}_c \equiv \frac{\mathbf{V_{up}} \times (\mathbf{p_c} - \mathbf{p_{lookat}})}{\parallel \mathbf{V_{up}} \times (\mathbf{p_c} - \mathbf{p_{lookat}}) \parallel}, \qquad \hat{\mathbf{y}}_c \equiv \hat{\mathbf{z}}_\mathbf{c} \times \hat{\mathbf{x}}_\mathbf{c} \; .$$

These definitions are a bit subtle. Although we define $\mathbf{V}_{up}$ with $\mathbf{y}_c$ in mind, in fact $\mathbf{V}_{up}$ is used directly to define the $\hat{\mathbf{x}}_c$ vector, which then in turn defines $\hat{\mathbf{y}}_c$. Bottom line: we have a simple method for computing orthonormal camera coordinate vectors $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$.

Example

Suppose that we place a viewer at world coordinate position $(2, 1, 1)$ and we orient the viewer to be looking at the origin. Then

$$\mathbf{p}_c - \mathbf{p}_{lookat} = (2, 1, 1) - (0, 0, 0) = (2, 1, 1).$$

Then $\mathbf{z}_c$ is just the normalized vector $\frac{1}{\sqrt{6}}(2, 1, 1)$. Suppose we let $\mathbf{V}_{up} = (0, 1, 0)$. What are $\mathbf{x}_c$ and $\mathbf{y}_c$? It is easy to calculate that

$$\mathbf{V}_{up} \times (\mathbf{p}_c - \mathbf{p}_{lookat}) = (1, 0, -2)$$

and so

$$\mathbf{x}_c = \frac{1}{\sqrt{5}}(1, 0, -2)$$

and

$$\mathbf{y}_c = \hat{\mathbf{z}}_c \times \hat{\mathbf{x}}_c = \frac{1}{\sqrt{30}}(-2, 5, -1).$$

Eye coordinates in OpenGL

OpenGL makes it very easy to specify the camera/eye coordinates. The user needs to specify three vectors: where the eye is located, the 3D point that the eye is looking at, and the "up" vector.

```
eye    =  ...    #  specify 3D coordinates of eye
lookat =  ...    #  discussed below
up     =  ...    #     "

gluLookAt(eye[0], eye[1], eye[2], lookat[0], lookat[1], lookat[2], up[0],up[1],up[2])
```

Transforming from world to viewer/camera/eye coordinates

We would like to make images from the viewpoint of an camera (eye, viewer) and so we need to express the positions of points in the world in camera coordinates. We transform from world coordinates to viewer coordinates using a 3D translation matrix $\mathbf{T}$ and a rotation matrix $\mathbf{R}$. Suppose we have a point that is represented in world coordinates by $(x, y, z)_{world}$. We wish to represent that point in camera coordinates $(x, y, z)_c$. To do so, we multiply by the matrices $\mathbf{R}\ \mathbf{T}$. What should these matrices be and in which order should we apply them?

We first use the matrix $\mathbf{T}$ to translate the eye position (expressed in world coordinates) to the origin. The rotation matrix $\mathbf{R}$ then should rotate $\Re^3$ such that the eye's axes (expressed in world coordinates) are mapped to the canonical vectors $(1, 0, 0), (0, 1, 0), (0, 0, 1)$. Together we have

$$\mathbf{M}_{viewer \leftarrow world} \ = \ \mathbf{R}\ \mathbf{T}.$$

The translation matrix $\mathbf{T}$ that takes $\mathbf{p}_c$ to the origin must be:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -p_{c,x} \\ 0 & 1 & 0 & -p_{c,y} \\ 0 & 0 & 1 & -p_{c,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note that we are transforming from the world coordinate to eye coordinates and so the position of the eye (in world coordinates) should be transformed to the origin (in eye coordinates).

These two vectors are the first two rows of $\mathbf{R}$. The rotation matrix that rotates the world coordinates to viewer coordinates is $\mathbf{R}$ below. To simplify the notation, let $\mathbf{u}$, $\mathbf{v}$, $\mathbf{n}$ be $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$, respectively, where these vectors are expressed in world coordinates.

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[ASIDE: After class, one student asked about the "axis" of this rotation matrix. Such an axis certainly exists (see last lecture), but it is not usually how we think of the camera orientation. Rather, we usually think of the camera orientation in terms of the direction in which the camera is looking and the up vector.]

Object coordinates

Now that the camera has been specified, let's return to the issue of model transformations which we introduced last lecture. We will now start thinking about the objects that we want to draw! We begin by introducing some basic object models in OpenGL. Then we will look at how to transform these models so as to place them in 3D space i.e. world coordinates. We will finish the lecture by combining all our transformations together: from object coordinates to world coordinates to camera coordinates.

In classic OpenGL, one defines a point with a `glVertex()` command. There are several forms depending on whether the point is 2D or 3D or 4D, whether the coordinates are integers or floats, etc. whether the coordinates are a vector ("tuple") or are separate arguments.
`https://www.opengl.org/sdk/docs/man2/xhtml/glVertex.xml`
For example, if x, y, and z are floats, we can define a point:

```
glVertex3f(x,  y , z)
```

To define a line segment, we need two vertices which are grouped together:

```
glBegin(GL_LINES)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glEnd()
```

To define n line segments, we can list 2 n vertices. For example, to define two line segments:

```
glBegin(GL_LINES)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glVertex3f(x3,  y3, z3)
glVertex3f(x4,  y4 , z4)
glEnd()
```

To define a sequence of lines where the endpoint of one line is the begin point of the next, i.e. a polygon, we write:

```
glBegin(GL_POLYGON)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glVertex3f(x3,  y3 , z3)
glVertex3f(x4,  y4 , z4)
glEnd()
```

This automatically connects the last point to the first.

Typically we define surfaces out of triangles or quadrangles ("quads"). Quads are four sided polygons and ultimately are broken into triangles. One can declare triangles or quads similar to the above, but use GL_TRIANGLES or GL_QUADS instead of GL_POLYGON.

Quadrics and other surfaces

One simple class of surfaces to draw is the sphere and cylinder and other quadric (quadratic) surfaces such as cones and hyperboloids. A general formula for a quadric surface in $\Re^3$ is

$$ax^2 + by^2 + cz^2 + dxy + eyz + fxz + gx + hy + iz + j = 0.$$

Simple examples are:

- an ellipsoid centered at (x_0, y_0, z_0), namely

  $$a(x - x_0)^2 + b(y - y_0)^2 + c(z - z_0)^2 - 1 = 0$$

  where $a, b, c > 0$. Note that this includes the special case of a sphere.

- a cone with apex at (x_0, y_0, z_0), and axis parallel to x axis

  $$a(x - x_0)^2 = (y - y_0)^2 + (z - z_0)^2$$

- a paraboloid with axis parallel to x axis

  $$ax = (y - y_0)^2 + (z - z_0)^2$$

- etc

The above general formula for a quadric can be rewritten as:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & \frac{d}{2} & \frac{f}{2} & \frac{g}{2} \\ \frac{d}{2} & b & \frac{e}{2} & \frac{h}{2} \\ \frac{f}{2} & \frac{e}{2} & c & \frac{i}{2} \\ \frac{g}{2} & \frac{h}{2} & \frac{i}{2} & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

or

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \mathbf{Q} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

where $\mathbf{Q}$ is a symmetric 4×4 matrix. For example, if a, b, c are all positive, then

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

is an ellipsoid centered at the origin.

[——————**The rest of the page was modified Feb. 16**————————-]

Suppose that we *arbitrarily* insert a rotation and translation matrix as follows:

$$(\mathbf{RTx})^T \mathbf{Q} (\mathbf{RTx}) = 0 \qquad (*)$$

Intuitively, this must give us a new ellipsoid since all we are doing is translating and rotating. So what is the new quadratic matrix that describes this new ellipsoid?

To answer this question, consider a new variable **x'**,

$$\mathbf{x}' = \mathbf{RTx}. \qquad (**)$$

Since

$$(\mathbf{x}')^{\mathbf{T}} \mathbf{Q} \mathbf{x}' = \mathbf{0}$$

the points **x'** that satisfy this equation must be the points on the original ellipsoid. We now rewrite (*) to get

$$\mathbf{x}^T (\mathbf{T}^T \mathbf{R}^T \mathbf{QRT}) \mathbf{x} = 0$$

that is,

$$\mathbf{x}^T \mathbf{Q}' \mathbf{x} = 0$$

for some new quadratic matrix **Q'**. This is the quadratic matrix for the new ellipsoid that we obtain when we map **x** using (**).

How is the new ellipsoid related to the original one? Well, from (**),

$$\mathbf{T}^{-1} \mathbf{R}^{\mathbf{T}} \mathbf{x}' = \mathbf{x}.$$

Thus, if we take the points **x'** on the original ellipsoid, rotate them by $\mathbf{R}^T$ and apply the inverse translation $\mathbf{T}^{-1}$, we get the points **x** on the new ellipsoid. So, the new ellipsoid is obtained from the original by a rotation and then a translation.

OpenGL does not include general quadrics as geometric primitives but there is an OpenGL *Utility Library* (GLU library) that includes some quadrics, in particular, spheres and cylinders.

```
  myQuadric = gluNewQuadric()
  gluSphere(myQuadric,   ...)
  gluCylinder(myQuadric, ...)
```

where `gluSphere()` and `gluCylinder()` have parameters that need to be defined such as the number of vertices used to approximate the shape. The cylinder is really a more general shape which includes a cone. `https://www.opengl.org/sdk/docs/man2/xhtml/gluCylinder.xml`

There is another library called the OpenGL Utility Toolkit (GLUT) which allows us to define other shapes:

```
glutSolidCone()
glutSolidCube()
glutSolidTorus()
glutSolidTeapot()     //  yes, a teapot
```

How to transform an object in OpenGL?

When you define an object, whether it is a vertex or line or a teapot, the object is defined in "object coordinates". To place the object somewhere in the world and to orient it in the world, you need to apply a transformation – typically a rotation and translation, but also possibly a scaling. To transform object, OpenGL provides translation, rotation and scaling transformations:

```
glTranslatef(x,y,z)
glRotatef(vx, vy, vz, angle )
glScalef(sx, sy, sz)
```

Notice from the syntax that these do not operate on particular objects (or geometric primitives). Rather, they are applied to all (subsequently defined) geometry primitives.

Before we see how this works, let me sketch the general pipeline. You define an object – say its a dog – by specifying a bunch of spheres (head and torso) and cylinders (legs, tail) or maybe you are more sophisticated and build the dog out of thousands of triangles. These primitives are defined with respect to an object coordinate system. You place each sphere, triangle, etc at some position in this coordinate system and you orient the primitives. This object modelling step is done with a bunch of transform commands (translate, rotate, scale) and primitive declarations. Let's refer to one these transform commands specifically as $\mathbf{M}_{object \leftarrow vertex}$. It places a vertex (or primitive e.g. the dog's ear) somewhere in the dog coordinate system.

The dog object then needs to be placed in the scene and oriented in the scene, and so another set of transformations must be applied, namely a translation and rotation that map all the vertices of the dog from dog coordinates to world coordinates. Finally, as we discussed in the first part of this lecture, we will need to transform from world coordinates to viewer coordinates.

$$\mathbf{M}_{viewer \leftarrow world}\ \mathbf{M}_{world \leftarrow object}\ \mathbf{M}_{object \leftarrow vertex}$$

OpenGL GL_MODELVIEW matrix

We have covered all the transformations we need. We just now need to assemble them together and order them properly. Specifically, how is this done in OpenGL?

OpenGL is a "state machine" which means that it has a bunch of global variables (which you can access through getters and setters). One of these states is a 4×4 transformation matrix called GL_MODELVIEW. Whenever your program declares a vertex (glVertex) or any other geometric object, this vertex is immediately sent through the pipeline of transformations mentioned above, from object to world to camera coordinates.

The GL_MODELVIEW matrix should be initialized to be the identity matrix.

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
```

At this initialization stage, the view coordinates, camera coordinates, and object coordinates are all the same. The viewer is then defined with a gluLookat command (see p. 2).

```
gluLookat( ... )
```

This multiplies the GL_MODELVIEW matrix by the appropriate matrix that transforms from world to viewer coordinates. In the first part of this lecture I said how this matrix was constructed out of a rotation and translation.

At this point, the object coordinate system and world coordinate system are identical. That is, if an object is defined e.g. by glVertex commands, then the object is placed at the origin in world coordinates. OpenGL automatically multiplies the vertices of the object by the GL_MODELVIEW matrix, and thereby transforms from object coordinates (which are the same as world coordinates in this case) to viewer coordinates.

Generally we want to position objects somewhere other than the origin in world coordinates and to give them an arbitrary orientation as well. To do so, we need to transform from object to world coordinates. Prior to defining the vertices of the object, we call glTranslate and glRotate which together define $\mathbf{M}_{world \leftarrow object}$. So, recalling that gluLookAt defines $\mathbf{M}_{viewer \leftarrow world}$ and multiplies it with the GL_MODELVIEW matrix (which should have been initialized to the identity), the following commands will compute the position of the vertex in camera coordinates as follows:

$$\mathbf{M}_{viewer \leftarrow world} \mathbf{M}_{world \leftarrow object} \mathbf{x}_{obj}$$

Here is the OpenGL sequence:

```
gluLookAt( ... )         // transform from world to viewer  coordinates

glRotate( ... )          // transform from dog to world coordinates
glTranslate( ... )       //  i.e.  translate and rotate the dog

glVertex(  )             //  a vertex of the dog object
```

What's happening here is that each of the first three of the above commands multiplies the GL_MODELVIEW matrix on the right by some new transformation matrix, giving

$$\mathbf{M}_{\texttt{GL\_MODELVIEW}} = \mathbf{M}_{viewer \leftarrow world} \, \mathbf{R} \, \mathbf{T}.$$

Conceptually, the GL_MODELVIEW matrix is a composition of two maps. There is the map from object coordinates to world coordinates, and there is a map from world coordinates to viewer coordinates.

$$\mathbf{M}_{\texttt{GL\_MODELVIEW}} = \mathbf{M}_{viewer\leftarrow world}\ \mathbf{M}_{world\leftarrow object}$$

But there is no guarentee that this is what you will get. For example, if you mistakenly write out two consecutive gluLookAt commands, then $\mathbf{M}_{\texttt{GL\_MODELVIEW}}$ will be updated as follows:

$$\mathbf{M}_{\texttt{GL\_MODELVIEW}} \leftarrow \mathbf{M}_{\texttt{GL\_MODELVIEW}}\ \mathbf{M}_{viewer\leftarrow world}\ \mathbf{M}_{viewer\leftarrow world}$$

which is non-sense.

Finally, lets think about how to define multiple objects, for example, a dog and a house. You might think you could write:

```
gluLookAt( ... )         // transform from world to viewer  coordinates

glTranslate( ... )          // transform from dog to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the dog
defineDog(  )            //  all vertices of the dog object

glTranslate( ... )          // transform from house to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the house
defineHouse(  )          //  all vertices of the house object
```

However, this doesn't work since it would transform the house vertices from house coordinates to dog coordinates, when instead we want to transform the house vertices into world coordinates (unless of course we *wanted* to define the house with respect to the dog position, which is unlikely...)

To get around this problem, OpenGL has a GL_MODELVIEW stack. The stack allows you to remember (push) the state of the GL_MODELVIEW matrix and replace (pop) the GL_MODELVIEW matrix by the most recent remembered state. Here is an how the last example would work:

```
gluLookAt( ... )         // transform from world to viewer  coordinates

glPushMatrix()
glTranslate( ... )          // transform from dog to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the dog
defineDog(  )            //  all vertices of the dog object
glPopMatrix()

glPushMatrix()
glTranslate( ... )          // transform from house to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the house
defineHouse(  )          //  all vertices of the house object
glPopMatrix()
```

One final point: you'll note that in the above we first write translate and then rotate ($\mathbf{T}\ \mathbf{R}$), whereas when we transformed from world to viewer coordinates (bottom p. 2), we applied them in the opposite order ($\mathbf{R}\ \mathbf{T}$). As an exercise, ask yourself why.