CMSC 425: Lecture 6
3-d Viewing and Projections
Tuesday, Feb 12, 2013

Reading: See any standard reference on OpenGL or GLUT.

Viewing in OpenGL: Today we will discuss how viewing and perspective transformations are handled for 3-dimensional scenes. In OpenGL, and most similar graphics systems, the process involves the following basic steps, of which the perspective transformation is just one component. We assume that all objects are initially represented relative to a standard 3-dimensional coordinate frame, in what are called *world coordinates*.

Modelview transformation: Maps objects (actually vertices) from their world-coordinate representation to one that is centered around the viewer. The resulting coordinates are variously called *camera coordinates*, *view coordinates*, or *eye coordinates*. (Specified by the OpenGL command gluLookAt.)

Projection: This projects points in 3-dimensional eye-coordinates to points on a plane called the *image plane*. This projection process consists of three separate parts: the projection transformation (affine part), clipping, and perspective normalization. Each will be discussed below. The output coordinates are called *normalized device coordinates*. (Specified by the OpenGL commands such as gluOrtho2D, glOrtho, glFrustum, and gluPerspective.)

Mapping to the viewport: Convert the point from these idealized normalized device coordinates to the viewport. The coordinates are called *window coordinates* or *viewport coordinates*. (Specified by the OpenGL command glViewport.)

We have ignored a number of issues, such as lighting and hidden surface removal. These will be considered separately later. The process is illustrated in Fig. 1. We have already discussed the viewport transformation, so it suffices to discuss the first two transformations.

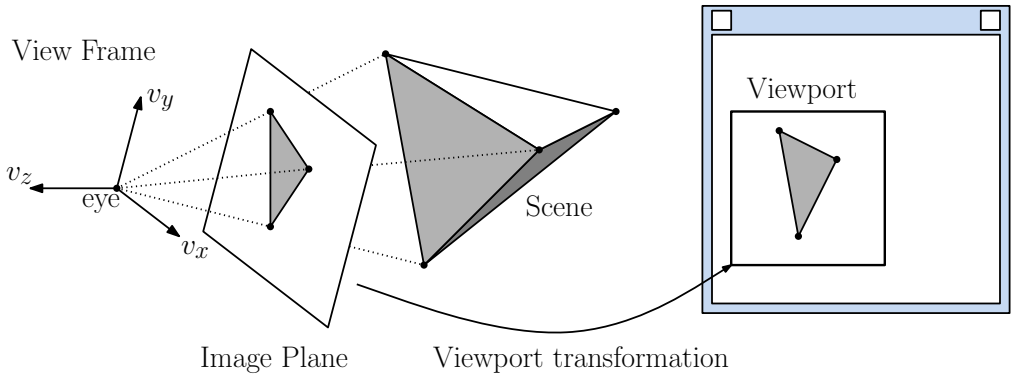

Fig. 1: OpenGL Viewing Process.

Converting to Camera-Centered Coordinate System: As we shall see below, the perspective transformation is simplest when the *center of projection*, the location of the viewer, is the origin and the *image*

plane (sometimes called the *projection plane* or *view plane*), onto which the image is projected, is orthogonal to one of the axes, say the z-axis. Let us call these *camera coordinates*. However the user represents points relative to a coordinate system that is convenient for his/her purposes. Let us call these *world coordinates*. This suggests that, prior to performing the perspective transformation, we perform a change of coordinate transformation to map points from world coordinates to camera coordinates.

In OpenGL, there is a nice utility for doing this. The procedure gluLookAt generates the desired transformation to perform this change of coordinates and multiplies it times the transformation at the top of the current transformation stack. (Recall OpenGL's transformation structure from the previous lecture on OpenGL transformations.) This should be done in Modelview mode.

Conceptually, this change of coordinates is performed *last*, after all other Modelview transformations are performed, and immediately before the projection. By the "reverse rule" of OpenGL transformations, this implies that this change of coordinates transformation should be the *first* transformation on the Modelview transformation matrix stack. Thus, it is almost always preceded by loading the identity matrix. Here is the typical calling sequence. This should be called when the camera position is set initially, and whenever the camera is (conceptually) repositioned in space.

Typical Structure of Redisplay Callback

```
void myDisplay() {
                                          // clear the buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();                       // start fresh
                                          // set up camera frame
  gluLookAt(eyeX, eyeY, eyeZ,  atX, atY, atZ,  upX, upY, upZ);
  myWorld.draw();                         // draw your scene
  glutSwapBuffers();                      // make it all appear
}
```

The arguments are all of type GLdouble. The arguments consist of the coordinates of two points and vector, in the standard coordinate system. The point *eye* $= (e_x, e_y, e_z)^\mathsf{T}$ is the *viewpoint*, that is the location of they viewer (or the camera). To indicate the direction that the camera is pointed, a central point at which the camera is directed is given by *at* $= (a_x, a_y, a_z)^\mathsf{T}$. The "at" point is significant only in that it defines the *viewing vector*, which indicates the direction that the viewer is facing. It is defined to be *at* $-$ *eye* (see Fig. 2).

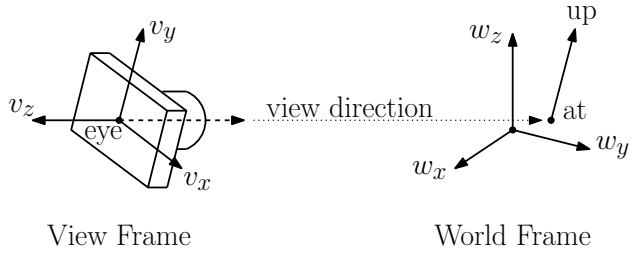

Fig. 2: The world frame, parameters to gluLookAt, and the camera frame.

These points define the position and direction of the camera, but the camera is still free to rotate about

the viewing direction vector. To fix last degree of freedom, the vector $\overrightarrow{up} = (u_x, u_y, u_z)^\mathsf{T}$ provides the direction that is "up" relative to the camera. Under typical circumstances, this would just be a vector pointing straight up (which might be $(0, 0, 1)^\mathsf{T}$ in your world coordinate system). In some cases (e.g. in a flight simulator, when the plane banks to one side) you might want to have this vector pointing in some other direction (e.g., up relative to the pilot's orientation). This vector *need not* be perpendicular to the viewing vector. However, it cannot be parallel to the viewing direction vector.

The Camera Frame: OpenGL uses the arguments to gluLookAt to construct a coordinate frame centered at the viewer. The x- and y-axes are directed to the right and up, respectively, relative to the viewer. It might seem natural that the z-axes be directed in the direction that the viewer is facing, but this is not a good idea.

To see why, we need to discuss the distinction between right-handed and left-handed coordinate systems. Consider an orthonormal coordinate system with basis vectors $\vec{v}_x$, $\vec{v}_y$ and $\vec{v}_z$. This system is said to be *right-handed* if $\vec{v}_x \times \vec{v}_y = \vec{v}_z$ (see Fig. 3), and left-handed otherwise ($\vec{v}_x \times \vec{v}_y = -\vec{v}_z$). Right-handed coordinate systems are used by default throughout mathematics. (Otherwise computation of orientations is all screwed up.) Given that the x- and y-axes are directed right and up relative to the viewer, if the z-axis were to point in the direction that the viewer is facing, this would result in left-handed coordinate system. The designers of OpenGL wisely decided to stick to a right-handed coordinate system, which requires that the z-axes is directed opposite to the viewing direction.

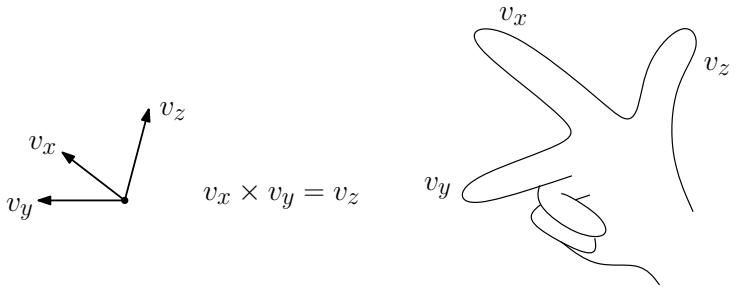

Fig. 3: Right-handed coordinate system.

Building the Camera Frame: (Optional) How does OpenGL implement this change of coordinate transformation? In order to answer this, you will need to recall a bit of linear algebra (assuming you studied linear algebra). We need to know two things. First, given two points p and q, the difference $p - q$ yields a vector that points from q to p. Given a vector $\vec{v}$, we define normalize to be a function that returns a vector pointing in the same direction as $\vec{v}$, but is of unit length. The Euclidean length of a vector $\vec{v} = (v_x, v_y, v_z)$ is defined to be $\|v\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$. Therefore, we can define normalize$(v) = v/\|v\|$. Finally, recall the *cross product* operation. Given two vectors $\vec{u}$ and $\vec{v}$, their cross product, $\vec{u} \times \vec{v}$ is a vector that is orthogonal to both of these vectors. (It length is equal to the area of the parallelogram spanner between these two vectors, but we won't need this.) The direction of the cross product is given by *right-hand rule*. A *coordinate frame* in three dimensional space consists of an *origin point* and three *axis vectors*. The most useful frame is *orthonormal*, which means that each of the three axis vectors is of unit length and they are pairwise orthogonal to each other.

Given this background, computing the camera frame turns out to be an exercise in linear algebra. We want to construct an orthonormal frame whose origin is the point *eye*, whose $-z$-basis vector

is parallel to the view vector, and such that the $\overrightarrow{up}$ vector projects to the up direction in the final projection. (This is illustrated in the Fig. 4, where the x-axis is pointing outwards from the page.)

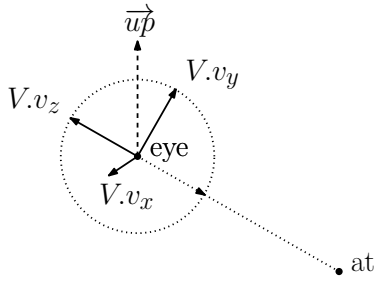

Fig. 4: The camera frame.

Let $V = (V.\vec{v}_x, V.\vec{v}_y, V.\vec{v}_z, V.o)^{\mathsf{T}}$ denote this frame, where $(\vec{v}_x, \vec{v}_y, \vec{v}_z)^{\mathsf{T}}$ are the three axis vectors for the frame and o is the origin. Clearly $V.o = eye$. As mentioned earlier, the view vector $\overrightarrow{view}$ is directed from *eye* to *at*. The z-basis vector is the normalized negation of this vector.

$$\overrightarrow{view} = \text{normalize}(at - eye) \qquad \text{and} \qquad V.\vec{v}_z = -\overrightarrow{view}$$

Next, we want to select the x-axis vector for our camera frame. It should be orthogonal to the viewing direction, it should be orthogonal to the up vector, and it should be directed to the camera's right. Recall that the cross product will produce a vector that is orthogonal to any pair of vectors, and directed according to the right-hand rule. Also, we want this vector to have unit length. Thus we choose

$$V.\vec{v}_x = \text{normalize}(\overrightarrow{view} \times \overrightarrow{up}).$$

The result of the cross product must be a nonzero vector. This is why we require that the view direction and up vector are not parallel to each other. We have two out of three vectors for our frame. We can extract the last one by taking a cross product of the first two:

$$V.\vec{v}_y = (V.\vec{v}_z \times V.\vec{v}_x).$$

There is no need to normalize this vector, because it is the cross product of two orthogonal vectors, each of unit length. Once the frame has been computed, it is an exercise in linear algebra to compute a matrix that performs this change-of-coordinates transformation. We'll skip the messy details, but if you are interested, most books on computer graphics will explain how it is done.

Projections: The next part of the process involves performing the projection. Projections fall into two basic groups, *parallel projections*, in which the lines of projection are parallel to one another, and *perspective projection*, in which the lines of projection converge a point.

In spite of their superficial similarities, parallel and perspective projections behave quite differently with respect to geometry. Parallel projections are *affine transformations*, while perspective projections are not. (In particular, perspective projections do not preserve parallelism, as is evidenced by a perspective view of a pair of straight train tracks, which appear to converge at the horizon.) Because parallel projections are rarely used, we will skip them and consider perspective projections only.

OpenGL's Perspective Projection: OpenGL provides a couple of ways to specify the perspective projection. The most general method is through glFrustum. We will discuss a simpler method called gluPerspective, which suffices for almost all cases that arise in practice. In particular, this simpler procedure assumes that the viewing window is centered about the view direction vector (the negative z-axis), whereas glFrustum does not.

Consider the following viewing model. In front of his eye, the user holds rectangular window, centered on the view direction, onto which the image is to be projected. The viewer sees any object that lies within a rectangular pyramid, whose axis is the $-z$-axis, and whose apex is his eye. In order to indicate the height of this pyramid, the user specifies its angular height, called the y *field-of-view* and denoted *fovy* (see Fig. 5). It is given in degrees.

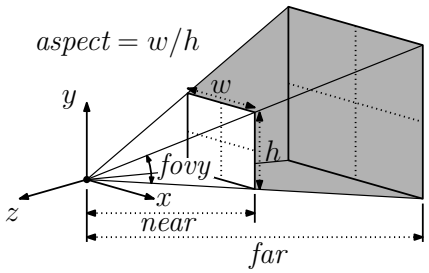

Fig. 5: OpenGL's perspective specification.

To specify the angular diameter of the pyramid, we could specify the x field-of-view, but the designers of OpenGL decided on a different approach. Recall that the *aspect ratio* is defined to be the width/height ratio of the window. The user presumably knows the aspect ratio of his viewport, and typically users want an undistorted view of the world, so the ratio of the x and y fields-of-view should match the viewport's aspect ratio. Rather than forcing the user to compute the number of degrees of angular width, the user just provides the aspect ratio of the viewport, and the system then derives the x field-of-view from this value.

Finally, for technical reasons related to depth buffering, we need to specify a distance along the $-z$-axis to the *near clipping plane* and to the *far clipping plane*. Objects in front of the near plane and behind the far plane will be clipped away. We have a limited number of bits of depth-precision, and supporting a greater range of depth values will limit the accuracy with which we can represent depths. The resulting shape is called the *viewing frustum*. (A *frustum* is the geometric shape that arises from chopping off the top of a pyramid. An example appears on the back of the US one dollar bill.) These arguments form the basic elements of the main OpenGL command for perspective.

```
gluPerspective(fovy, aspect, near, far);
```

All arguments are positive and of type GLdouble. This command creates a matrix which performs the necessary depth perspective transformation, and multiplies it with the matrix on top of the current stack. This transformation should be applied to the projection matrix stack. So this typically occurs in the following context of calls, usually as part of your initializations.

```
void myDisplay() {
```

```
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt( ... );                          // set up camera frame
  glMatrixMode(GL_PROJECTION);               // set up projection
  glLoadIdentity();
  gluPerspective(fovy, aspect, near, far);   // or glFrustum(...)
  glMatrixMode(GL_MODELVIEW);
  myWorld.draw();                            // draw everything
  glutSwapBuffers();
}
```

The function gluPerspective does not have to be called again unless the camera's projection properties are changed (e.g., increasing or decreasing zoom). For example, it does not need to be called if the camera is simply moved to a new location.

Canonical View Volume: In applying the perspective transformation, all points in projective space will be transformed. This includes point that are not within the viewing frustum (e.g., points lying behind the viewer). One of the important tasks to be performed by the system, prior to perspective division (when all the bad stuff might happen) is to clip away portions of the scene that do not lie within the viewing frustum.

OpenGL has a very elegant way of simplifying this clipping. It adjusts the perspective transformation so that the viewing frustum (no matter how it is specified by the user) is mapped to the same canonical shape. Thus the clipping process is always being applied to the same shape, and this allows the clipping algorithms to be designed in the most simple and efficient manner. This idealized shape is called the *canonical view volume* (also called *normalized device coordinates*).

The canonical view volume (after perspective division) is just a 3-dimensional rectangle. It is defined by the following constraints.

$$-1 \le x \le +1, \qquad -1 \le y \le +1, \qquad -1 \le z \le +1.$$

(See Fig. 6 for example. Imagine that the x-axis is pointing out of the paper towards you. The viewing frustum of Fig. 6(a) is mapped to the cube shown in Fig. 6(b).)

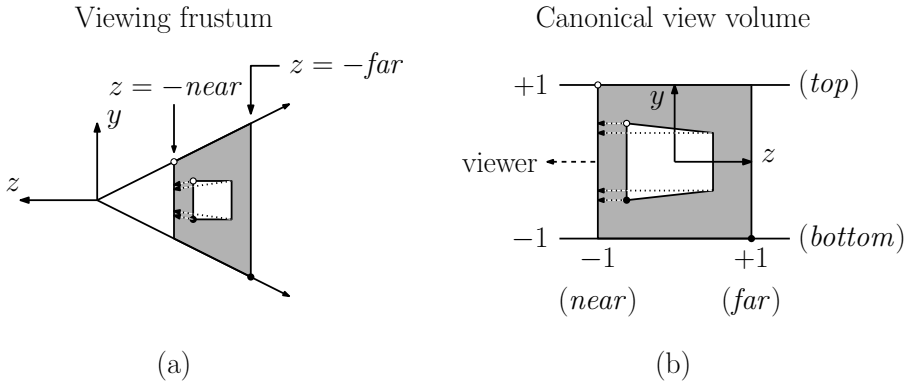

Fig. 6: Perspective with depth and the canonical view volume.

After applying this transformation, the (x, y) coordinates indicate the location of the projected point on the final viewing window. The z-coordinate is used for depth. There is a reversal of the z-coordinates, in the sense that before the transformation, points farther from the viewer have smaller z-coordinates (larger in absolute value, but smaller because they are on the negative z side of the origin). Now, the points with $z = -1$ are the closest to the viewer (lying on the near clipping plane) and the points with $z = +1$ are the farthest from the viewer (lying on the far clipping plane). Points that lie on the top (resp., bottom) of the canonical volume correspond to points that lie on the top (resp., bottom) of the viewing frustum.

Mapping to the Viewport, Rasterization, and Hidden Surface Removal: The last step of the process involves scaling the (x, y)-coordinates of each vertex in normalized device coordinates onto the viewport. The normalized device coordinates form a square $[-1, +1] \times [-1, +1]$. Given the corners of the viewport, it is a simple exercise to transform this to the corners of the viewport (see Fig. 7(a)). This transformation was derived at the end of Lecture 5.

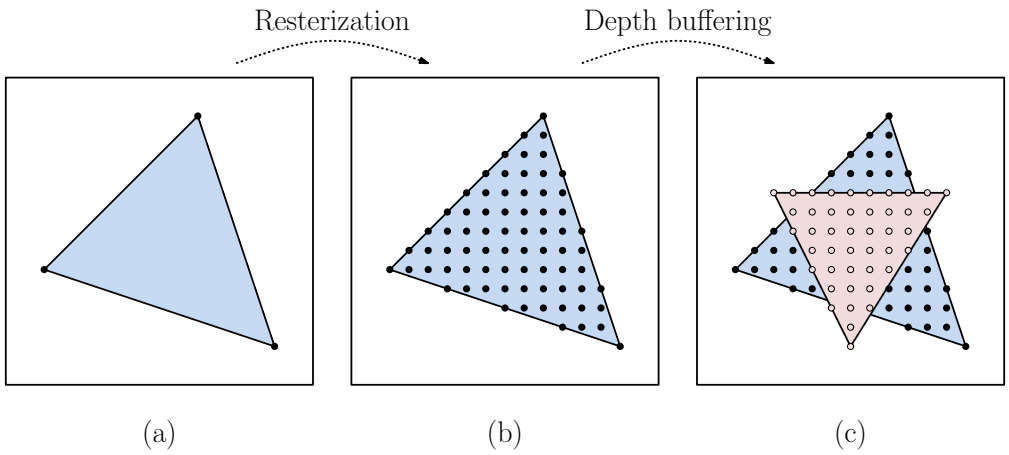

Fig. 7: Rasterization and hidden surface removal by depth buffering.

After the vertices of each polygon have been mapped to the viewport, OpenGL then converts the interior of the polygon into a collection of pixels, called *fragments*. Each fragment is associated with the color of the associated point on the polygon and its distance from the viewer (stored in the z-coordinate). These fragments are then sent to the color buffer for drawing. This is called *rasterization* (see Fig. 7(b)). If depth buffering is enabled, the z-coordinates of the fragments are stored in the depth buffer. If two fragments from different objects share the same (x, y)-coordinates then there is a conflict. If depth buffering is not enabled, fragments are written directly to the color buffer, and so you see the last one that was written. If depth buffering is enabled, then their depth values are compared, and the closer pixel is written to the color buffer.