Perspective Projections Of 3D Objects

Perspective Projection

Perspective projection is a means by which we graphically approximate the images of 3D objects onto a 2D surface, in such a way as to simulate actual visual perception (of depth and distance - objects far away appear smaller, etc...). The easiest way to think about how we convert 3D objects in the world into 2D images, is by using a camera - the resulting images are dependant on where the camera is positioned, it’s settings, etc...

The Graphics Pipeline

Definition

The OpenGL graphics pipeline is a sequence of operations that are applied to all points/vertices that are ”passed through” it. Some of these operations are determined by OpenGL commands specified by you within the code (various transformations), but some are internal to OpenGL (such as clipping and changing to the normalized view volume).

The Camera Analogy

Now imagine we have an object and a camera that are both free to move in 3D space. The movement of the object, is referred to as a modelling transformation (the composition of the object within the scene. eg. putting a bowl of fruit on the table).The movement of the camera, is referred to as a viewing transformation, and sets up the viewing volume in the world (setting up your camera and tripod and aiming at the fruit bowl). Adjustment of the camera settings (such as the zoom and the lens) is referred to as the projection transformation - this is what converts the 3D into 2D, and where the perspective would come into play (it would change the shape of the viewing volume). Clipping is done after the projection transformation - all that outside the viewing volume (outside the ”photo”) is ”cut”/clipped off so that it doesn’t appear in the final scene. The viewport transformation is basically the determination of how the photo is viewed (enlarged or stretched perhaps).
The Coordinate Systems In The Pipeline

Originally the vertices are in the object/local coordinate system. After the modelling transformation has been performed, but before the viewing transformation is performed, the coordinate system is the world coordinate system. The viewing coordinate system (eye coordinates) is that system of coordinates after the viewing transformation has been performed and before the projection transformation is performed - the x-axis is horizontal, the y-axis is vertical and the z-axis comes towards and away relative to the eye. The coordinate clipping system (clip coordinates) is that system after the projection transformation has been performed, and before the perspective projection. The normalized device coordinate system (normalized device coordinates) is that system after perspective division, but before viewport transformation. Lastly, the device coordinate system (window coordinates) is the system that results after viewport transformation.

See graphics pipeline figure

How The Graphics Pipeline Is Implemented

Mathematically, these transformations are performed by considering each point/vertex of the geometric primitive as a vector and then multiplying it by an appropriate matrix that performs the specific transformation. A vertex is passed into the pipeline by a command such as glVertex. All the matrix calculations and other concepts are explained in detail in Hill - I will only go into those regarding perspective projection.

The Modelling Transformation

Modelling transformations are performed by function calls such as glTranslate, glScale and glRotate - for more information regarding the actual matrix calculations that perform these transformations, please also refer to the notes ”The viewing pipeline” by Paul Bunn (2004 class).

The Viewing Transformation

By default, the OpenGL viewpoint is situated at the origin, looking towards the negative z-axis. We can view an object in the space by one of two ways - either the object must be moved so that it can be seen from the default viewpoint (using translates and rotates), OR the viewpoint must be moved to a new location from which the object can be seen. The latter method is usually preferred, as numerous translates and rotates can become rather complicated and tedious. It is important to note that modelling and viewing transformations have an inverse relationship regarding rotation direction.

Viewing Transformation - OpenGL Commands

GLdouble eyeX, eyeY, eyeZ;    /* viewpoint */
GLdouble referX, referY, referZ;    /* reference point */
GLdouble upX, upY, upZ;    /* view up vector */
gluLookAt(eyeX, eyeY, eyeZ, referX, referY, referZ, upX, upY, upZ);
Parameters:

**eyeX, eyeY, eyeZ**: The position of the eye point (where you are looking from).

**referX, referY, referZ**: The position of the reference point (centre of the scene/centre of where you are looking).

**upX, upY, upZ**: The direction of the up vector (usually the y-axis - use 0,1,0 if that is the case).

Due to the mapping performed by the viewing matrix (created by gluLookAt), when a typical projection matrix is used, the centre of the scene maps to the centre of the viewport. Additionally, the direction described by the up vector (which cannot be parallel to the line of sight from the eye to the reference point) is mapped so that it points upwards in the viewport.

**How To ”Move The Camera”**

Supposing the eye point is fixed:

To **pan the camera** (like shaking your head left and right): Move the reference point ”horizontally”.

To **tilt the camera** (like nodding your head up and down): Move the reference point ”vertically”.

To **rock the camera** (like tilting your head left and right): Let the eye point - the reference point be the normal vector of a plane A; change the direction of the up vector so that it’s projection onto A ”rotates” left and right.

These all correspond to rotating the viewing coordinate system along the y-, x- and z-axis respectively.

**The Projection Transformation**

OpenGL allows for two different kinds of projection transformation - namely orthographic and perspective. Both of these transformations define a volume of space called a **frustum** - the main difference between the two, is the shape of the defined frustum. For orthographic projection (a parallel projection), it’s a rectangular parallelepiped and for perspective projection, it’s a truncated pyramid. For orthographic projection, the scale is in three dimensions, but for perspective, the x and y have a depth dependent scale and there is a nonlinear change in z. Only that geometry which is inside the frustum is displayed on the screen, as that which is outside the frustum is clipped in the next stage of the pipeline.

**Orthographic Projection - OpenGL Commands**

```
GLdouble left, right, bottom, top, zNear, zFar;
glOrtho(left, right, bottom, top, zNear, zFar);
```

Parameters:

**left, right**: The coordinates for the left- and right-vertical clipping planes.

**bottom, top**: The coordinates for the bottom- and top-horizontal clipping planes.

**zNear, zFar**: The distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.
Perspective Projection - OpenGL Commands (gluPerspective)

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

Parameters:
fovy: The field of view angle, in degrees, in the y-direction (centred about y = 0).
aspect: The aspect ratio that determines the field of view in the x-direction. The aspect
ratio is the ratio of x (width) to y (height). Should match the aspect ratio of the associated
viewport to avoid distortion.
zNear: The distance from the viewer to the near clipping plane (always positive - can’t
be 0).
zFar: The distance from the viewer to the far clipping plane (always positive).

Perspective Projection - OpenGL Commands (gluFrustum)

GLdouble left, right, bottom, top, zNear, zFar);
glFrustum(left, right, bottom, top, zNear, zFar);

Parameters:
left, right: The coordinates for the left- and right-vertical clipping planes.
bottom, top: The coordinates for the bottom- and top-horizontal clipping planes.
zNear, zFar: The distances to the near- and far-depth clipping planes. Both distances
must be positive.

The Difference Between Using gluPerspective And gluFrustum

Both gluPerspective and gluFrustum produce perspective projection matrices that you
can use for projection transformations (ie. transformation from eye coordinates to clip
coordinates). The main difference between these two commands, is that gluFrustum is
more general and allows for off-axis projections, whereas gluPerspective only produces
symmetrical/on-axis projections. There is no real advantage of using one command over
the other, but glFrustum can be used in cases where gluPerspective can’t - for example,
in stereo views, tiled renderings and projection shadows. In fact, glFrustum can actually
be used to implement gluPerspective.

How To Make A Call To glFrustum To Match That Of A Call To
gluPerspective

The field of view (fovy) of the glFrustum(...) call is:
fovy × 0.5 = arctan(((top − bottom) × 0.5) ÷ near)
Since bottom ≡ -top for the symmetrical projection that gluPerspective(...) produces,
then:
top = tan(fovy × 0.5) × near and bottom = −top
Note: fovy must be in radians for the above formulae to work with the C math library.
If the fovy has been computed in degrees (as in the call to gluPerspective(...)), then
calculate top as follows:
top = tan((fovy × 3.14159) ÷ 360.0) × near
The left and right parameters are simply functions of the top, bottom, and aspect:
left = aspect × bottom and right = aspect × top
Perspective Projection In Terms Of Matrices

The OpenGL command `glFrustum(left, right, bottom, top, zNear, zFar)`; results in the matrix calculation $C' = CF$ being performed, where $C'$ is the transformed/new current matrix, $C$ is the current matrix and $F$ is the projection matrix that OpenGL creates when `glFrustum` is called.

$$F = \begin{pmatrix}
\frac{2 \times |z_{Near}|}{right - left} & 0 & \frac{right - left}{right - left} & 0 \\
0 & \frac{2 \times |z_{Near}|}{top - bottom} & \frac{top - bottom}{top - bottom} & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & -\frac{(2 \times |z_{Far}| \times |z_{Near}|)}{|z_{Far}| - |z_{Near}|}
\end{pmatrix}$$

The OpenGL command `gluPerspective(fovy, aspect, near, far)`; sets up the same matrix $F$ as that generated by `glFrustum(...)`, but the values of top, bottom, right and left are computed as follows:

- $top = z_{Near} \times \tan(\frac{\pi}{180} \times \frac{fovy}{2})$
- $bottom = -top$
- $right = top \times aspect$
- $left = -right$

Clipping And Normalization

As previously mentioned, both clipping and normalization of the view volume, are operations that are internal to OpenGL. Thus no commands are required to be specified. Normalization of the view volume is simply achieved as normal through division, but for more information on clipping, please see the notes ”3D Clipping” by Terrence Naidoo (2004 class).

Viewport Transformation

The viewport is a portion of the screen window, which is open on the screen. Viewport transformations are calculated internally based on the OpenGL call to `glViewport`, and involve translations and scaling to fit the window.

Viewport Specification - OpenGL Commands

```
glint x, y;  
glsizei width, height;  
glViewport(x, y, width, height);
```

Parameters:
- $x, y$: The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).
- $width, height$: The width and height, respectively, of the viewport. When an OpenGL context is first attached to a window, width and height are set to the dimensions of that window. ie. by default, the viewport is the whole screen window.
How To Implement Perspective Projection Of 3D Objects In OpenGL

The graphics pipeline in terms of matrices can be represented as
\[ v' = P V M v, \] where \( P \) is the projection matrix, \( V \) is the viewing matrix, \( M \) is the modelling matrix, \( v \) is the vertex coordinate and \( v' \) is the transformed vertex coordinate. This is due to the fact that the matrix that is farthest to the right is applied to the vertex first, thus applying first modelling transformation, then viewing transformation and finally projection transformation on the vertex coordinate as required. Since the order of the transformations is thus significant, the OpenGL functions must be invoked in the reverse order in which they will be applied.

The correct order of OpenGL code:

```c
/* Viewport Transformation */
glViewport(...); /* set the screen viewport */

/* Projection Transformation */
glMatrixMode(GL_PROJECTION); /* specify the projection matrix */
glLoadIdentity(); /* initialize current value to identity */
gluPerspective(...); /* or glOrtho(...) for orthographic */
/* or glFrustum(...), also for perspective */

/* Viewing And Modelling Transformation */
glMatrixMode(GL_MODELVIEW); /* specify the modelview matrix */
glLoadIdentity(); /* initialize current value to identity */
gluLookAt(...); /* specify the viewing transformation */

glTranslate(...); /* various modelling transformations */
glScale(...);
glRotate(...);
...
```