**Projections and their transformations**

We have looked at modeling 3-dimensional figures in a 3-dimensional world. The final image, however, will be displayed on a 2-dimensional surface.

The transformation from the 3-dimensional model space to the 2-dimensional viewing space is accomplished by projecting the model onto the view or projection plane.

Here is where our “virtual camera” model offers a good analogy.

Modeling transforms position the model. Viewing transforms position the camera. Projection transforms control the type of lens, the focal length, etc...

Things are really a bit different, however, because with computer graphics we have rather more freedom than we have with a camera.
There are two basic projections we will discuss — parallel projections and perspective projections.

**Parallel projections**
The basic property of a parallel projection is that features are projected by parallel rays onto the viewing surface.

There are again two types of parallel projections — projections in which the projection is parallel to the normal of the viewing surface are orthographic projections; projections where the projection rays are not normal to the surface are oblique projections.
A common set of orthographic projections are the front, side, and top views typical of architectural plans. In this case the projections are parallel to the model’s principal axes.

Orthographic projections preserve the distance along lines which are parallel to the plane of projection. Thus, one can take measurements along such lines directly from their projections. Also, lines which are parallel will be projected to parallel lines. The isometric projection has the direction of projection at the same angle to all three principal axes.

The isometric projection has the property that all 3 axes can be measured on the same scale. For an isometric projection of cube, all projected lines have the same length.
Isometric projections are a special case of axonometric orthographic projections. Such projections use projection planes that are not normal to a principal axis. Axonometric orthographic projections give the illusion of perspective but do not exhibit true perspective foreshortening, where the length of projected lines are dependent upon their distance from the projection plane.
Such projections have been popular in video games for many years.

[From urlwww.kirupa.com/developer/isometric/perspective.htm]


**Oblique projections**

Oblique projections are the most general parallel views. In an oblique projection, the projectors make some arbitrary angle with the projection plane.

These views are somewhat unnatural, but can be used to convey more information than orthographic projections.

The human eye, and most other physical viewing devices, have a lens fixed with respect to (and usually parallel to) the viewing plane. This is what makes oblique projections seem unnatural.
**Perspective projections**

In a perspective projection, the projecting rays are not parallel, but instead pass through one or more **vanishing points** or points at infinity.

![Perspective projection diagram]

Note that lines which are parallel in the model actually converge in the “far distance.” This creates an illusion of depth in a 2-dimensional figure.
Two point perspective would use 2 vanishing points, roughly orthogonal to each other in the “model world.” Three point perspective would use 3 vanishing points.

[From www.csse.monash.edu.au/~aland/notes]

Note that OpenGL has no concept of the number of vanishing points. This is a notion from the worlds of classical art and drafting.

In a perspective projection, we need to specify a point as the center of projection; for parallel projections we needed to specify a direction of projection.
The viewing surface

The plane onto which the image is projected is usually called the **view plane**. It has coordinate axes \( u, v, \) and \( n \) orthogonal to each other.

It is defined by the **view reference point** or VRP, and the **view plane normal** or VPN and points along the \( n \) axis.

The view plane corresponds to the “film” in our virtual camera. We also need to describe the orientation of the film or **viewport** in the view plane. This is done by specifying a vector called the **view up vector** (VUP). The projection of the VUP vector defines the \( v \) axis in the plane. This, in effect, orients the camera. The \( u \) axis is perpendicular to \( v \) and \( n \) (\( u = v \times n \)). Defining minimum and maximum values of \( u \) and \( v \) determines the viewport.
In many graphics systems, there are functions which set the viewing parameters explicitly, using functions like:

```plaintext
set_view_reference_point(x, y, z)
set_view_plane_normal(nx, ny, nz)
set_view_up(vx, vy, vz)
```

In OpenGL, there is a convenient function which is rather more direct, and uses the virtual camera model explicitly. Basically, the position of the camera (or eye), and the position of the object are specified. These specify the VPN and VRP. In fact, these are specified in the `MODELVIEW` transformation.

The orientation of the camera also needs to be specified (VUP).

```plaintext
gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)
```

`eyex`, `eyey`, and `eyez` specify the camera (or eye) position, `atx`, `aty`, and `atz` specify the object center.

`upx`, `upy`, and `upz` specify VUP.

The `glu..` functions are not OpenGL functions, but are part of a standard library which forms part of most OpenGL systems. (GLUT is another library which provides user interaction.)

The tutorial `projection` shows the use of the projection functions for orthogonal and perspective transformations.
Clipping

Viewing involves both projection and clipping. In three dimensions, we must clip against a 3-dimensional view volume.

For the parallel projection case, the additions to the 2-dimensional clipping we discussed earlier are a front clipping plane and a back clipping plane, forming a clipping rectangular volume.

The perspective projection case is a bit different. Here, the front and back clipping planes are parallel to the view plane, but are of different sizes. This shape is known as a frustum.
Parallel projections in OpenGL

In OpenGL, there is only one parallel viewing function, the orthographic viewing function

\texttt{glOrtho(left, right, bottom, top, near, far)}

where \((\texttt{left}, \texttt{bottom}, -\texttt{near})\) are the coordinates of the lower front corner of the rectangular volume, and \((\texttt{right}, \texttt{top}, -\texttt{far})\) are the coordinates of the back top corner.

\texttt{near} or \texttt{far} are the distances of the near and far clipping planes from the eye. If either \texttt{near} or \texttt{far} is negative, this means that the corresponding clipping plane is located behind the eye.

For example, for a view volume consisting of a 2x2x2 cube centered at the origin, the function calls would be

\begin{verbatim}
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
\end{verbatim}
Perspective projections in OpenGL

There are two functions for perspective projections.

\texttt{glFrustum(left, right, bottom, top, near, far)}

defines a frustum similar to the rectangular volume for \texttt{glOrtho(\ldots)}.

Note that \texttt{near} and \texttt{far} must be positive for \texttt{glFrustum}.

A typical use would be

\begin{verbatim}
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 10.0);
\end{verbatim}

Here the front clipping plane is (-1, -1, -1) to (1, 1, -1). The rear clipping plane is 9 units beyond the front clipping plane with corners (-10, -10, -10) to (10, 10, -10).

While conceptually easy, \texttt{glFrustum()} can be difficult to use.
An alternate way to determine the view volume is by specifying an angular field (in the y direction, in this case), an aspect ratio (width/height) and the distance of the near and far clipping planes.

\[
gluPerspective(\text{fovy}, \text{aspect}, \text{near}, \text{far})
\]

With this function, it is necessary to pick appropriate values for the field of view, or the view looks distorted.

With both \text{gluPerspective()} and \text{glFrustum()} rotations and translations can be applied to change the default orientation of the viewing volume.

Without such transformations, the viewpoint remains at the origin, with the line of sight along the negative z-axis.

The tutorial \textit{projection} illustrates the use of \text{gluPerspective()}. 
Simple orthogonal projection

Let us consider a projection onto a plane perpendicular to the $z$-axis, at $z = 0$.

In this case, points retain their $x$ and $y$ values, and the equations of projection are

$$x_p = x, \ y_p = y, \text{ and } z_p = 0,$$

In homogeneous coordinates,

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

We can use the transformation matrices to further modify this projection matrix.
Simple perspective projection

We will again consider the case where the projection is on a plane on the $z$-axis, at a point $(0, 0, z_p)$

Here, figure (b) is an orthogonal projection of (a) looking directly along the $y$ axis.

By similar triangles, $x/z = x_p/d$

Similarly, $y/z = y_p/d$

So, $x_p = \frac{x}{z/d}$ and $y_p = \frac{y}{z/d}$

It is possible to find a matrix which accomplishes the perspective projection on the $z$-axis. We must relax the condition that $W = 1$ in our homogeneous coordinate system, however.
The transformation matrix

\[ M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \]

Transforms the point \((x, y, z, 1)\) into \((x, y, -z, z/d)\)

Homogenizing (i.e. dividing by \(W = z/d\)), gives

\[ \left( \frac{x}{z/d}, \frac{y}{z/d}, -d, 1 \right) \]

which is the required transformation.

All of our usual transformations (translation, rotation, scaling, shearing) were affine, meaning that they preserved parallel lines. This transformation is not affine, nor is it invertible — all points along a projector map into the same point.

Again, it is possible to transform this matrix to project in an arbitrary direction.
The viewing pipeline

What happens, in OpenGL, in order to view an image?

The objects themselves are described in their own coordinate spaces, and placed in the global model world by the appropriate MODELVIEW transformation. After transformation all vertices are expressed in eye coordinates.

The PROJECTION matrix applies the requested projection. However, under the hood, OpenGL also applies projection normalization which maps the model world into a canonical clipping volume (more on this below). After projection, vertices are expressed in clip coordinates.

We saw above in “Simple perspective projection” that projec-
tion requires both multiplication by a transformation matrix and a division step (a.k.a renormalization). Thus, perspective division forms the third stage of the pipeline. After this stage we have vertices in normalized device coordinates.

Finally, the viewport transformation maps the projected vertices onto window coordinates. Window coordinates are measured in units of pixels on the display, with the origin in the lower-left corner. In fact, window coordinates maintain depth information which is necessary for hidden surface removal.

**Projection normalization?**

Whatever view volume is specified by calls to `glOrtho` or `glFrustum` (or `gluPerspective`), OpenGL transforms that volume into the canonical view volume. This volume is defined by planes at $x \pm 1$, $y \pm 1$, and $z \pm 1$. The canonical view volume may be generated directly as follows:

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```
For a general orthographic projection, the projection normalization does the following:

Perspective projections will also be converted to the same canonical view volume.

where COP means centre of projection. On the left is a top-down view of a cube within a frustum-shaped view volume. After projection normalization, the view volume has been made canonical, and the cube has been appropriately distorted.
The purpose of projection normalization is to simplify clipping. It is easier to transform primitives to the canonical viewing volume and then clip, than to clip them beforehand to an arbitrary view volume.
The planet.c example revisited...

Look at the `display` call below. What is the plane of the planet’s travel about the sun?

```c
void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);

    glPushMatrix();
    glutWireSphere(1.0, 20, 16); /* draw sun */
    glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
    glutWireSphere(0.2, 10, 8);    /* draw smaller planet */
    glPopMatrix();
    glutSwapBuffers();
}
```
The *reshape* function below positions the eye within the \(xz\) plane

```c
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

Let’s say that we want to zoom in a little, so that when *year* is either 0 or 180, the outside of the planet just touches the window’s edge.

By examining the code in *display* we can see that, in eye coordinates, the planetary system has a radius of 2.2 (extending out to the planet’s outer edge). Examining the arguments to *gluLookAt* we can see that the system is 5 units away from the eye. Simple trigonometry tells us that the horizontal field of view, \(\text{fov}_x\), is
We define a variable \texttt{aspect} as follows

\[
\texttt{aspect} = \frac{\text{width}}{\text{height}}
\]

\[
\texttt{aspect} = \frac{\text{angular width}}{\text{angular height}}
\]

\[
\texttt{aspect} = \frac{\texttt{fovx}}{\texttt{fovy}}
\]

Therefore, we can find the required value of \texttt{fovy} as follows

\[
\texttt{fovy} = \frac{\texttt{fovx}}{\texttt{aspect}}
\]
The following code for `reshape` implements this zoomed-in view.

```c
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    double aspect = w / (GLfloat) h;
    double fovx = 47.5;      /* 2*arctan(2.2/5.0)=47.5 */
    double fovy = fovx / aspect;
    gluPerspective(fovy, aspect, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

Note that when executing the example code (in `planet2.c`) the planet is clipped out when close to the viewer. Why is this?