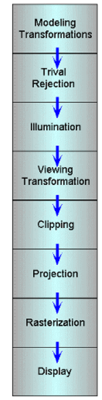
Three-Dimensional Transformations

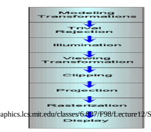
- The 3-D graphics pipeline
- Rigid-body transforms
  Hearn & Baker pp. 408-429 (Study)
- Homogeneous coordinates
  Hearn & Baker pp. 188-190 (Study)
- Viewing transformations
  Hearn & Baker pp. 432-438 (Study)
- Projection
  Hearn & Baker pp. 438-456 (Study)

The 3-D Graphics Pipeline



- Almost every disscusion of 3-D graphics begins here
- Seldom are any two versions drawn the same way
- Seldom are any two versions implemented the same way
- Primitives are processed in a set series of steps
- Each step forwrds its result on to the next step

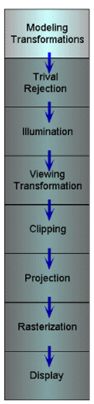
Nearly all discussions of three-dimensional computer graphics start with the graphics pipeline. However, the notion of "the graphics pipeline" is somewhat of a misnomer. Seldom are any two depictions, or implementations for that matter, of a graphics pipeline identical.



The basic idea is that the processing of information in 3-dimensional computer graphics occurs in a series of steps, where each step generates results for the successive one.

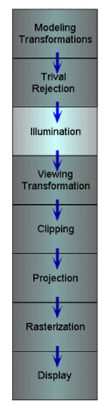
The process starts with models that are usually described by three-dimensional coordinates *(x, y, z)*. The models usually define either solids or the boundaries of object. Each of these modeling styles has a name. In *solid modeling* the primitives used to describe an object have volume. In *boundary representations, or B-reps*, the objects are defined in terms of their skins, or two-dimensional surfaces that approximate the interfaces between different material types. These models are usually defined within there own coordinate system, with their own origins and scales of measurement.

Modeling transformations



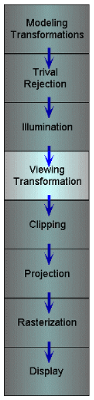
- We starts with 3-D models defined in their own *model space*
- **Modeling transformations** orient models within a common coordinate system called *world space*
- All objects, light sources, and the viewer live in *world space*
- **Trivial rejection** attempts to eliminate objects that cannot possibly be seen (an optimization)

Illumination



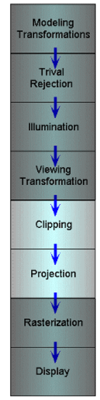
- Next we **illuminate** potentially visible objects
- Object colors are determined by their material properties, and the light sources in the scene
- Illumination algorithm depends on the shading model and the surface model
- More about this in later on

Viewing Transformation

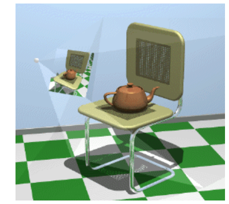


- Another change of coordinate systems
- Maps points from *world space* into *eye space*
- Viewing position is transformed to the origin
- Viewing direction is oriented along some axis
- A viewing volume is defined

Clipping and Projection

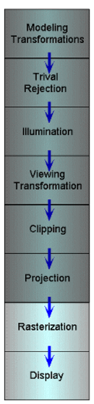


- Next we perform clipping of the scene's objects againist a three dimensional viewing volume called a *viewing frustum*
-



- This step totally eliminates any objects (and pieces of objects) that are not visible in the image
- A clever trick is used to straighten out the viewing frustum in to a cube
- Next the objects are projected into two-dimensions
- Transformation from eye space to *screen space*

Rasterization and Display



- One last transformation from our screen-space coordinates into a *viewport coordinates*
- The rasterization step scan converts the object into pixels
- Involve interpolating parameters as we go
- Purely 2D operation

Almost every step in the rendering pipeline involves a change of coordinate systems. Transformations are central to understanding three-dimensional computer graphics.

Rigid-Body Transformations

- Rigid-body, or *Euclidean transformations*
- Preserve the shape of the objects that they act on
- Includes rotations and trnaslations (just as in two dimensions)

First we begin with a representation for 3-D points. We will consider points as column vectors. Thus, a typical point with coordinates *(x, y, z)* is represented as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This is not the only possible representation. You may encounter textbooks that consider points as row vectors. What is most important is that you use a consistent representation.

Vector and Matrix Algebra

You've probably been exposed to vector algebra in previous courses. These include *vector addition*, the *vector dot product*, and the *vector cross product*. Let's take a minute to discuss an equivalent set of matrix operators.

We begin with the *dot product*. This operation acts on two vectors and returns a scalar. Geometrically, this operation can be described as a projection of one vector onto another. The dot product has the following matrix formulation.

$$\bar{a} \cdot \bar{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \alpha$$

Cross Product in Matrix Form

The vector cross product also acts on two vectors and returns a third vector. Geometrically, this new vector is constructed such that its projection onto either of the two input vectors is zero.

$$\bar{a} \times \bar{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \bar{c} \qquad \begin{matrix} \bar{a} \cdot \bar{c} = 0 \\ \bar{b} \cdot \bar{c} = 0 \end{matrix}$$

In order for one vector to project onto another with a length of zero, it must either have a length of zero, or be *perpendicular* to the second vector. Yet the vector generated by the cross-product operator is perpendicular to two vectors. Since the two input vectors define a plane in space, the vector that results from the cross product operation is perpendicular, or *normal* to this plane.

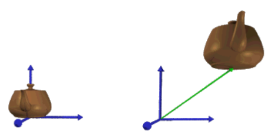
For any plane there are two possible choices of a normal vector, one on each side of a plane. The cross product is defined to be the one of these two vectors where the motion from the tip of the first input vector to the tip of the second input vector is in a counter-clockwise direction when observed from the side of the normal. This is just a restatement of the right-hand rule that you are familiar with.

Translations

Objects are usually defined relative to their own coordinate system. We can *translate* points in space to new positions by adding offsets to their coordinates, as shown in the following vector equation.

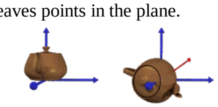
$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

The following figure shows the effect of translating a teapot.



Rotations

Rotations in three-dimensions are considerably more complicated than two-dimensional rotations. In general, rotations are specified by a rotation axis and an angle. In two-dimensions there is only one choice of a rotation axis that leaves points in the plane.



There are several different ways to present rotations. I will use a different approach than that used in the book. In the book all possible rotations are treated as the compostion of three canonical rotations, one about the x-axis, one abount the y-axis and one about the z-axis. In order to use this model you need to do the following. Memorize the three canonical rotations, which aside from the signs of the sines, isn't too hard. An then you have to go through a series of rotations which move the desired rotation axis onto one of your canonical rotations and then you have to rotate it back without introducing any extraneous twists. **This is a difficult and error-prone process. But worst of all it is ambiguous. There exist several different combinations canonical rotations that result in the same overall result.**

Decomposing Rotations

There is another way that I beleive is both easier to understand and provides you with more insights into what rotation is really about. Instead of specifying a rotation by a series of canonical angles, instead we will specify an arbitrary axis of rotation and an angle.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \left(\textbf{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)(1-\cos\theta) + \textbf{Skew}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)\sin\theta + \mathbf{I}\cos\theta \right)\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

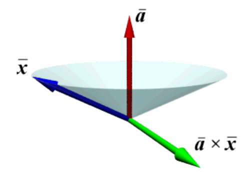
The vector *a* specifies the axis of rotation. *This axis vector must be normalized.* The rotation angle is given by θ.

You might ask "How am I going to remember **this** equation?". However, once you understand the geometry of rotation, the equation will seem obvious.

The basic idea is that *any rotation can be decomposed into weighted contributions from three different vectors*.

The Geometry of a Rotation

These vectors are rotation axis, a vector that is perpendicular to both the rotation axis and the point being rotated, and the point itself. These vectors correspond to the each respective term in the expression.



Let's review each term in greater detail.

The Symmetric Matrix

First we'll see why the symmetric matrix of a vector generates a vector in the the direction of the axis.

$$\text{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\begin{bmatrix} a_x & a_y & a_z \end{bmatrix} = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}$$

The symmetric matrix is composed of the outer product of a row vector and an column vector of the same value.

$$\text{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \bar{a}(\bar{a} \cdot \bar{x})$$

The Skew Symmetric Matrix

Next, consider how the skew symmetric matrix of a vector generates a vector that is perpendicular to both the axis and it's input vector.

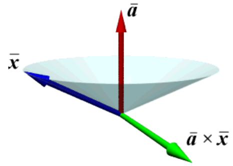
$$\text{Skew}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

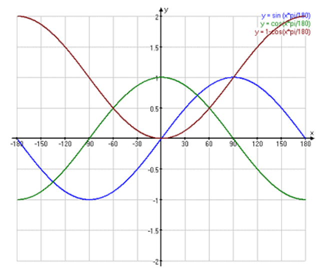
$$\text{Skew}(\bar{a})\bar{x} = \bar{a} \times \bar{x}$$

From the idenity matrix in the third term it is easy to see how it will generate a vector in the same direction as the input vector.

Weighting Factors

Even the weighting factors can be easily explained in terms of the component three vectors.





- When θ = 0 then the sum should reduce to the identity matrix
- The resulting vector after the rotation will always be in the same direction as the axis. Thus, its weight is always positive *(1 -* cos*(θ))*

Some Sanity Checks

We can now test our new found knowledge. First, consider a rotation by 0.

$$Rotate\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, 0\right) = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}(1-1) + \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} 0 + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} 1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can also derive the rotation matrices given in the book. For instance, a rotation about the x-axis:

$$Rotate\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}\sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta$$

$$Rotate\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

More Sanity Checks

A rotation about the y-axis.

$$Rotate\left(\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta$$

$$Rotate\left(\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

This approach to rotations is completely general and unambiguous.

More Sanity Checks

Rotations about the z-axis.

$$Rotate\left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta\right) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta$$

$$Rotate\left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta\right) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you grasp this simple concept and understand the geometry, then don't be surprized if, in a few years, everyone is coming to you to compute their rotation matrices. They will also drop their jaws in amazement when you just write down the matrix.

Homogeneous Coordinates

Just as in the 2-D case, 3-D rotations operations do not compose together well. They require a mixture of matrix multiplications and vector additions.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} r'_{11} & r'_{12} & r'_{13} \\ r'_{21} & r'_{22} & r'_{23} \\ r'_{31} & r'_{32} & r'_{33} \end{bmatrix} \left(\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \right)$$

Once again we can solve this problem by embedding our 3-D space within a 4-dimensional space that excludes the origin. The origin has several properties that make it undesirable. For instance, it is perpendicular to all vectors. We will also normalize points in this 4-D space by identifying all points that lie on lines through the origin. If we exclude the origin from this space then every point is uniquely defined.

$$\mathbb{P}^n = \mathbb{R}^{n+1} - \bar{0} \Big| \text{ where } \bar{x} \equiv \lambda \bar{x}$$

This type of representation is called a *projective space*.

Projective Points

Projective space adds a new coordinate to each point, but not a dimension. Each point in projective space has many possible representations all of which vary by a non-zero scale factor.

We can identify a canonical representative for each point, by scaling the point so that its fourth coordinate is always a 1.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This actually excludes a few points, (a planes-worth, in fact), which have a fourth component that equals 0. In our flat model of projective space these special points can be regarded as pure directions, sometimes called *points at infinity*.

Homogeneous Transformations

The addition of an extra coordinate to each of our points also enlarges our transformation matrices to 4 x 4.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_x \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

It also permits translations to be specified as a matrix operation. As a result we can compose any combination of transformations as a matrix product. This approach exactly parallels the approach we took with 2-D transformations.

$$\bar{x}' = \mathbf{M}_1 \mathbf{M}_2 \mathbf{M}_3 \cdots \mathbf{M}_n \bar{x}$$

More Modeling Transforms

It is possible to expand our repertoire of modeling transformations in a way parallel to our development of 2-D transformations (We can add Similitudes, Affines, and Projective groups of operators). However, most of these are either not that useful in everyday practice, or they are commonly reduced to one or two special cases, which generalize via a series of more typical operations.

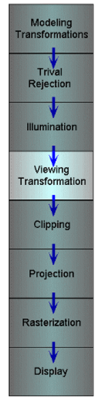
Among, those transforms remaining scales and shears are the most common:

$$Scale(\sigma_x, \sigma_y, \sigma_z) = \begin{bmatrix} \sigma_x & 0 & 0 & 0 \\ 0 & \sigma_y & 0 & 0 \\ 0 & 0 & \sigma_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad Shear(\kappa_{xy}, \kappa_{xz}, \kappa_{yz}) = \begin{bmatrix} 1 & \kappa_{xy} & \kappa_{xz} & 0 \\ 0 & 1 & \kappa_{yz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can generate any 3-D affine transformation using a combination of a rotation, a scale, a shear, and a translation.

Viewing Transformations

Strictly speacking, there is no real need for a special set of viewing transfromations. We can always compose one using a combination of rotations and translations. However, they occur so frequently that it is useful to develop a special class of transfromations specifically for the purpose of mapping points from *world space* to *eye space*.



Some Motivation

Let's look at an example.

In the scene above the origin of world space is shown as a blue coordinate frame located under the chair. This was a convenient coordinate system for orienting the floor and walls (Notice how the axes are aligned with the floor tiles). The chair and teapot could also be easily placed and oriented in this coordinate system.

The goal of a viewing transformation is to specify the position and orientation of our *3-D graphics camera* in the scene. However, its effect is almost the opposite.

Camera-Centered View

We will find that it is much more useful to reorient the entire scene such that the camera is located at the origin. Moreover, if we align the scene so that its optical axis (the direction it is looking) is along one of the coordinate axes and twist the scene so that the desired *up* direction is aligned with our camera's up direction we can greatly simply the clipping and projection steps that follow.

Once more, the mechanics of this specification can be expressed using the rigid body transformations discussed before. First, we need to perform the rotations needed to align the two coordinate frames.

Viewing Steps

First, we need to perform the rotations needed to align the two coordinate frames.

Then we need to perform a translation that will move the origin of our world space to the camera's origin. (Why did I rotate first and then translate?)

A More Intuitive Approach

It is more natural to position the camera in the world space as if it was another object.

Suppose that we identify a point where the camera is located (in world space), and call it the *eye point*. Then we identify some other world-space point in the scene that we wish to appear in the center of our view. We'll call this point the *look-at* point. Next, we identify a world space vector that we wish to be oriented upwards in our final image, and this point we'll call the *up-vector*.

This specification allows us to specify an arbitrary camera path by changing only the eye point and leaving the look-at and up vectors untouched. Or we could pan the camera from object to object by leaving the eye-point and up-vector fixed and changing only the look-at point.

Deriving a Natural Viewing Transform

As discussed before, we will compute the rotation part of the viewing transformation first. Fortunately, we know some things about the rotation matrix that we are looking for.

For instance, consider a vector along the look-at direction:

$$\begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} = \begin{bmatrix} lookat_x \\ lookat_y \\ lookat_z \end{bmatrix} - \begin{bmatrix} eye_x \\ eye_y \\ eye_z \end{bmatrix}$$

After normalizing it

$$\hat{l} = \frac{\bar{l}}{\sqrt{l_x^2 + l_y^2 + l_z^2}}$$

We expect our desired rotation matrix to map it to the vector *[0, 0, -1]*^{T} (Why?).

$$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} = \mathbf{V}\hat{l}$$

Deriving a Natural Viewing Transform

There is another special vector that we can compute. If we find the cross product between the look-at vector with our up vector, we will get a vector that points to the right.

$$\bar{r} = \bar{l} \times \overline{up}$$

We expect that this vector when normalized will transform to the vector *[1, 0, 0]*^{T}.

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{V}\frac{\bar{r}}{\sqrt{r_x^2 + r_y^2 + r_z^2}}$$

One More Vector

Finally, from these two vectors we can sythesize a third vector in the up direction.

$$\bar{u} = \bar{r} \times \bar{l}$$

And this vector, when normalized, should transform to *[0, 1, 0]*^{T}.

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{V}\frac{\bar{u}}{\sqrt{u_x^2 + u_y^2 + u_z^2}}$$

Now we know everything that our viewing matrix must do. Now we need only to assemble it.

Composing a Result

Now lets consider all of these results together.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{V}\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix}$$

In order to compute the matrix, **V** we need only compute the inverse of the matrix formed by concatenating our 3 special vectors. We will instead employ a little trick from linear algebra.

Remember that each of our vectors are unit length (we normalized them). Also, each vector is perpendicular to the other two. These two conditions on a matrix makes it, **orthogonal**. Rotations are also orthogonal. Orthonormal matrices have the unique property that:

$$\mathbf{V}^{-1} = \mathbf{V}^{T} \quad \text{if } \mathbf{V} \text{ is orthonormal}$$

Rotations are Easy to Invert

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{V}\begin{bmatrix} \hat{r} & \hat{u} & -\hat{l} \end{bmatrix}$$

Therefore, the rotation component of our viewing transformation is just the transpose of the matrix formed by our selected vectors.

$$\mathbf{V}_{rotate} = \begin{bmatrix} \hat{r} \\ \hat{u} \\ -\hat{l} \end{bmatrix}$$

Now for the Translation

The rotation that we just derived is specified about the eye point. However, we are still in world space at this point. In order for the rotation to occur about the correct point if we must first subtract the eye point from any given world space point.

$$\begin{bmatrix} \hat{r} \\ \hat{u} \\ -\hat{l} \end{bmatrix}\begin{bmatrix} x - eye_x \\ y - eye_y \\ z - eye_z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

Next we isolate the rotation and translation parts.

$$\begin{bmatrix} \hat{r} \\ \hat{u} \\ -\hat{l} \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} \hat{r} \\ \hat{u} \\ -\hat{l} \end{bmatrix}\begin{bmatrix} eye_x \\ eye_y \\ eye_z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

The Viewing Transformation

Finally we can compose the rotation and translation into a single 4 by 4 matrix giving **V**:

$$\begin{bmatrix} & \hat{r} & & -\hat{r}\cdot\overline{eye} \\ & \hat{u} & & -\hat{u}\cdot\overline{eye} \\ & -\hat{l} & & \hat{l}\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

And the viewing transform is complete:

Projection Transformation

Our lives are greatly simplified by the fact that viewing transformations transform the eye to the origin and the look-at direction (optical axis) to a specified coordinate axis. This greatly reduces the range of possible projection matrices.

The projection transformation maps all of our 3-D coordinates onto our desired viewing plane. Thus, making our 3-D world into an image. We've already discussed this process when we considered 2-D projective transformations.

Orthographic Projection

The simplist form of projection, is to simply allong lines parallel to the z-axis. This form of projection is called **orthographic** or **parallel**. It is the common form of projection used by draftspeople for top, bottom, and side views. The advantage of parallel projection is that the you can make accurate measurments of image features in the two dimensions that remain. The disadvantage is that the images don't appear natural (i.e. they lack perspective foreshorting).

Here is an example of an parallel projection of our scene.

Notice that the parallel lines of the tiled floor remain parallel after orthographic projection.

Orthographic Projection

The projection matrix for orthographic projection is very simple

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

There are some problems with this simple form, however. To begin with the units of the transformed points are still the same as the model. This is great for drafting, but in our case we'd like for the units to be in pixels.

Screen-Space Transformation

We can incorporate this change of units, and perform the flip of the y-axis required for raster coordinates into our projection matrix as follows.

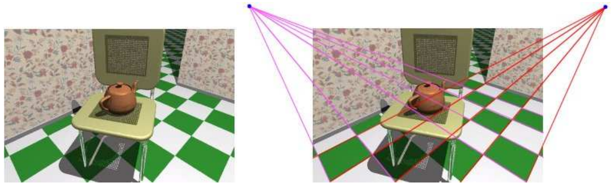
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{width}{right-left} & 0 & 0 & \frac{-left \times width}{right-left} \\ 0 & \frac{height}{bottom-top} & 0 & \frac{-top \times height}{bottom-top} \\ 0 & 0 & \frac{z_{max}}{far-near} & \frac{-near \times z_{max}}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This process is often called the viewport transformation. The variables, *left*, *right*, *top* and *bottom* refer to the extents of the viewing frustum in modeling units. The values *width* and *height* are in unit of pixels.

This transformation is little more than a scale and a translation.

Perspective Projection

Artists (Donatello, Brunelleschi, and Da Vinci) during the renaissance discovered the importance of perspective for making images appear realistic. This outdates mathematicians by more than 300 years. Perspective causes objects nearer to the viewer to appear larger than the same object would appear farther away. Another for introducing homogenous coordinates to computer graphics was to accomplish perspective projections using linear operators.



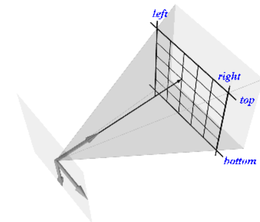
Note how lines known to be parallel in image space appear to converge to a single point when viewed in perspective. This is an important attribute of lines in projective spaces, they always intersect at a point.

Perspective Projection

The projection matrix for perspective projection matrix is:

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

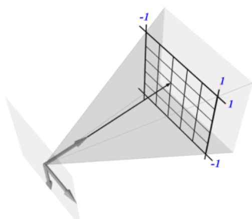
Notice how similar this transform is to the original parallel projection. Once more the units of the tranform are those of the model and not pixels.



Perspective Projection

The following transformation accomplishs the projection and the conversion to pixels in a single transform.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{width \times near}{right - left} & 0 & \frac{-left \times width}{right - left} & 0 \\ 0 & \frac{height \times near}{bottom - top} & \frac{-height \times top}{bottom - top} & 0 \\ 0 & 0 & \frac{z_{\max} \times far}{far - near} & \frac{-z_{\max} \times far \times near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



For the purpose of clipping we can modify this transformation so that it is mapped into a canonical space.

Canonical Perspective Projection

Following is a canonical space mapping for orthographic projections.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & \frac{-(right + left)}{right - left} \\ 0 & \frac{2}{bottom - top} & 0 & \frac{-(bottom + top)}{bottom - top} \\ 0 & 0 & \frac{2}{far - near} & \frac{-(far + near)}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Next is a canonical space mapping for perspective projections.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2 \times near}{right - left} & 0 & \frac{-(right + left)}{right - left} & 0 \\ 0 & \frac{2 \times near}{bottom - top} & \frac{-(bottom + top)}{bottom - top} & 0 \\ 0 & 0 & \frac{far + near}{far - near} & \frac{-2 \times far \times near}{far - near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Next Time