Rendering Pipeline, Projection, Viewing Transform

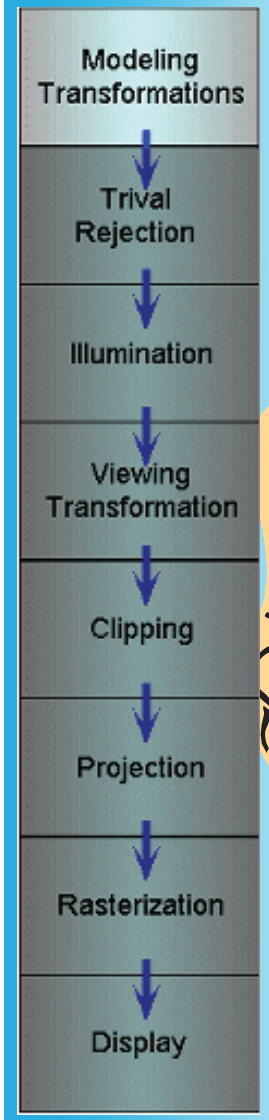
Lecture 9

CISC 440/640

Spring 2015

COMPUTER GRAPHICS

Modeling Transforms

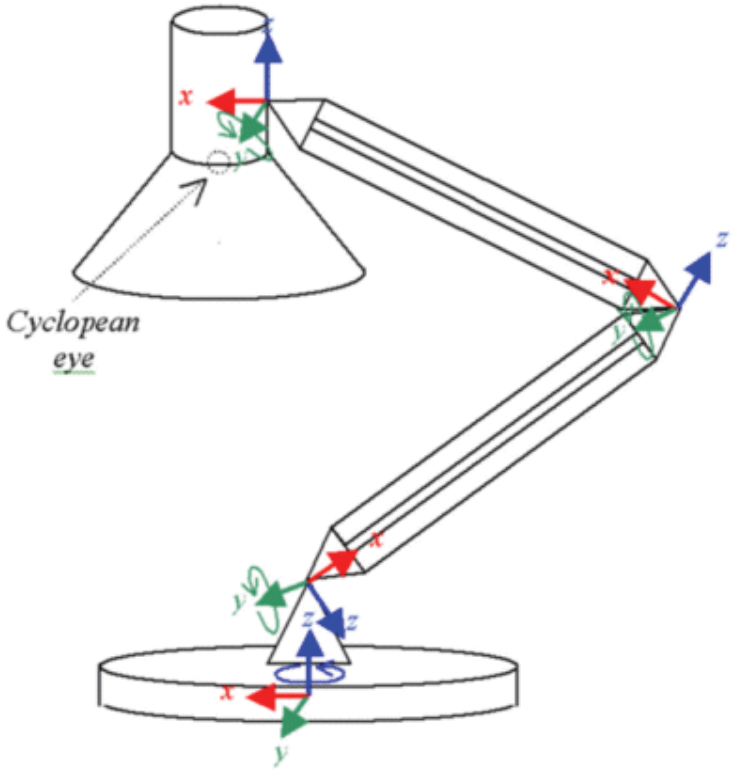


- Vast majority of transformations are modeling transforms
- Generally fall into one of two classes
- Transforms that relate a local model's frame to the scene's world frame
- Transforms that move parts within the model
- Generally, only Similitude and Euclidean transforms are needed

COMPUTER GRAPHICS

Transformation Hierarchies

- Many models are composed of independent moving parts
- Each part defined in it's own coordinate system
- Compose transforms to position and orient the model parts
- A Simple "One-chain" Example



Using Graphs to Model Hierarchies

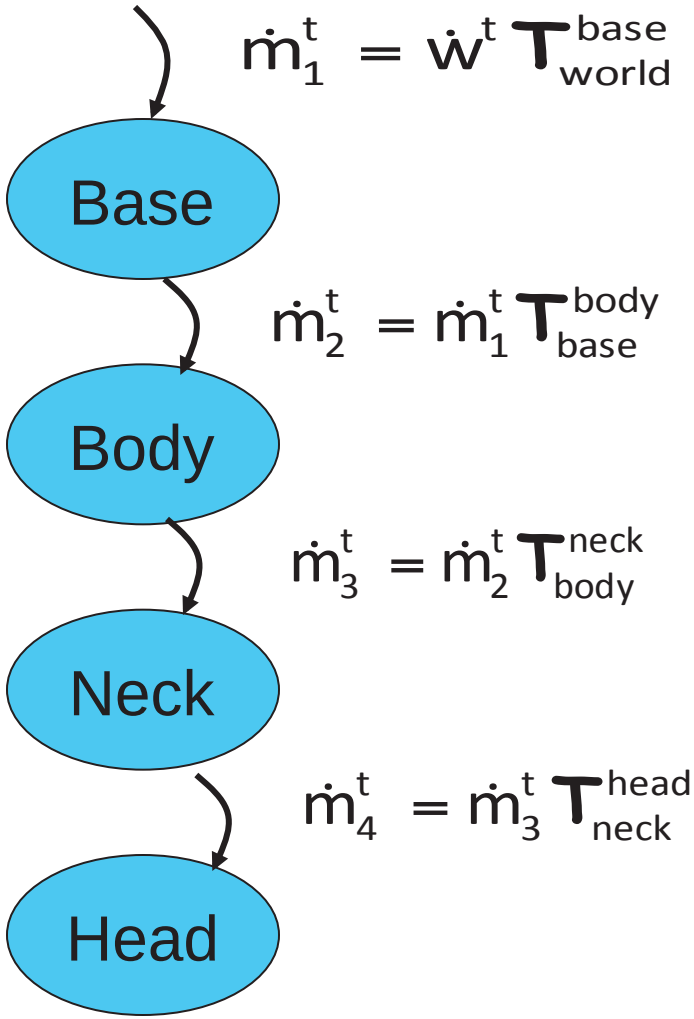
- Model parts are nodes
- Transforms are edges
- What transform is applied to the Head part to get it into world coordinates?

$$\dot{m}_4^t = \dot{w}^t \mathbf{T}_{world}^{base} \mathbf{T}_{base}^{body} \mathbf{T}_{body}^{neck} \mathbf{T}_{neck}^{head}$$

- Suppose that you'd like to rotate the Neck joint at the point where it meets the Body. Then what is the Head's transform to world space?

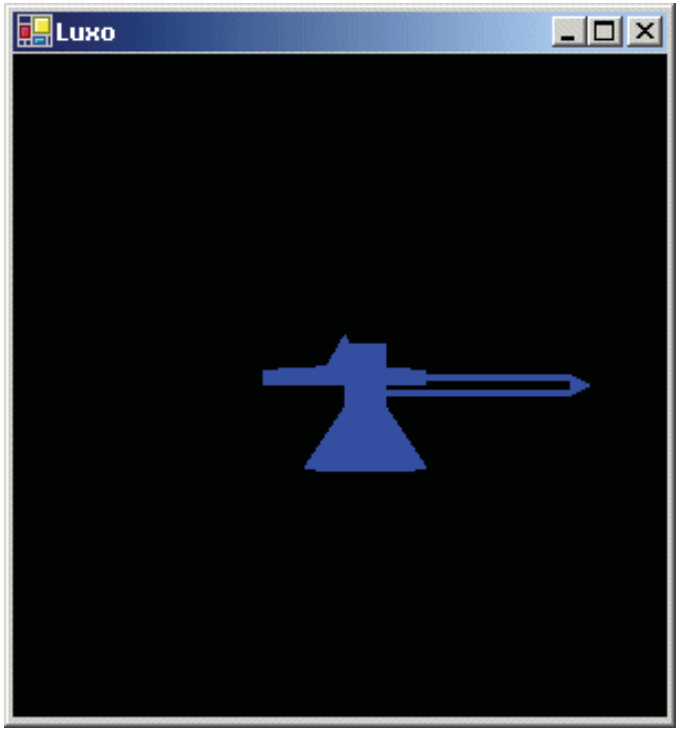
$$\dot{m}_3^t = \dot{m}_2^t \mathbf{T}_{body}^{neck} \mathbf{R}$$

$$\dot{m}_4^t = \dot{w}^t \mathbf{T}_{world}^{base} \mathbf{T}_{base}^{body} \mathbf{T}_{body}^{neck} \mathbf{R} \mathbf{T}_{neck}^{head}$$



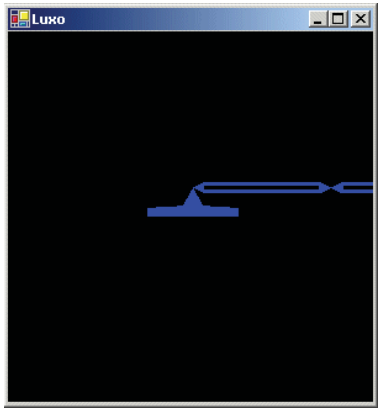
Code Example (1st try)

```
public override void Draw() {
      glClear(GL_COLOR_BUFFER_BITGL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      glTranslated(0.0, 0.0, -60.0);   // world-to-view transform
      glColor3d(0,0,1);
      glRotated(-90, 1, 0, 0);              // base-to-world transform
      mom.Draw(Lamp.BASE);
      mom.Draw(Lamp.BODY);
      mom.Draw(Lamp.NECK);
      mom.Draw(Lamp.HEAD);
      glFlush();
  }
```



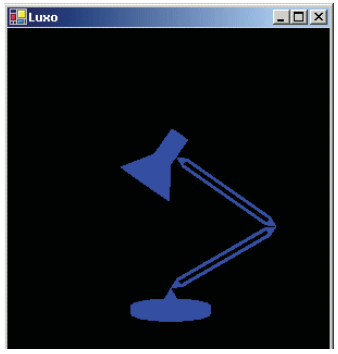
Code Example (2nd Try)

```
public override void Draw() {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslated(0.0, 0.0, -60.0);     // world-to-view transform
        glColor3d(0,0,1);
        glRotated(-90, 1, 0, 0);              // base-to-world transform
        mom.Draw(Lamp.BASE);
        glTranslated(0,0,2.5);                // body-to-base transform
        mom.Draw(Lamp.BODY);
        glTranslated(12,0,0);                  // neck-to-body transform
        mom.Draw(Lamp.NECK);
        glTranslated(12,0,0);                  // head-to-neck transform
        mom.Draw(Lamp.HEAD);
        glFlush();
    }
```



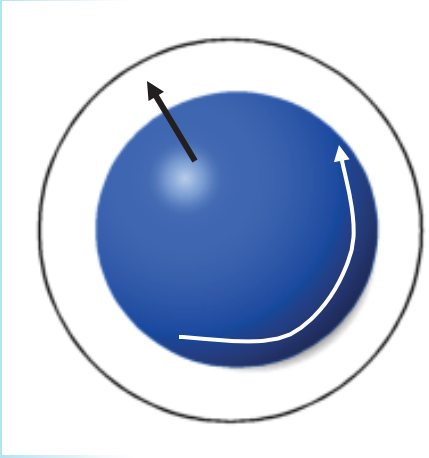
Code Example (3rd try)

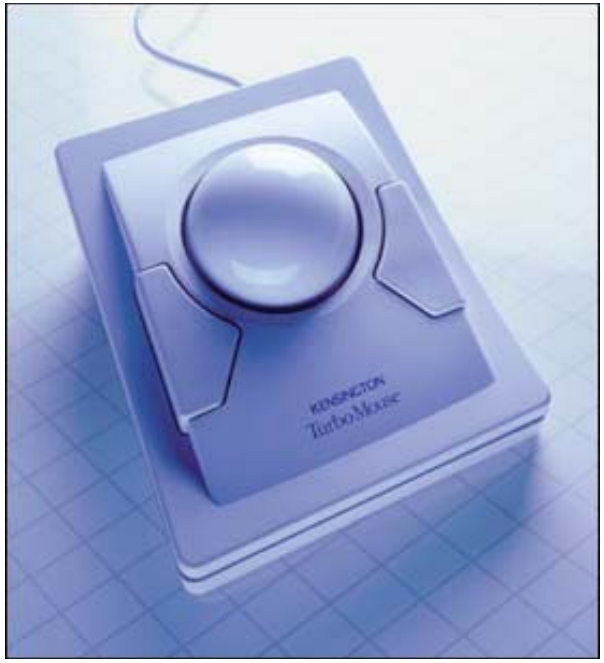
```
public override void Draw() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslated(0.0, -12.0, -60.0);   // world-to-view transform
    glColor3d(0,0,1);
    glRotated(-90, 1, 0, 0);           // base-to-world transform
    mom.Draw(Lamp.BASE);
    glTranslated(0,0,2.5);             // body-to-base transform
    glRotated(-30, 0, 1, 0);           // rotate body at base pivot
    mom.Draw(Lamp.BODY);
    glTranslated(12,0,0);              // neck-to-body transform
    glRotated(-115, 0, 1, 0);          // rotate neck at body pivot
    mom.Draw(Lamp.NECK);
    glTranslated(12,0,0);              // head-to-neck transform
    glRotated(180, 1, 0, 0);           // rotate head at neck pivot
    mom.Draw(Lamp.HEAD);
    glFlush();
}
```



Exercise: A Trackball Interface

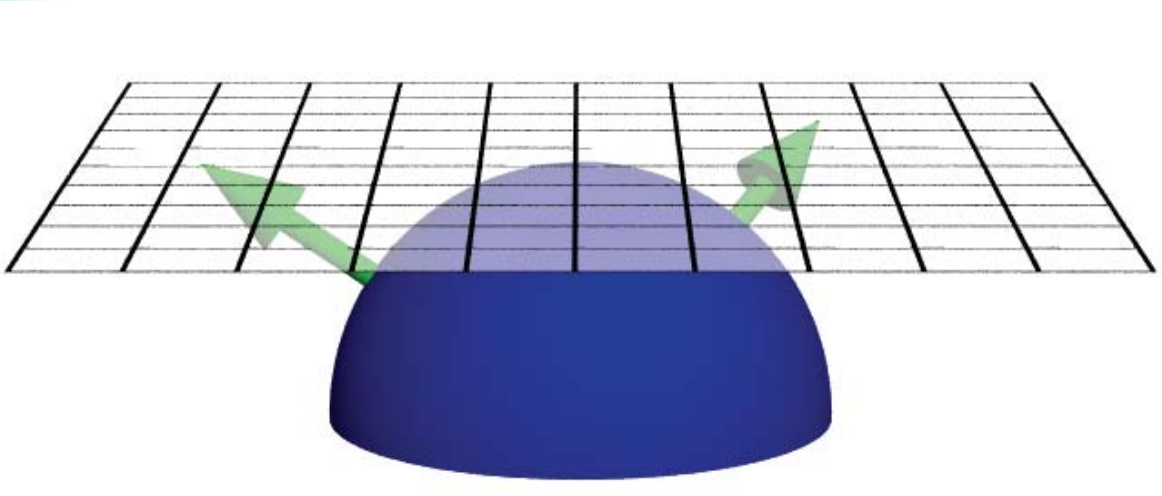
- A common UI for manipulating objects
- Virtual trackball
- 2 degree of freedom device
- However, its differential behavior provides a intuitive rotation specification





A Virtual Trackball

- Imagine the viewport as floating above, and just touching an actual trackball.
- You receive the coordinates in screen space of the MouseDown() and MouseMove() events.
- What is the axis of rotation?
- What is the angle of rotation?



Some help with virtual trackball

- http://www.cse.ohio-state.edu/~crawfis/cis781/Slides/VirtualTrackball.html
- Map mouse to the sphere

- Treat the mouse position as the projection of a point on the hemi-sphere down to the image plane (along the z-axis), and determine that point on the hemi-sphere.

```
//
// Utility routine to calculate the 3D position of a
// projected unit vector onto the xy-plane. Given any
// point on the xy-plane, we can think of it as the projection
// from a sphere down onto the plane. The inverse is what we
// are after.
//
Vec3f CSierpinskiSolidsView::trackBallMapping(CPoint point)
{

        Vec3f v;
        float d;
        v.x = (2.0*point.x - windowSize.x) / windowSize.x;
        v.y = (windowSize.y - 2.0*point.y) / windowSize.y;
        v.z = 0.0;
        d = v.Length();
        d = (d<1.0) ? d : 1.0;
        v.z = sqrtf(1.001 - d*d);
        v.Normalize(); // Still need to normalize, since we only capped d, not v.
        return v;

}
```

Determine Rotation Axis and Angle

- Detect the mouse movement

```
void CSierpinskiSolidsView::OnMouseMove(UINT nFlags, CPoint point)
{

        //
        // Handle any necessary mouse movements
        //
        Vec3f direction;
        float pixel_diff;
        float rot_angle, zoom_factor;
        Vec3f curPoint;
        switch (Movement)
        {

        case ROTATE : // Left-mouse button is being held down
        {
        curPoint = trackBallMapping( point ); // Map the mouse position to a logical
        // sphere location.
        direction = curPoint - lastPoint;
        float velocity = direction.Length();
        if( velocity > 0.0001 ) // If little movement - do nothing.
        {
```

- Determine the great circle connecting the old mouse-hemi-sphere point to the current mouse-hemi-sphere point.
- Calculate the normal to this plane. This will be the axis about which to rotate.

```
        //
        // Rotate about the axis that is perpendicular to the great circle connecting the
        mouse movements.
        //
        Vec3f rotAxis;
        rotAxis.crossProd( lastPoint, curPoint );
        rot_angle = velocity * m_ROTSCALE;
```

Apply GL Rotation

- Very important: the order!

- Read off the current matrix, since we want this operation to be the last transformation, not the first, and OpenGL does things LIFO.
- Reset the model-view matrix to the identity
- Rotate about the axis
- Multiply the resulting matrix by the saved matrix.

```
//
// We need to apply the rotation as the last transformation.
// 1. Get the current matrix and save it.
// 2. Set the matrix to the identity matrix (clear it).
// 3. Apply the trackball rotation.
// 4. Pre-multiply it by the saved matrix.
//
glGetFloatv( GL_MODELVIEW_MATRIX, (GLfloat *) objectXform
);
glLoadIdentity();
glRotatef( rot_angle, rotAxis.x, rotAxis.y, rotAxis.z );
glMultMatrixf( (GLfloat *) objectXform );
```

Projections

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 reduces the range of projection matrices.

A projection maps all 3-D coordinates onto a desired viewing plane. Thus, making our 3-D world into a 2-D image. This sort of mapping is not affine like all of the transforms we've discussed thus far. In fact, projection matrices do not transform points from our affine space back into the same space.

They transform points into something different. Usually, we will use a projection matrix to reduce the dimensionally of our affine points. Thus, we should expect projection matrices to be *less than full rank*.

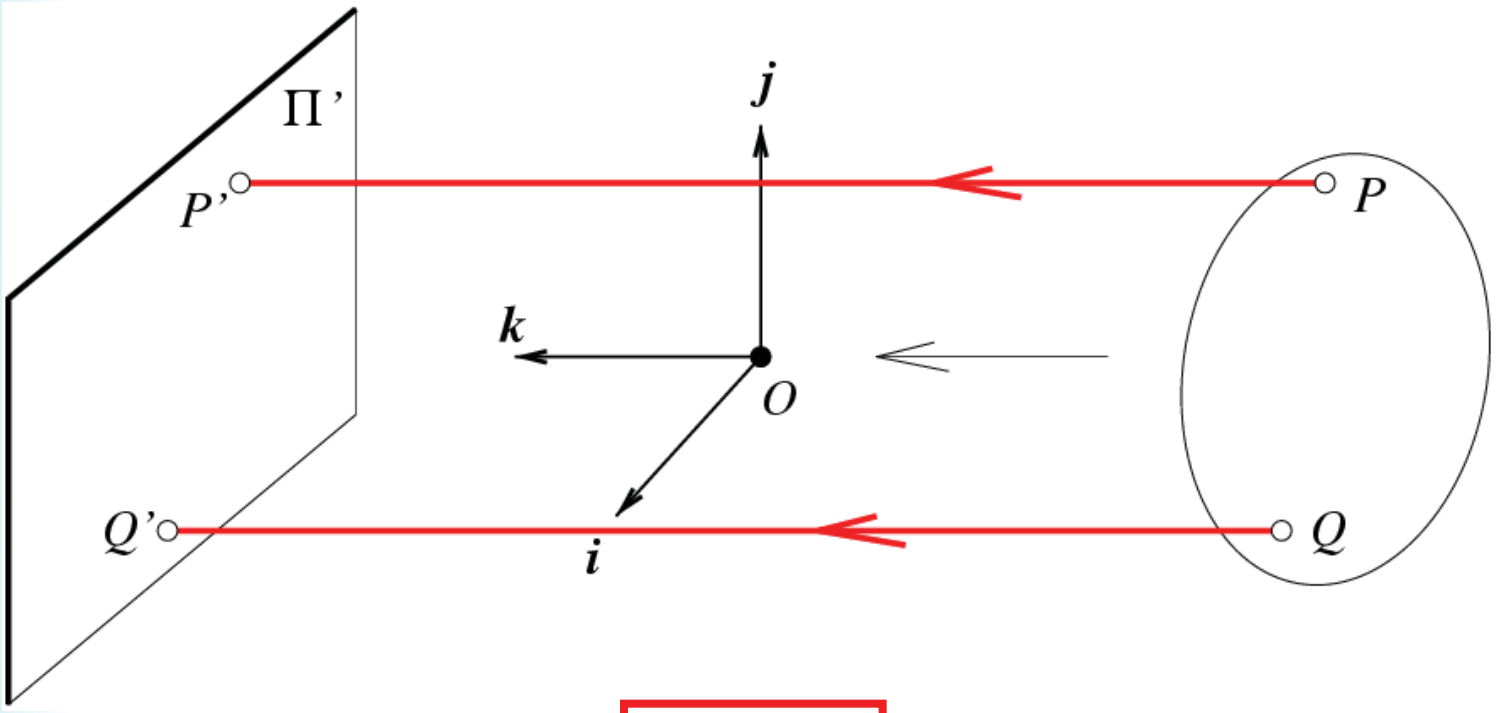
Orthographic Projection

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

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



$$\begin{cases} x' = x \\ y' = y \\ z' = 0 \end{cases}$$

Orthographic Projection

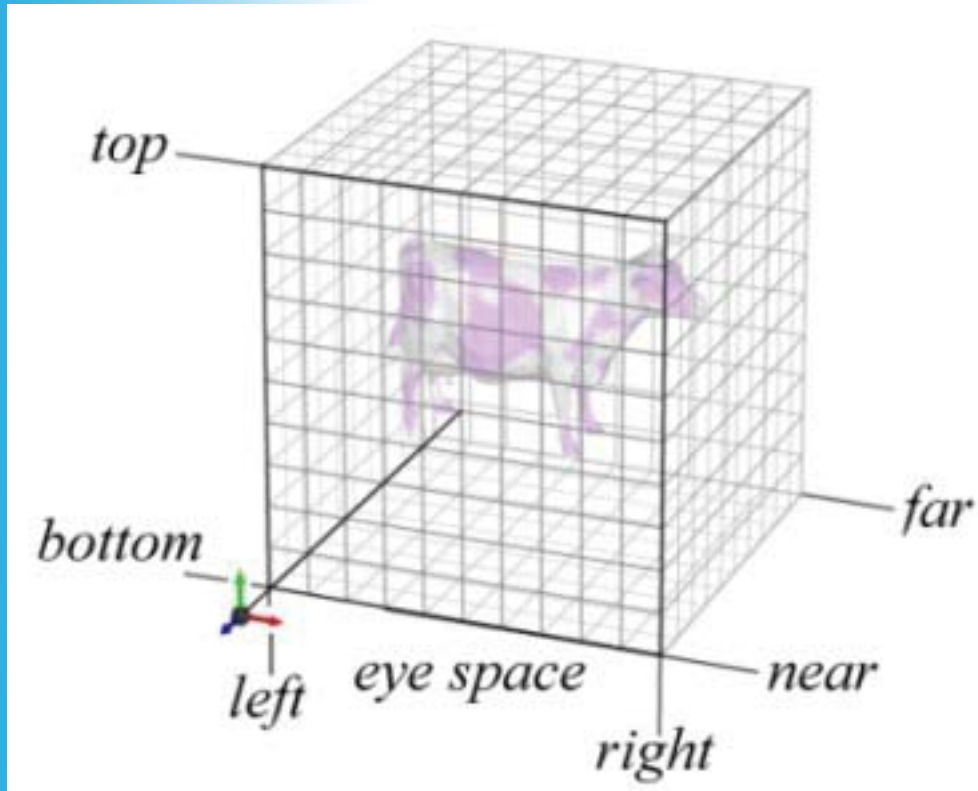
The projection matrix for orthographic projection is very simple

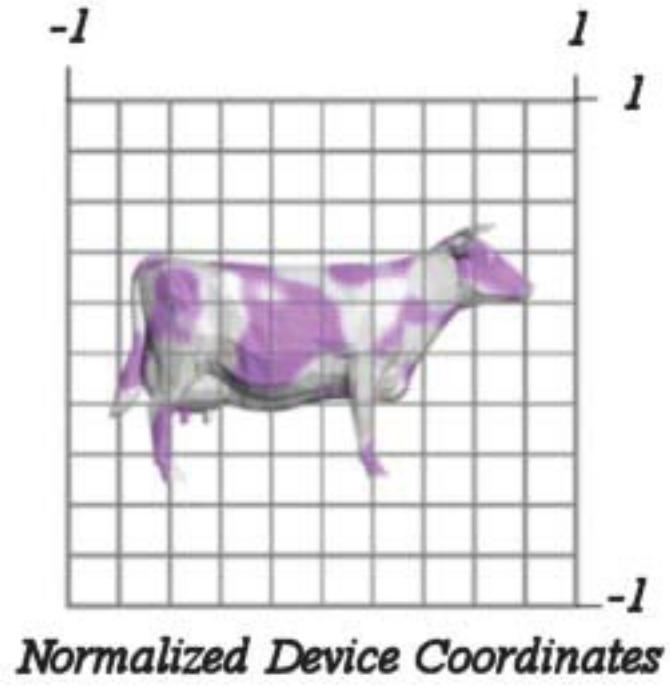
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

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 to units that are model-independent. This will allow us to perform a wide range of operations using normalized coordinates.

Normalized Device Coordinates

Therefore we will compose our projection with a set of scale and a translation that maps our coordinates in world units to a normalized coordinates.





Orthographic Projections to NDC

Here is the mapping:

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

We also scale the z coordinate in exactly the same way (i.e. all z values between near and far are mapped from –1 to 1 respectively).

Technically, this coordinate is not part of the projection. But, we will use this value of z for other purposes.

Some sanity checks:

$$x = \text{left} \Rightarrow x' = \frac{2 \cdot \text{left}}{\text{right}-\text{left}} - \frac{\text{right}+\text{left}}{\text{right}-\text{left}} = -\frac{\text{right}-\text{left}}{\text{right}-\text{left}} = -1$$

$$x = \text{right} \Rightarrow x' = \frac{2 \cdot \text{right}}{\text{right}-\text{left}} - \frac{\text{right}+\text{left}}{\text{right}-\text{left}} = \frac{\text{right}-\text{left}}{\text{right}-\text{left}} = 1$$

Orthographic Projection in OpenGL

This matrix is constructed by the following OpenGL call:

void **glOrtho**(double *left*, double *right*,
double *bottom*, double *top*,
double *near*, double *far*);

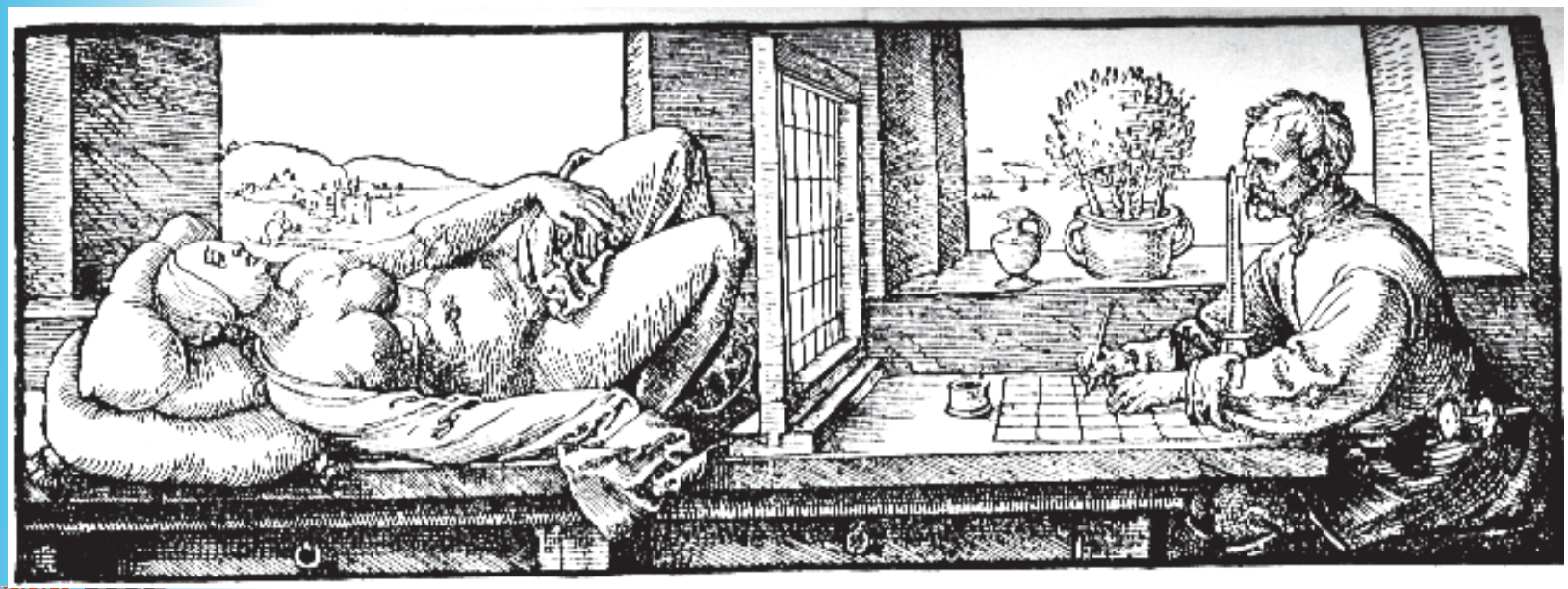
And the 2-D version (another GL utility function):

void **gluOrtho2D**(double *left*, GLdouble *right*,
double *bottom*, GLdouble *top*);

Which is just a call to glOrtho( ) with near = -1 and far = 1;

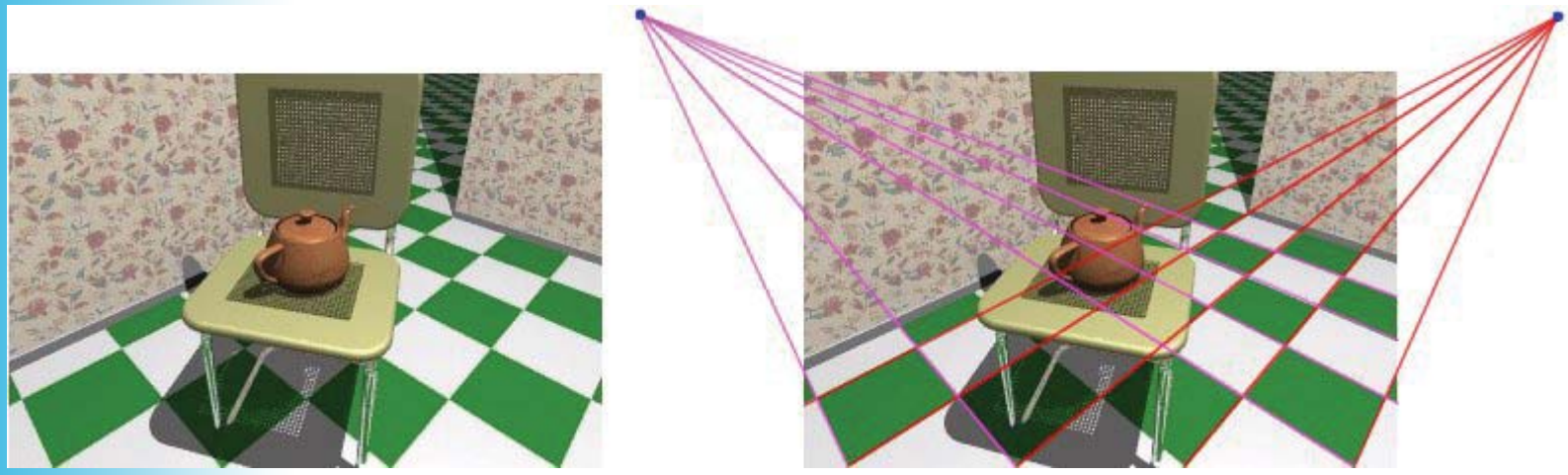
Perspective Projection

Artists (Donatello, Brunelleschi, Durer, 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.



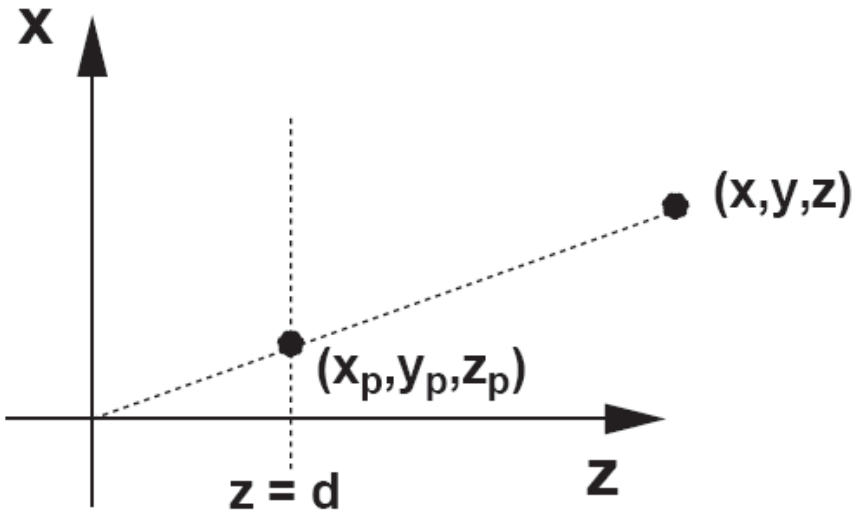
Signs of Perspective

Notice 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 1

- Assume project through origin onto plane $z = d$



- What are the coordinates of the projected point x_p, y_p, z_p?

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d} \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d} \qquad z_p = d$$

Perspective Projection

The simplest transform for perspective projection is:

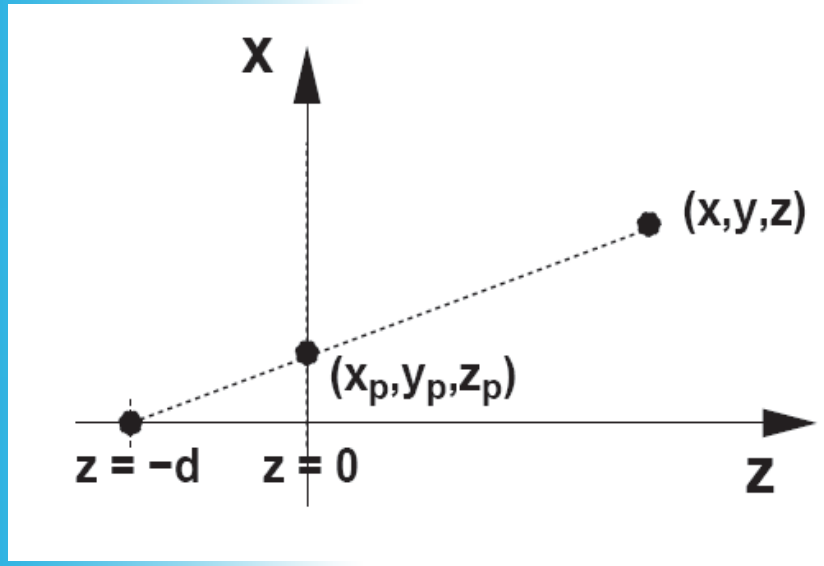
$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Here we assume the COP is at the origin and image plane at z = d;

We then apply our rules for a projective spaces, to find our preferred point (the one with a fourth component of 1) by dividing each element of the vector by w. In this example projection matrix, w is simply the z component.

Another Perspective Projection

- Projection Plane at z = 0
- COP at z = -d



- What happens when d goes to infinity

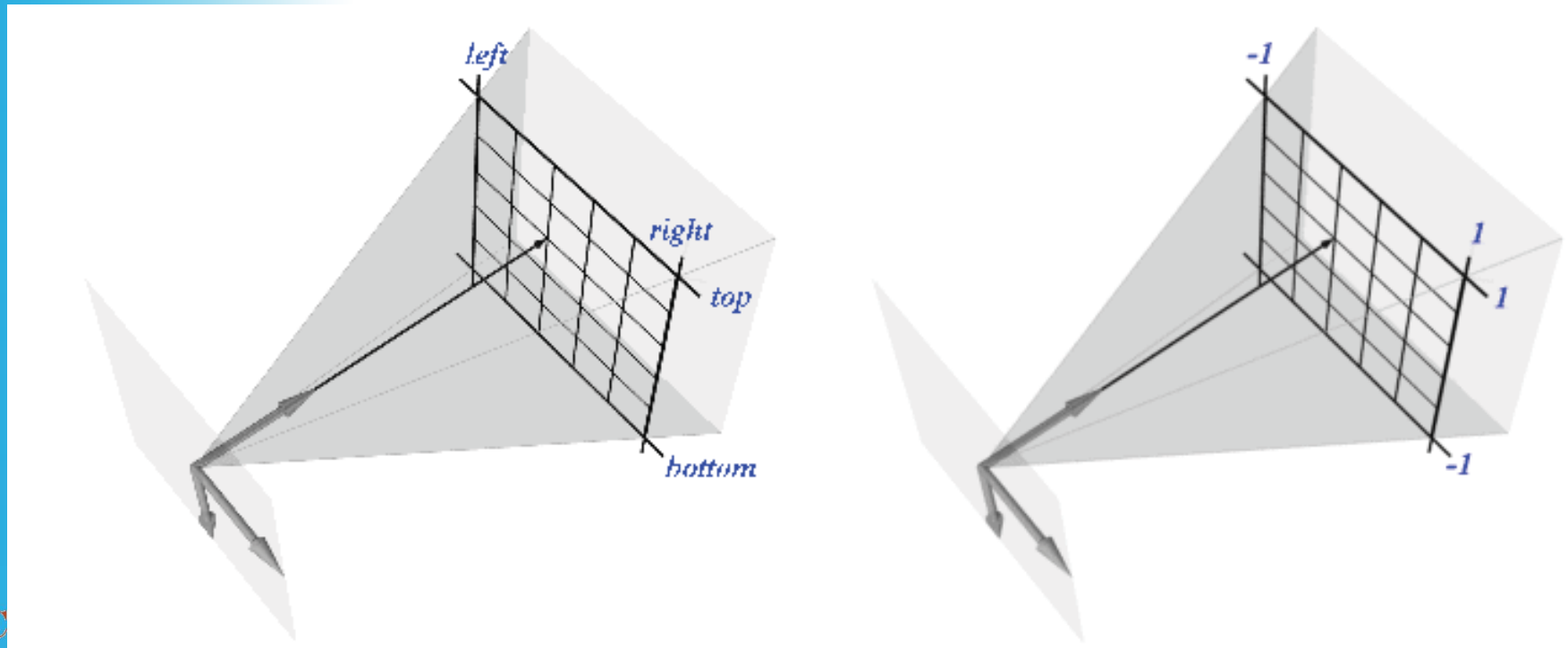
$$x_p = \frac{d \cdot x}{z + d} = \frac{x}{(z/d) + 1}$$

$$y_p = \frac{d \cdot y}{z + d} = \frac{y}{(z/d) + 1}$$

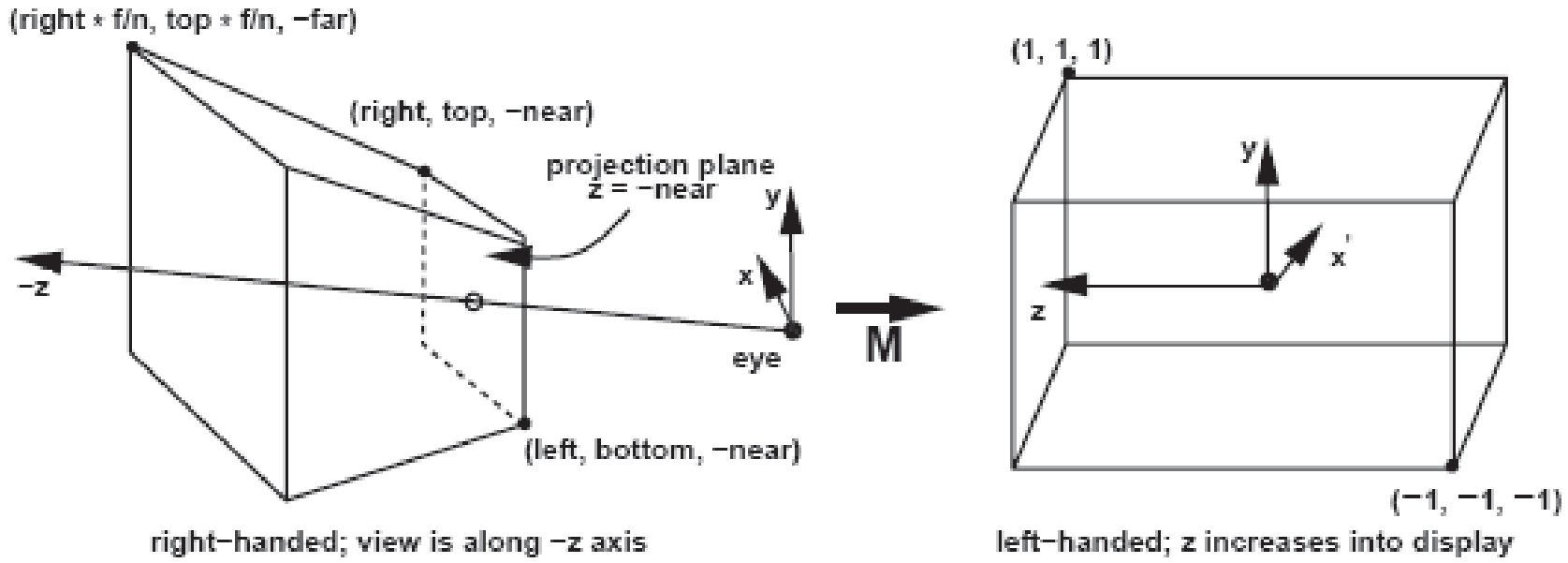
$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Normalized Perspective

As in the orthographic case, perspective projection preserves the units of world-space. Once again, to simplify later operations we would like to specify a perspective projection where some specific range of world-space coordinates are mapped to a Normalized coordinate system.



Viewing Frustum



right−handed; view is along −z axis

left−handed; z increases into display

NDC Perspective Matrix

This can be accomplished with a clever composition of transforms with our projection matrix.

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

The values of left, right, top, and bottom are specified at the near depth.

Let's try some sanity checks:

$$\left.\begin{array}{l} x = left \\ z = near \end{array}\right\} \Rightarrow x' = \frac{\frac{2\cdot near\cdot left}{right-left} - \frac{near(right+left)}{right-left}}{near} = \frac{-near}{near} = -1$$

$$\left.\begin{array}{l} x = right \\ z = near \end{array}\right\} \Rightarrow x' = \frac{\frac{2\cdot near\cdot right}{right-left} - \frac{near(right+left)}{right-left}}{near} = \frac{near}{near} = 1$$

Perspective in OpenGL

OpenGL provides the following function to define perspective transformations:

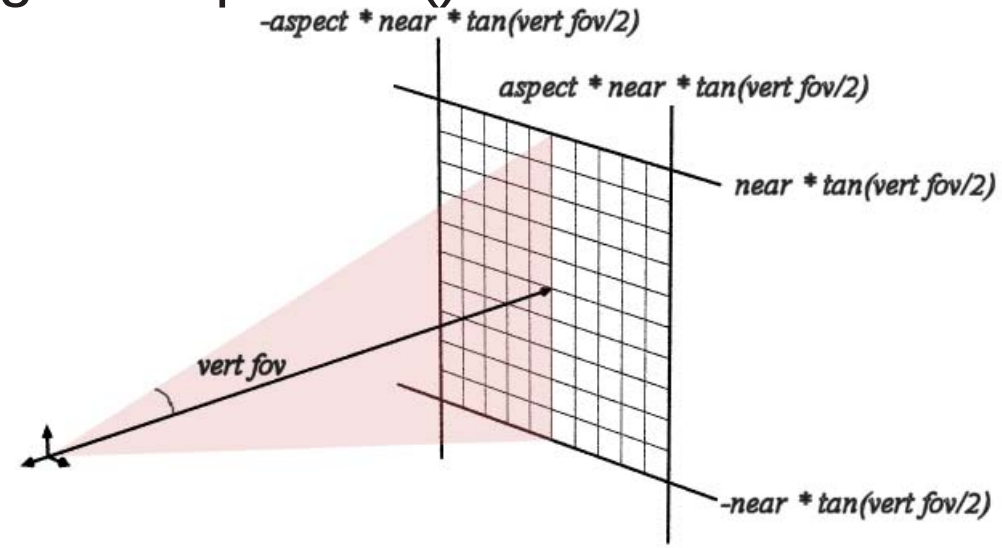
void **glFrustum**(double *left*, double *right*,
double *bottom*, double *top*,
double *near*, double *far*);

Some think that using glFrustum( ) is nonintuitive. So OpenGL provides a utility function with simpler, but less general capabilities.

void **gluPerspective**(double *vertfov*, double *aspect*,
double *near*, double *far*);

gluPerspective()

The following figure illustrates the parameters of gluPerspective():



Simple "camera-like" model

Can only specify symmetric frustums.

Substituting the extents into glFrustum()

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} \frac{COT(\frac{vertfov}{2})}{aspect} & 0 & 0 & 0 \\ 0 & COT(\frac{vertfov}{2}) & 0 & 0 \\ 0 & 0 & \frac{far+near}{far-near} & \frac{-2 \cdot far \cdot near}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

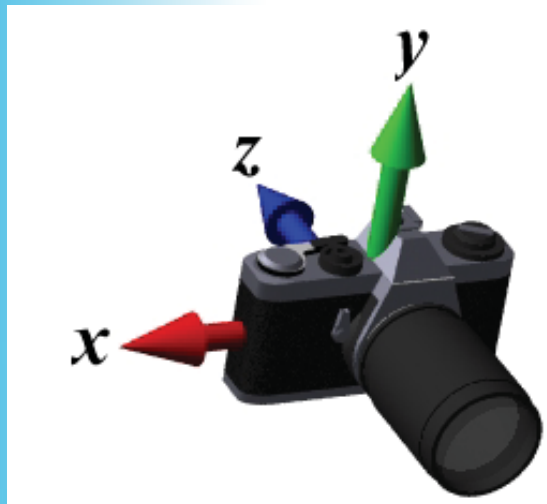
Frame verses Coordinate

In order to render the scene it is necessary 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**.

However, it is more “natural” to think of the camera as if it were an object positioned in the world frame. So we will define the mapping between frames using the more intuitive model (considering the camera as a model).

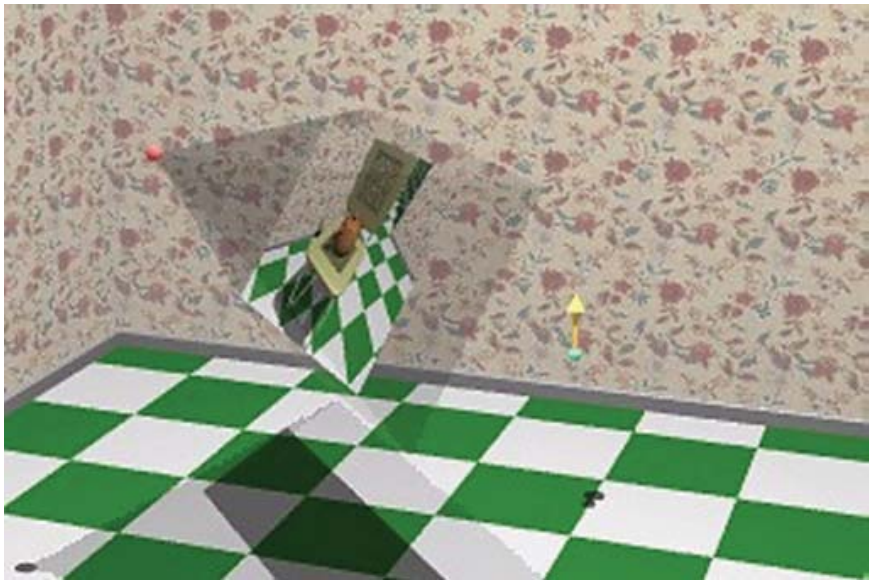
Viewing Steps

First, we 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 do we rotate first and then translate?)



An Intuitive Specification

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 the Viewing Transform

We 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:

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

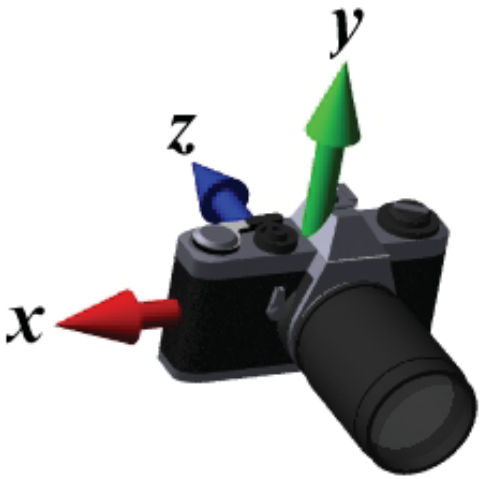
After normalizing it

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

First Constraint

The resulting unit vector, $\hat{l}$, is often called the "*look-at*" vector.

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



$$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} = \mathbf{R}_v \begin{bmatrix} \hat{l}_x \\ \hat{l}_y \\ \hat{l}_z \end{bmatrix}$$

Second Constraint

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.

$$\vec{r} = \vec{l} \times \overrightarrow{up}$$

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

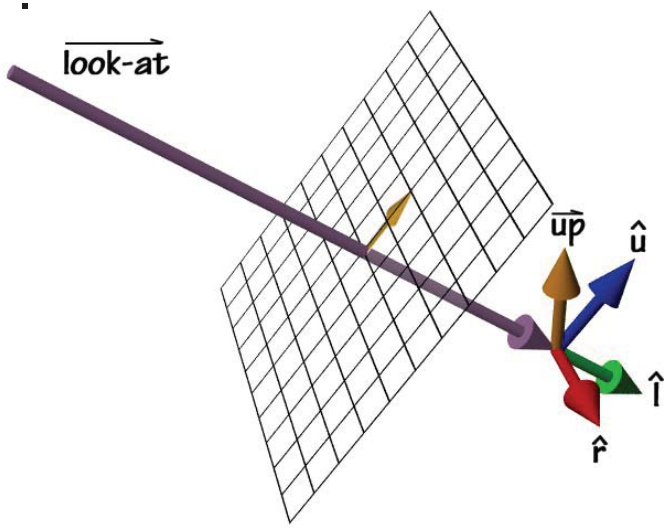
$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{R}_v \frac{\vec{r}}{\sqrt{r_x^2 + r_y^2 + r_z^2}}$$

Third Constraint

Finally, from these two vectors we can synthesize a third vector that is perpendicular to both the look-at and right vectors. It is also oriented in the up direction.

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

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



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

Now we've specified everything that the viewing matrix must do. We need only construct it.

Putting it all Together

Now lets consider all of these constraints together.

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

In order to compute the matrix, $\mathbf{R_v}$, 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.

When Transpose is Inverse

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:

$$\text{if } \mathbf{M} \text{ is Orthonormal} \quad \mathbf{M}^{-1} = \mathbf{M}^{T}$$

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

$$\mathbf{R}_v = \begin{bmatrix} \hat{r} \\ \hat{u} \\ -\hat{l} \end{bmatrix}$$

Now for the Translation

The rotation that we just derived is specified about the *eye point* in *world space*.

Therefore, before we can apply this rotation, we need to translate all world-space coordinates so that the eye point is at the origin.

$$\dot{e}^t = \dot{w}^t \mathbf{R}_v \mathbf{T}_{-eye}$$

Composing these transformations gives our viewing transform, V.

$$\mathbf{V} = \mathbf{R}_v \mathbf{T}_{-\text{eye}} = \begin{bmatrix} \hat{r}_x & \hat{r}_y & \hat{r}_z & 0 \\ \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ -\hat{l}_x & -\hat{l}_y & -\hat{l}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\text{eye}_x \\ 0 & 1 & 0 & -\text{eye}_y \\ 0 & 0 & 1 & -\text{eye}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r} \cdot \overrightarrow{\text{eye}} \\ & \hat{u} & & -\hat{u} \cdot \overrightarrow{\text{eye}} \\ & -\hat{l} & & \hat{l} \cdot \overrightarrow{\text{eye}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing transforms in OpenGL

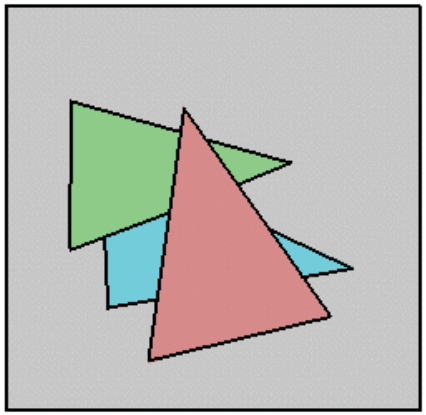
OpenGL provides a function for computing viewing transformations specified in terms of world space coordinates in its utility library:

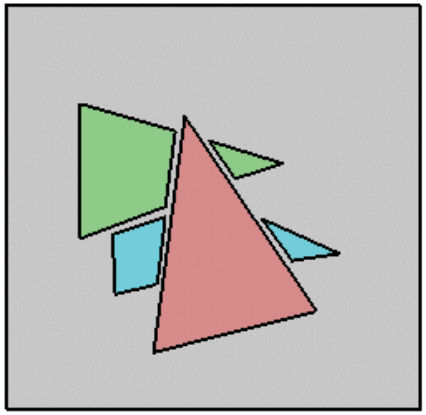
```
gluLookAt(double eyex, double eyey, double eyez,
          double centerx, double centery, double centerz,
          double upx, double upy, double upz);
```

It computes the same transformation that we derived and composes it with the current matrix.

Handling Occlusions

- For most interesting scenes and viewpoints, some polygons will overlap; somehow, we must determine which portion of each polygon is visible to eye.





- Solving occlusion problems used to be one of the

BIG PROBLEMS in Computer Graphics

A Painter's Algorithm

•The painter's algorithm, sometimes called depth-sorting, gets its name from the process which an artist renders a scene using oil paints. First, the artist will paint the background colors of the sky and ground. Next, the most distant objects are painted, then the nearer objects, and so forth. Note that oil paints are basically opaque, thus each sequential layer completely obscures the layer that its covers.

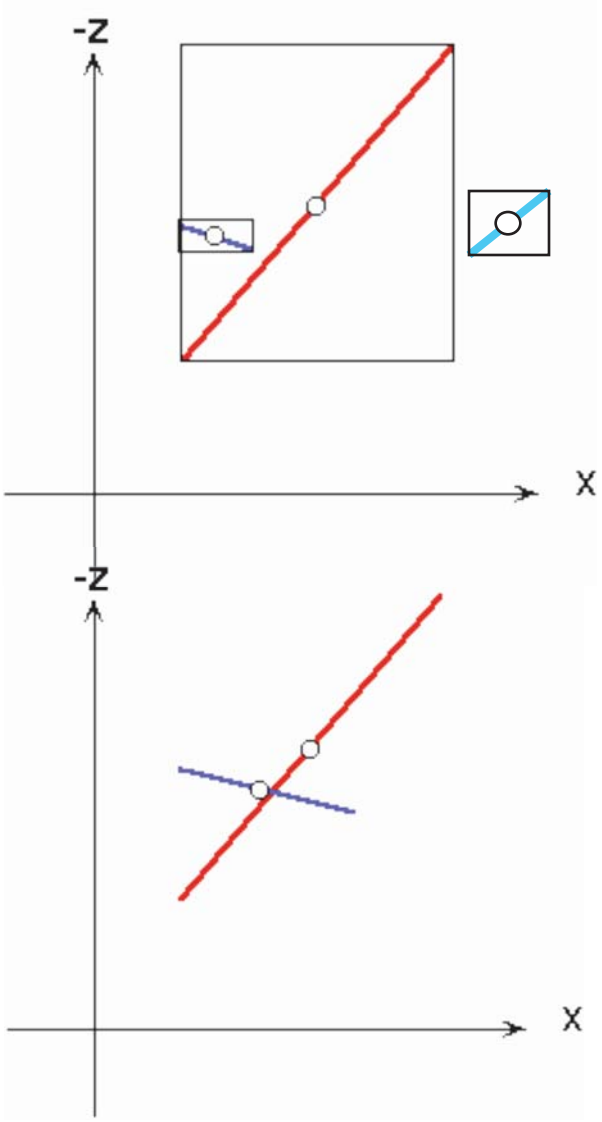
A very similar technique can be used for rendering objects in a three-dimensional scene. First, the list of surfaces are sorted according to their distance from the viewpoint. The objects are then painted from back-to-front.

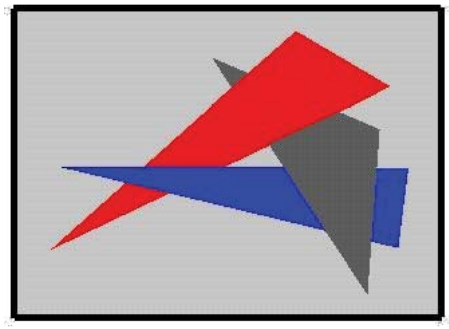
While this algorithm seems simple there are many subtleties. The first issue is which depth-value do you sort by? In general a primitive is not entirely at a single depth. Therefore, we must choose some point on the primitive to sort by.

Problems with Painters

•The painter's algorithm works great... unless one of the following happens:

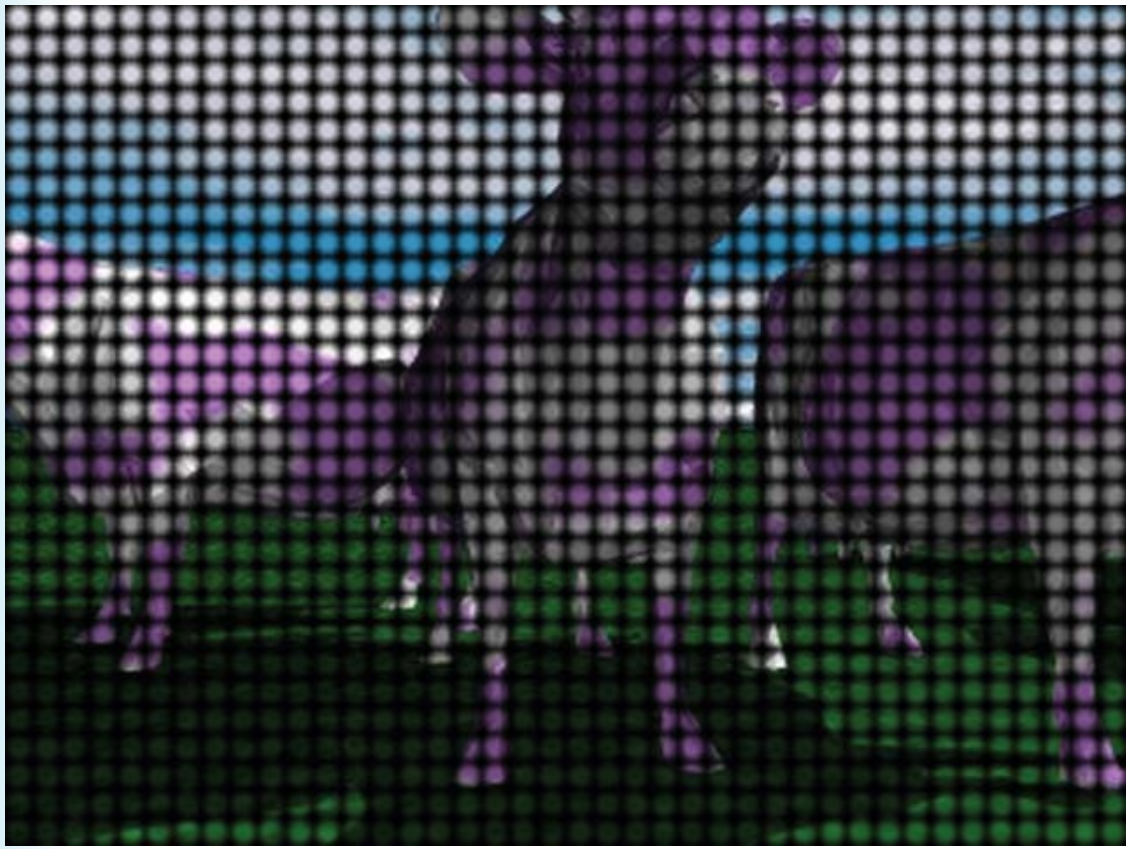
- Big triangles and little triangles. This problem can usually be resolved using further tests. Suggest some.
- Another problem occurs when the triangle from a model interpenetrate as shown below. This problem is a lot more difficult to handle. generally it requires that primitive be subdivided (which requires clipping).
- Cycles among primitives





Pixel-Level Visibility

Thus far, we've considered visibility at the level of primitives. Now we will turn our attention to a class of algorithms that consider visibility at each pixel.

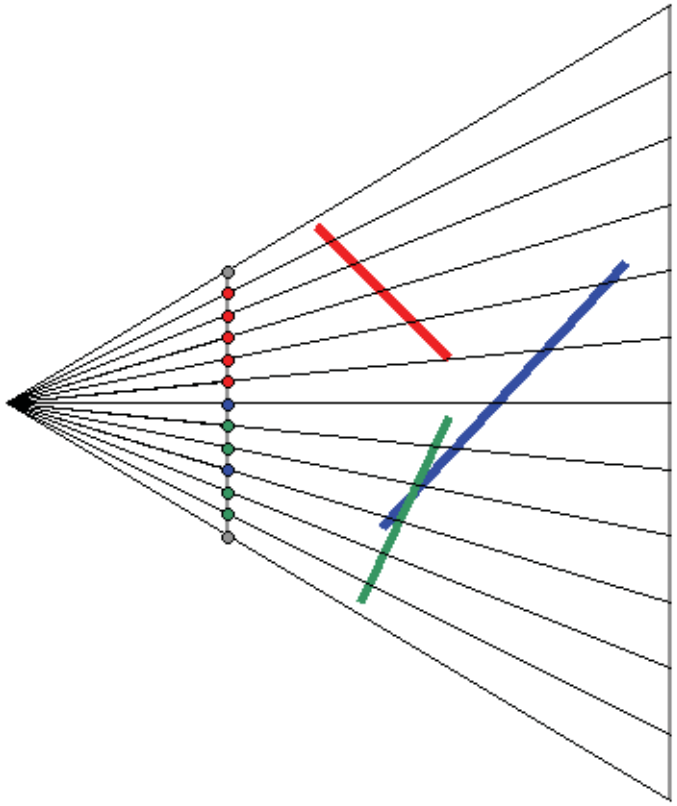


Ray Casting

Algorithm: Cast a ray from the viewpoint through each pixel to find the closest surface

Rendering Loop:

```
foreach pixel in image
    compute ray for pixel
    set depth = Z_MAX
    foreach primitive in scene
        if (ray intersects primitive) then
            if (distance < depth) then
                pixel = object color
                depth = distance to object
            endif
        endif
    endfor
endfor
```



Ray Casting

- Advantages
  - Conceptually simple
  - Can support CSG
  - Can take advantage of spatial coherence in scene
  - Can be extended to handle global illumination effects (ex: shadows and reflectance)
- Disadvantages
  - Renderer must have access to entire retained model
  - Hard to map to special-purpose hardware
  - Visibility determination is coupled to sampling
    - Subject to aliasing
    - Visibility computation is a function of resolution

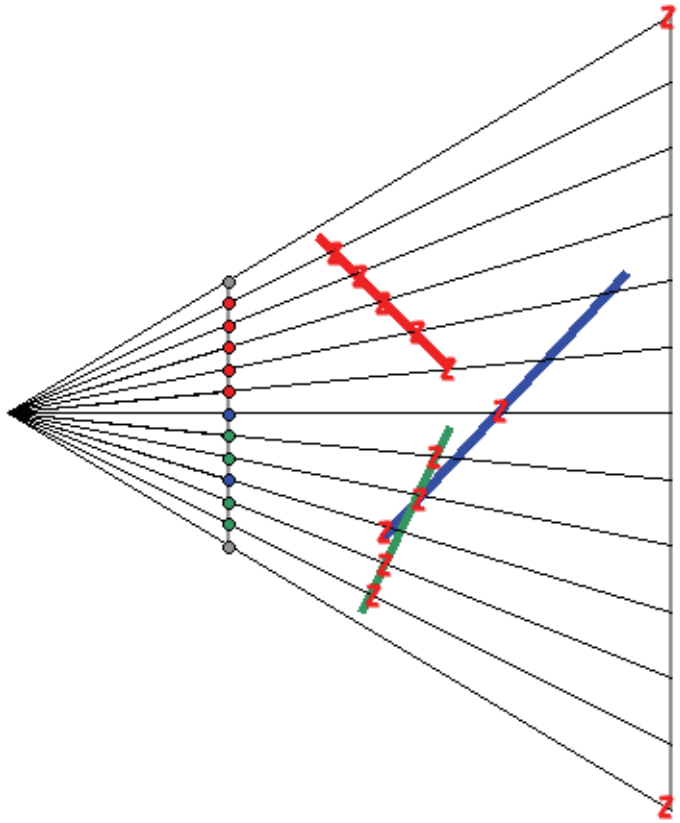
Depth Buffering

Algorithm:

Cast a ray from the viewpoint through each pixel to find the closest surface

Rendering Loop:

```
set depth of all pixels to Z_MAX
foreach primitive in scene
    determine pixels touched
    foreach pixel in primitive
        compute z at pixel
        if (z < depth) then
            pixel = object color
            depth = z
        endif
    endfor
endfor
```



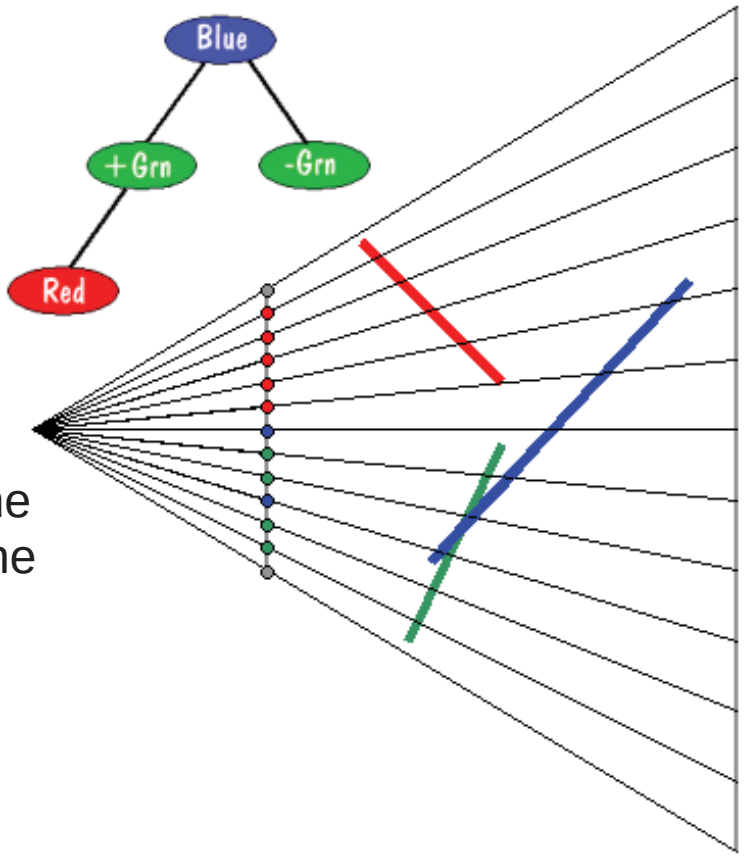
Depth Buffering

- Advantages
  - Primitives can be processed immediately (Hence: Immediate mode graphics API's)
  - Primitives can be processed in any order (Exception: primitives at same depth)
  - Well suited to H/W implementation simple control of low-level (per pixel) operations
  - Spatial coherence
    - Incremental evaluation of loops
    - Good memory access pattern
- Disadvantages
  - Visibility determination is coupled to sampling (Subject to aliasing)
  - Requires a Raster-sized arrray to store depth
  - Read-Modify-Write (Hard to make fast)
  - Excessive over-drawing

I think that I shall never see…

A Binary Space Partition (BSP) tree is a simple spatial data structure:

1. Select a partitioning plane/face.
2. Partition the remaining planes/faces according to the side of the partitioning plane that they fall on (+ or -).
3. Repeat with each of the two new sets.

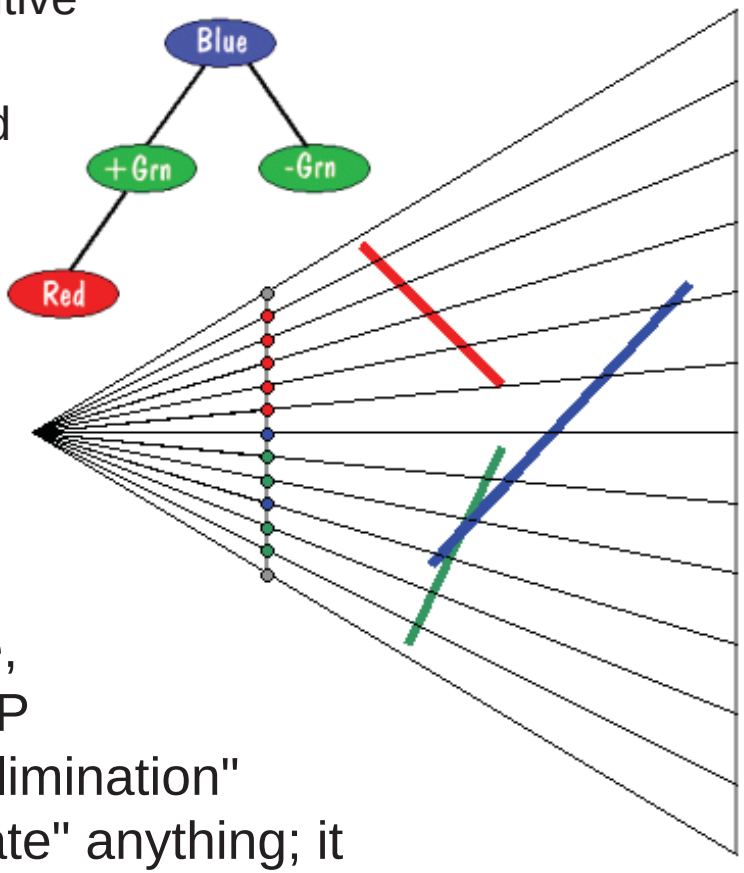


Partitioning facets:
Partitioning requires testing all facets in the active set to find if they 1) lie entirely on the positive side of the partition plane, 2) lie entirely on the negative side, or 3) if they cross it. In the 3rd case of a crossing face we clip the offending face into two halves (using our plane-at-a-time clipping algorithm).

Computing Visibility with BSP trees

Starting at the root of the tree.

1. Classify viewpoint as being in the positive or negative halfspace of our plane
2. Call this routine with the negative child (if it exists)
3. Draw the current partitioning plane
4. Call this routine with the positive child (if it exists)



Intuitively, at each partition, we first draw the stuff further away than the current plane, then we draw the current plane, and then we draw the closer stuff. BSP traversal is called a "hidden surface elimination" algorithm, but it doesn't really "eliminate" anything; it simply orders the drawing of primitive in a back-to-front order.

BSP Tree Example

Computing visibility or depth-sorting with BSP trees is both simple and fast.

It resolves visibility at the primitive level.

Visibility computation is independent of screen size

Requires considerable preprocessing of the scene primitives

Primitives must be easy to subdivide along planes

Supports Constructive Solid Geometry (CSG) modeling