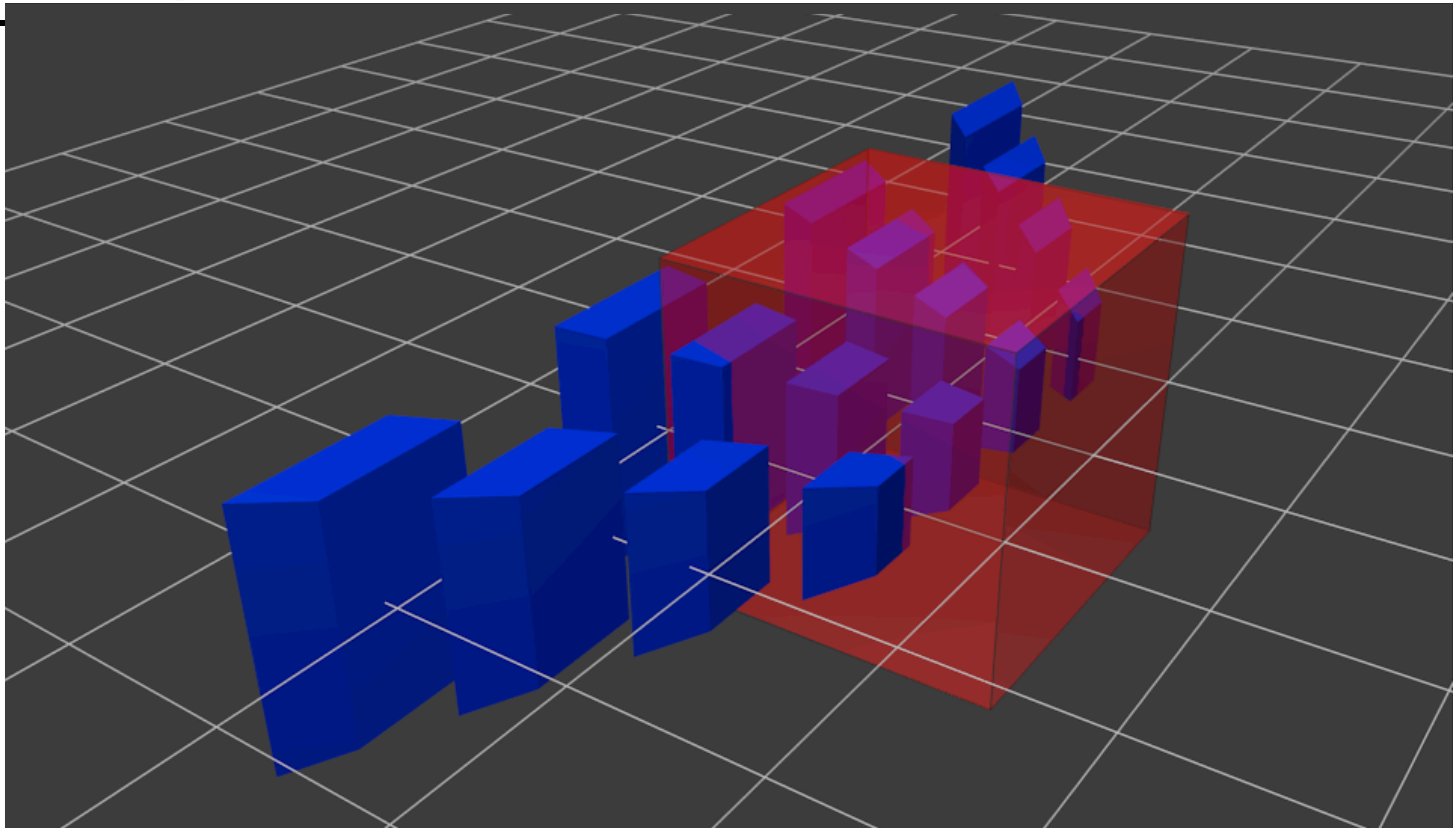
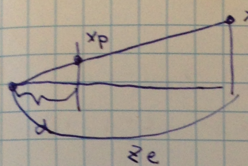
CSE 5542 - Real Time Rendering
Week 6

OpenGL Perspective Matrix

Courtesy: Prof. H-W. Shen

Perspective Transform





(x_e, y_e, z_e)

$$\frac{x_e}{x_p} = \frac{-z_e}{d} \Rightarrow x_p = \frac{x_e}{\frac{-z_e}{d}}$$

Same for y

$$\frac{y_e}{y_p} = \frac{-z_e}{d} \Rightarrow y_p = \frac{y_e}{\frac{-z_e}{d}}$$

Basic perspective projection matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{po} \quad \text{why?}$$

Because

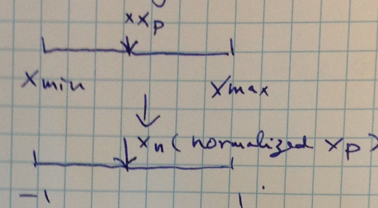
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_e \\ y_e \\ z_e \\ \frac{z_e}{-d} \end{bmatrix}$$

Normalize: $x_p = \frac{x_e}{\frac{z_e}{-d}}$ $\quad y_p = \frac{y_e}{\frac{z_e}{-d}}$ $\quad z_e = -d.$

But we need normalize $[x_{min}, x_{max}] \rightarrow [-1, 1]$

$[y_{min}, y_{max}] \rightarrow [-1, 1]$

Visualize this.



We have:

$$\frac{x_p - x_{min}}{x_{max} - x_{min}} = \frac{x_n - (-1)}{1 - (-1)} = \frac{x_n + 1}{2}$$

$$\Rightarrow x_n = \frac{2(x_p - x_{min})}{x_{max} - x_{min}} - 1$$

$$= \frac{2}{x_{max} - x_{min}} x_p - \frac{x_{max} + x_{min}}{x_{max} - x_{min}}$$

Remember earlier we have

$$x_p = \frac{x_e}{\frac{z}{-d}}.$$

So now we have

$$x_n = \frac{2}{x_{max} - x_{min}} \times \frac{x_e}{\frac{z}{-d}} - \frac{x_{max} + x_{min}}{x_{max} - x_{min}}$$

P. 3

So modify the basic perspective matrix

$$\begin{bmatrix} \frac{2}{x_{max}-x_{min}} & 0 & \frac{1}{d}\left(\frac{x_{max}+x_{min}}{x_{max}-x_{min}}\right) & 0 \\ 0 & \frac{2}{y_{max}-y_{min}} & \frac{1}{d}\left(\frac{y_{max}+y_{min}}{y_{max}-y_{min}}\right) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_1}$$

Why? Because

$$M_{p_1} \times \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2x_e}{x_{max}-x_{min}} + \frac{z_e}{d}\cdot\frac{x_{max}+x_{min}}{x_{max}-x_{min}} \\ \frac{2y_e}{y_{max}-y_{min}} + \frac{z_e}{d}\,\frac{y_{max}+y_{min}}{y_{max}-y_{min}} \\ z_e \\ \frac{z_e}{-d} \end{bmatrix} = \begin{bmatrix} x_n' \\ y_n' \\ z_n' \\ w_n' \end{bmatrix}$$

Normalize (xn', yn', zn', wn')

$$x_n = \frac{2x_e}{x_{max}-x_{min}} \times \frac{z}{-d} - \frac{x_{max}+x_{min}}{x_{max}-x_{min}}$$

$$y_n = \frac{2y_e}{y_{max}-y_{min}} * \frac{z}{-d} - \frac{y_{max}+y_{min}}{y_{max}-y_{min}}$$

$$z_n = -d$$

$$w_n = 1$$

P. 4

But we are not done yet.

We need to map z range between [-near, -far]

to [-1, 1]. This requires change of M_{p_1}

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p_2}$$

(copy x from M_{p_1} in the previous page)

We need to find A & B

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -near \\ 1 \end{bmatrix} = \begin{bmatrix} xx \\ xx \\ A\cdot -near + B \\ \frac{near}{d} \end{bmatrix}$$

(xx means "don't care")

$$\Rightarrow \frac{A\cdot -near + B}{\frac{near}{d}} = -1$$

$$\Rightarrow -Ad + \frac{B\cdot d}{near} = -1 \quad \rightarrow (1)$$

P. 5

Also:

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & A & B \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ -far \\ 1 \end{bmatrix} = \begin{bmatrix} x\ x \\ x\ x \\ A \cdot -far + B \\ \frac{far}{d} \end{bmatrix}$$

$$\Rightarrow \frac{A \cdot -far + B}{\frac{far}{d}} = +1$$

$$\Rightarrow -Ad + \frac{B \cdot d}{far} = 1 \quad \text{—— (2)}$$

Solve A. B from (1) & (2):

$$A = \frac{N+F}{(N-F)d} \qquad B = \frac{2N \times F}{d(N-F)}$$

where $N = near$ $F = far$

so we have the new projection matrix

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & 0 & \frac{-(N+F)}{(F-N)d} & \frac{-2NF}{d(F-N)} \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix}$$

P. 6

That is:

$$\begin{bmatrix} \frac{2}{x_{max}-x_{min}} & 0 & \frac{x_{max}+x_{min}}{x_{max}-x_{min}} & 0 \\ 0 & \frac{2}{y_{max}-y_{min}} & \frac{y_{max}+y_{min}}{y_{max}-y_{min}} & 0 \\ 0 & 0 & \frac{-(N+F)}{(F-N)d} & \frac{-2NF}{(F-N)d} \\ 0 & 0 & \frac{1}{-d} & 0 \end{bmatrix} = M_{p2}$$

Scale ~~Multiply~~ the whole matrix by d.

$$\begin{bmatrix} \frac{2d}{x_{max}-x_{min}} & 0 & \frac{x_{max}+x_{min}}{x_{max}-x_{min}} & 0 \\ 0 & \frac{2d}{y_{max}-y_{min}} & \frac{y_{max}+y_{min}}{y_{max}-y_{min}} & 0 \\ 0 & 0 & \frac{-(N+F)}{F-N} & \frac{-2NF}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

now, if we set the image plane at the near plane, that is, $d = N$.

then we have the following Final matrix

$$
\begin{bmatrix}
\frac{2N}{X_{max} - X_{min}} & 0 & \frac{X_{max} + X_{min}}{X_{max} - X_{min}} & 0 \\
0 & \frac{2N}{Y_{max} - Y_{min}} & \frac{Y_{max} + Y_{min}}{Y_{max} - Y_{min}} & 0 \\
0 & 0 & \frac{-(N+F)}{F-N} & \frac{-2NF}{F-N} \\
0 & 0 & -1 & 0
\end{bmatrix}
$$

OpenGL perspective projection Matrix

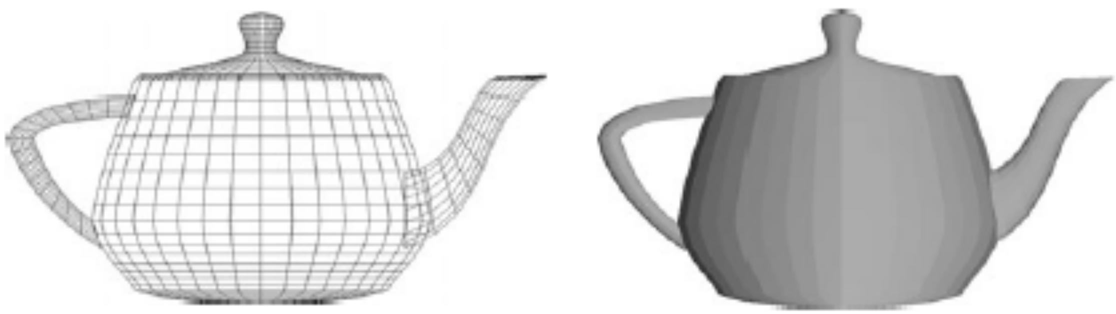


FIGURE 10.41 Rendered teapots.

CHAPTER 10

CURVES AND SURFACES

Modeling

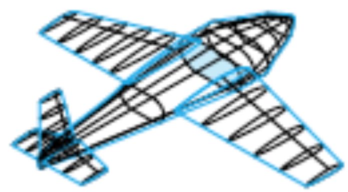


FIGURE 10.5 Model airplane

FIGURE 10.6 Cross-section curve.

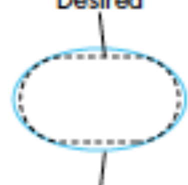


FIGURE 10.7 Approximation of cross-section curve.

Parametric Curve

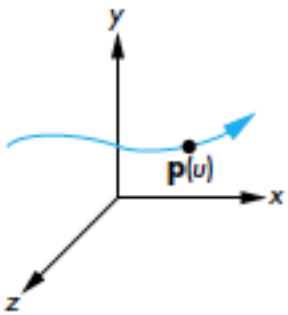


FIGURE 10.1 Parametric

$$x = x(u),$$
$$y = y(u),$$
$$z = z(u).$$

$$\frac{d\mathbf{p}(u)}{du} = \begin{bmatrix} \frac{dx(u)}{du} \\ \frac{dy(u)}{du} \\ \frac{dz(u)}{du} \end{bmatrix}$$

Parametric Curve

Consider a curve of the form[2]

$$\mathbf{p}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}.$$

A polynomial parametric curve of degree[3] n is of the form

$$\mathbf{p}(u) = \sum_{k=0}^{n} u^k \mathbf{c}_k,$$

where each $\mathbf{c}_k$ has independent x, y, and z components; that is,

$$\mathbf{c}_k = \begin{bmatrix} c_{xk} \\ c_{yk} \\ c_{zk} \end{bmatrix}.$$

The $n+1$ column matrices $\{\mathbf{c}_k\}$ are the coefficients of $\mathbf{p}$; they give us $3(n+1)$ degrees of freedom in how we choose the coefficients of a particular $\mathbf{p}$. There is no coupling, however, among the x, y, and z components, so we can work with three independent equations, each of the form

$$p(u) = \sum_{k=0}^{n} u^k c_k,$$

where p is any one of x, y, or z. There are $n+1$ degrees of freedom in $p(u)$. We can define our curves for any range interval of u:

$$u_{min} \le u \le u_{max};$$

however, with no loss of generality (see Exercise 10.3), we can assume that $0 \le u \le 1$. As the value of u varies over its range, we define a **curve segment**, as shown in Figure 10.3.

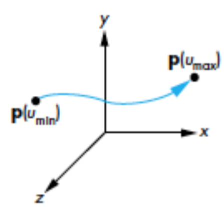


FIGURE 10.3 Curve segment.

Cubic Parametric Curves

$$\mathbf{p}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3 = \sum_{k=0}^{3} \mathbf{c}_k u^k = \mathbf{u}^T \mathbf{c},$$

where

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}, \qquad \mathbf{c}_k = \begin{bmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{bmatrix}.$$

Control Points

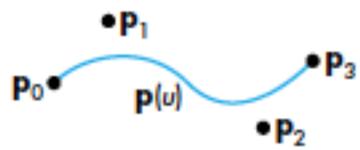


FIGURE 10.9 Curve segment and control points.

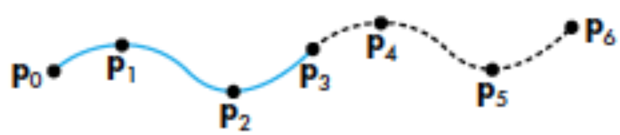


FIGURE 10.10 Joining of interpolating segments.

Bezier

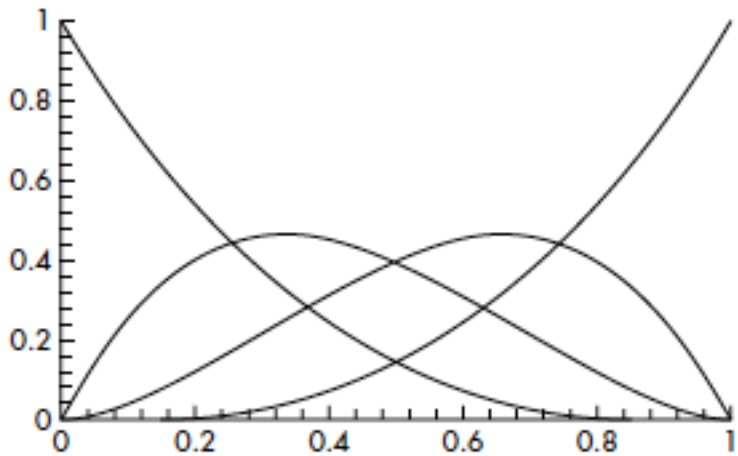


FIGURE 10.18 Blending polynomials for the Bézier cubic.

$$\mathbf{p}(u) = \sum_{i=0}^{3} b_i(u)\mathbf{p}_i,$$

$$\mathbf{p}(u) = \mathbf{b}(u)^T\mathbf{p},$$

$$\mathbf{b}(u) = \mathbf{M}_B^T\mathbf{u} = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}.$$

$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

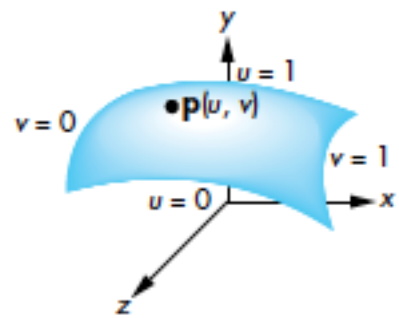
Parametric Surface

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix}$$

$$x = x(u, v),$$
$$y = y(u, v),$$
$$z = z(u, v),$$

$$\frac{\partial \mathbf{p}}{\partial u} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial u} \\ \frac{\partial y(u,v)}{\partial u} \\ \frac{\partial z(u,v)}{\partial u} \end{bmatrix} \qquad \frac{\partial \mathbf{p}}{\partial v} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial v} \\ \frac{\partial y(u,v)}{\partial v} \\ \frac{\partial z(u,v)}{\partial v} \end{bmatrix}$$

$)/\partial v.$



$\mathbf{p}(u, v) =$ FIGURE 10.4 Surface patch.

Parametric Surface

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} u^i v^j.$$

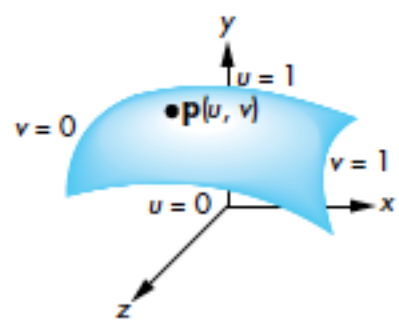


FIGURE 10.4 Surface patch.

Bezier Surface Patches

$$\mathbf{p}(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u) b_j(v) \mathbf{p}_{ij} = \mathbf{u}^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T \mathbf{v}.$$

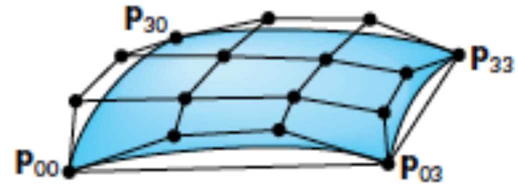


FIGURE 10.20 Bézier patch.

Subdivision

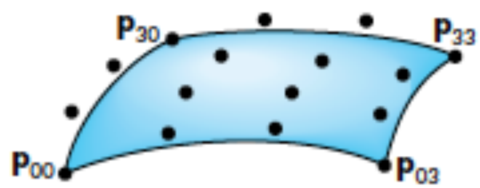


FIGURE 10.37 Cubic Bézier surface.

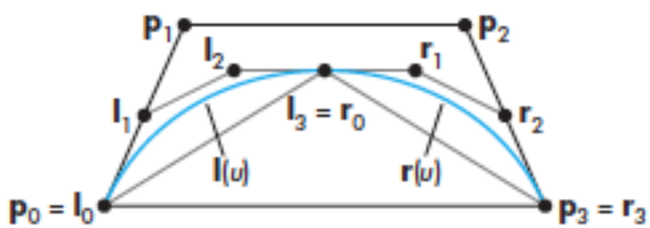


FIGURE 10.34 Convex hulls and control points.

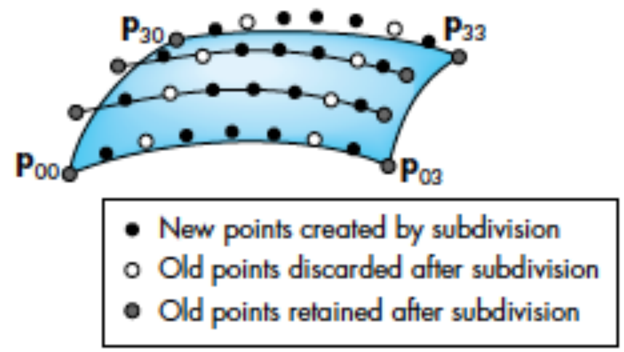


FIGURE 10.38 First subdivision of surface.

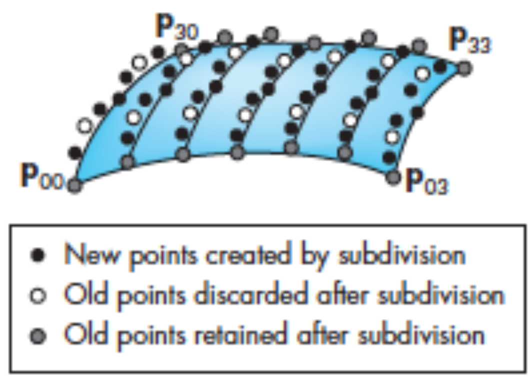


FIGURE 10.39 Points after second subdivision.

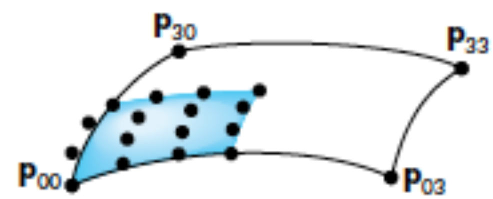


FIGURE 10.40 Subdivided quadrant.

Code

```
void draw_patch(point4 p[4][4])
{
    points[n] = p[0][0];
    n++;
    points[n] = p[3][0];
    n++;
    points[n] = p[3][3];
    n++;
    points[n] = p[0][3];
    n++;
}
```

```
void divide_curve(point4 c[4], point4 r[4], point4 l[4])
{

/* division of convex hull of Bezier curve */

   int i;
   point4 t;
   for(i=0;i<3;i++)

       l[0][i]=c[0][i];
       r[3][i]=c[3][i];
       l[1][i]=(c[1][i]+c[0][i])/2;
       r[2][i]=(c[2][i]+c[3][i])/2;
       t[i]=(l[1][i]+r[2][i])/2;
       l[2][i]=(t[i]+l[1][i])/2;
       r[1][i]=(t[i]+r[2][i])/2;
       l[3][i]=r[0][i]=(l[2][i]+r[1][i])/2;

   for(i=0; i<4; i++) l[i][3] = r[i][3] = 1.0;
}
```

```
void divide_patch(point4 p[4][4], int n)
{
   point4 q[4][4], r[4][4], s[4][4], t[4][4];
   point4 a[4][4], b[4][4];
   int k;
   if(n==0) draw_patch(p); /* draw patch if recursion done */

/* subdivide curves in u direction, transpose results, divide
in u direction again (equivalent to subdivision in v) */

   else
       {
       for(k=0; k<4; k++) divide_curve(p[k], a[k], b[k]);
       transpose4(a);
       transpose4(b);
       for(k=0; k<4; k++)
          {
          divide_curve(a[k], q[k], r[k]);
          divide_curve(b[k], s[k], t[k]);
          }

/* recursive division of 4 resulting patches */

       divide_patch(q, n-1);
       divide_patch(r, n-1);
       divide_patch(s, n-1);
       divide_patch(t, n-1);
      }
}
```

Code for GL

Courtesy:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL).

Provides classes and functions designed and implemented following as strictly as possible the GLSL conventions and functionalities.

When a programmer knows GLSL, he knows GLM as well, making it really easy to use.

C++

```
glm::mat4 myMatrix;
glm::vec4 myVector;

// fill myMatrix and myVector somehow

glm::vec4 transformedVector = myMatrix * myVector;

 // Again, in this order ! this is important.
```

GLSL

```
mat4 myMatrix;
vec4 myVector;

// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector;

// Yeah, it's pretty much the same than GLM
```

Identity

```
glm::mat4 myIdentityMatrix = glm::mat4(1.0f);
```

Translate

GLM -

```
#include <glm/transform.hpp> // after <glm/glm.hpp>
glm::mat4 myMatrix = glm::translate(10.0f, 0.0f, 0.0f);
glm::vec4 myVector(10.0f, 10.0f, 10.0f, 0.0f);
glm::vec4 transformedVector = myMatrix * myVector;
```

GLSL -

```
vec4 transformedVector = myMatrix * myVector;
```

Scaling

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include
<glm/gtx/transform.hpp>

glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f ,2.0f);
```

Rotation

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include
<glm/gtx/transform.hpp>

glm::vec3 myRotationAxis( ??, ??, ??);

glm::rotate( angle_in_degrees, myRotationAxis );
```

Accumulating Transforms

TransformedVector =
TranslationMatrix * RotationMatrix * ScaleMatrix * OriginalVector;

In Code

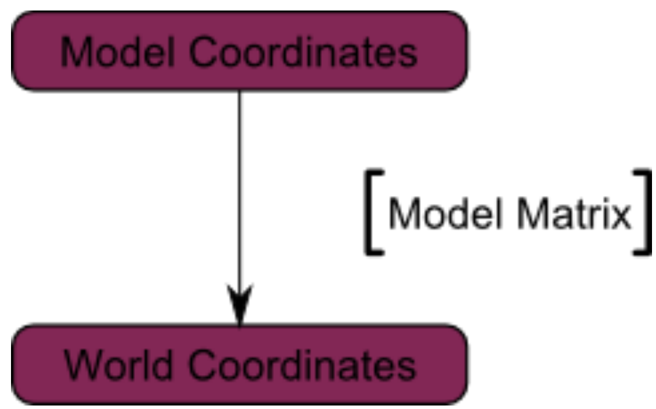
GLM

```
glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;

glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
```

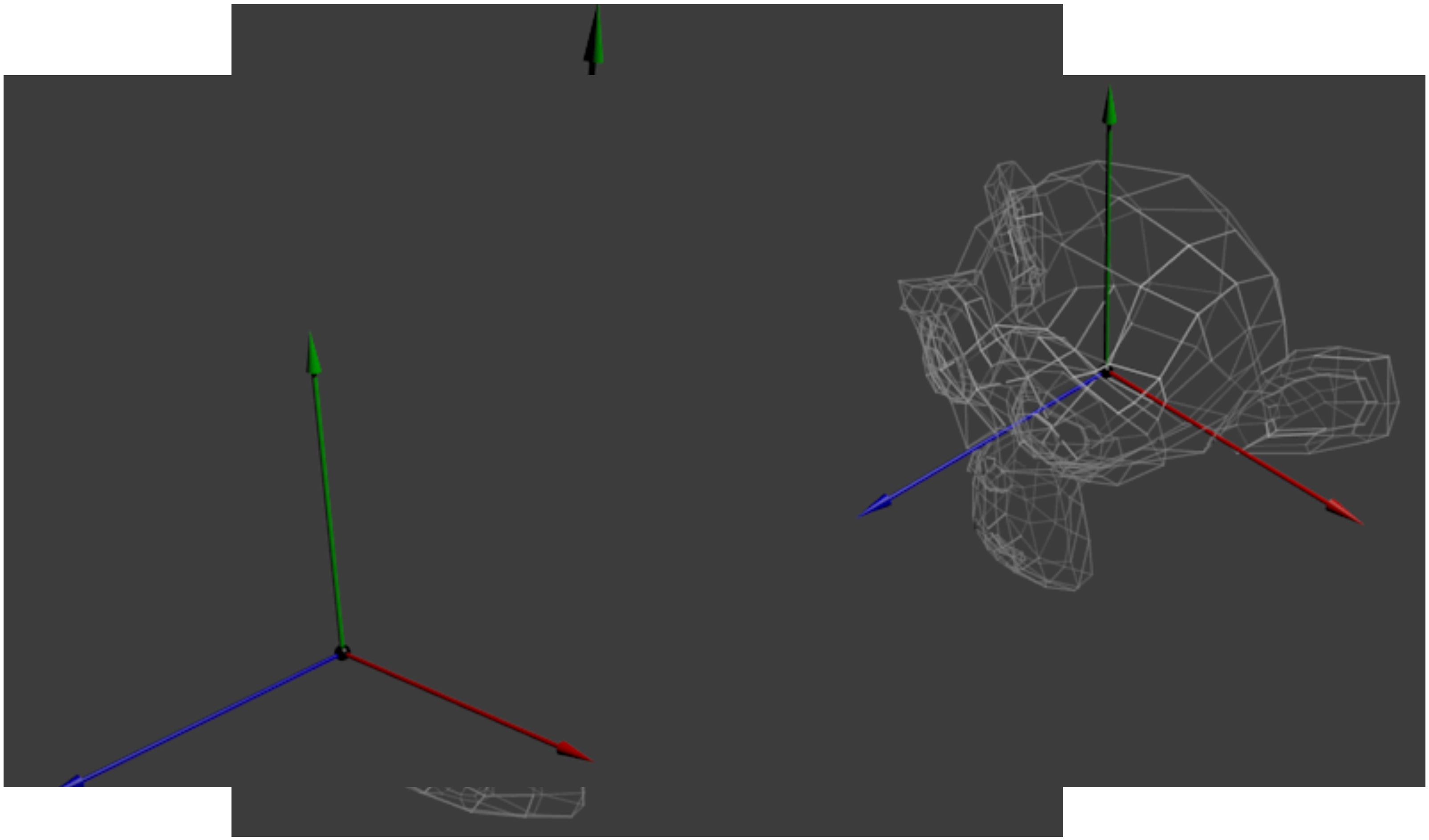
GLSL

```
mat4 transform = mat2 * mat1;
vec4 out_vec = transform * in_vec;
```

In Diagrams



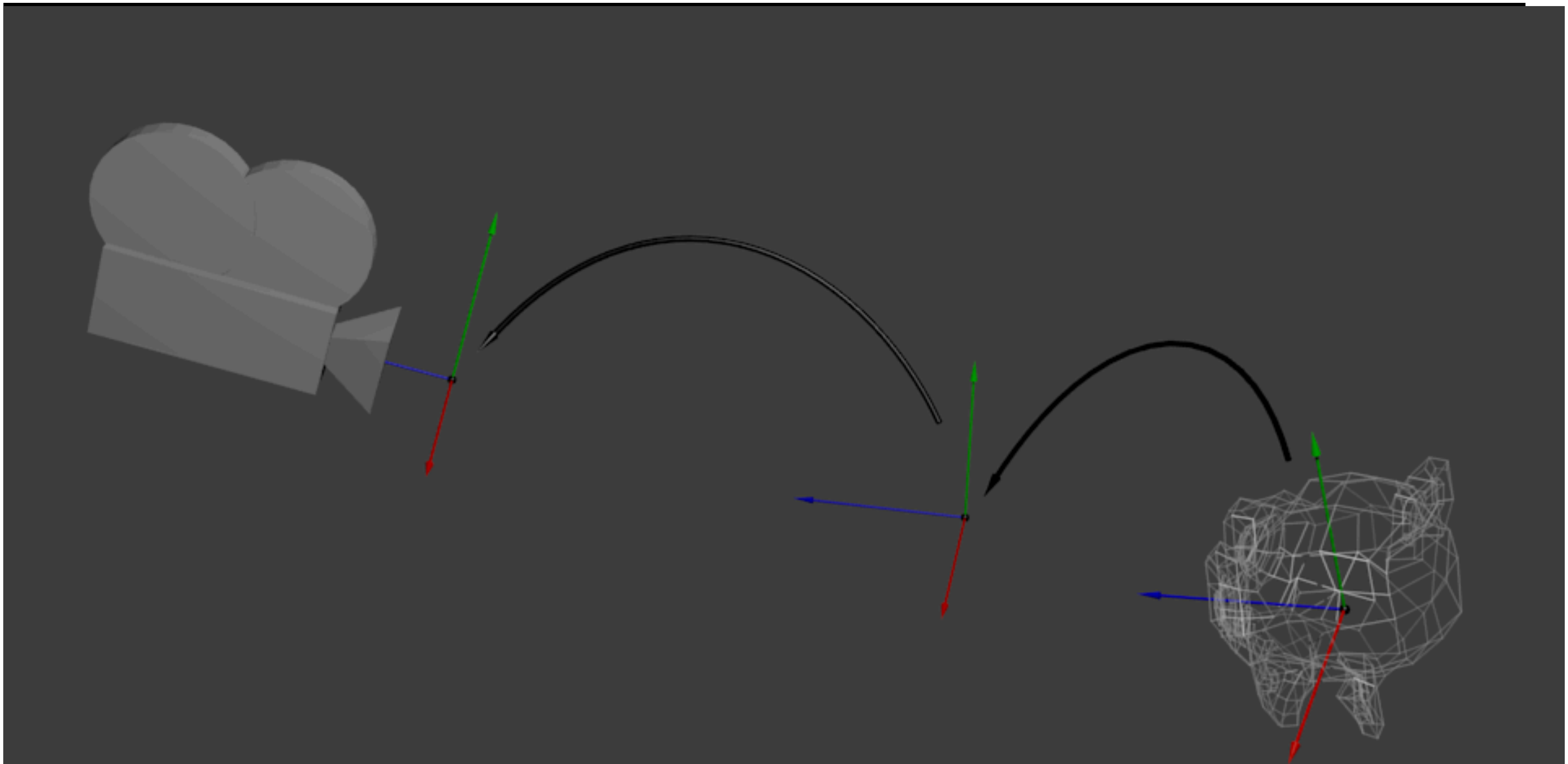
In Pictures



DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING


Camera

Camera/Eye Space

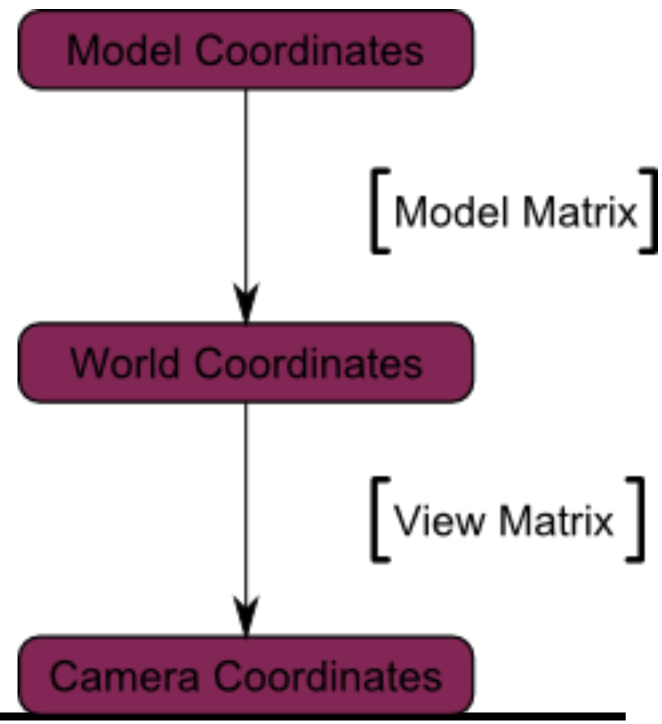


```
glm::mat4 ViewMatrix = glm::translate(-3.0f, 0.0f ,0.0f);
```

Camera/Eye Space

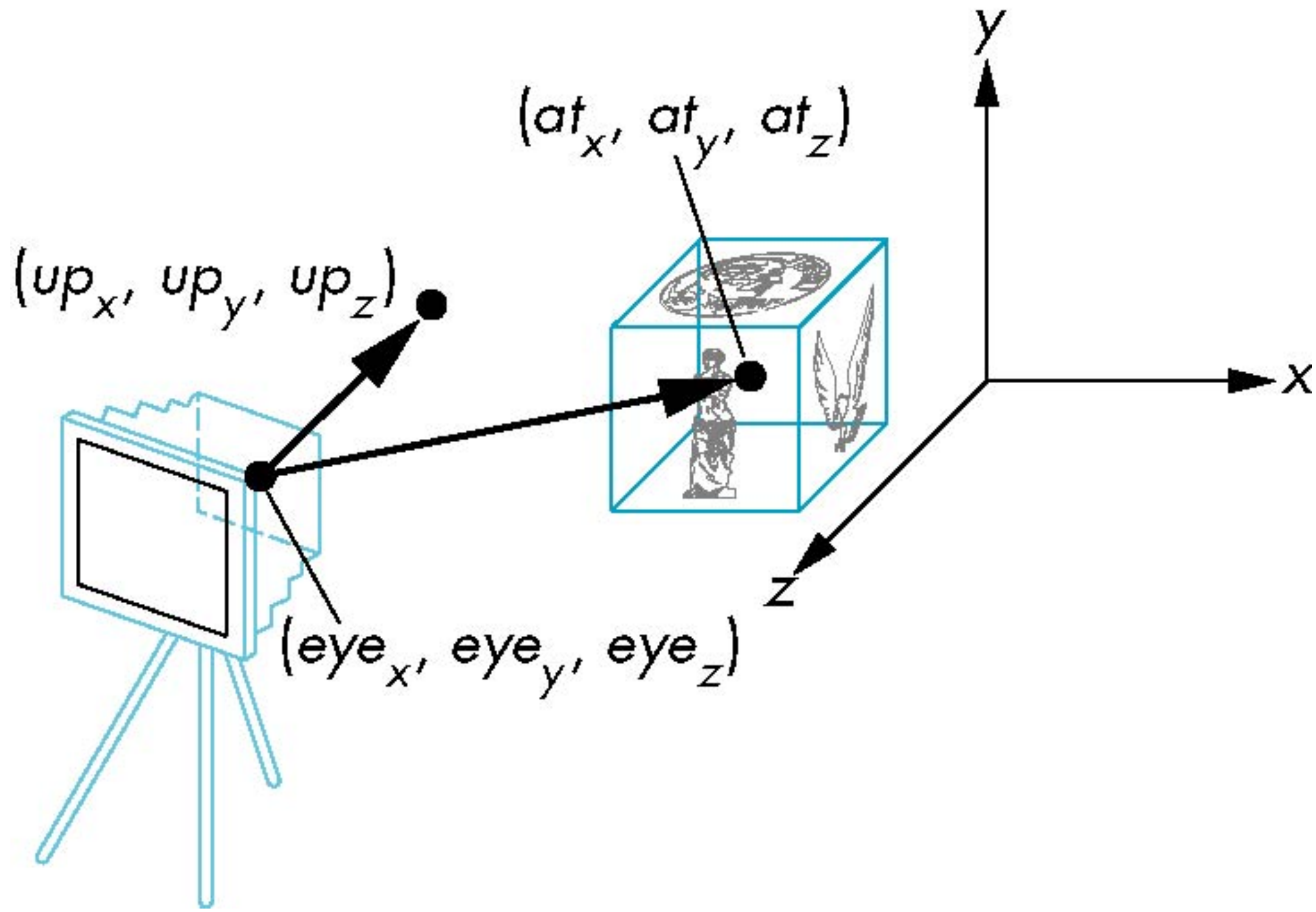
```
glm::mat4 CameraMatrix = glm::LookAt (
    cameraPosition, // the position of your camera, in world space
    cameraTarget,   // where you want to look at, in world space
    upVector        // probably glm::vec3(0,1,0),
                    // but (0,-1,0) would make you looking upside-down
);
```

<u>Transform objects from world to eye space</u>

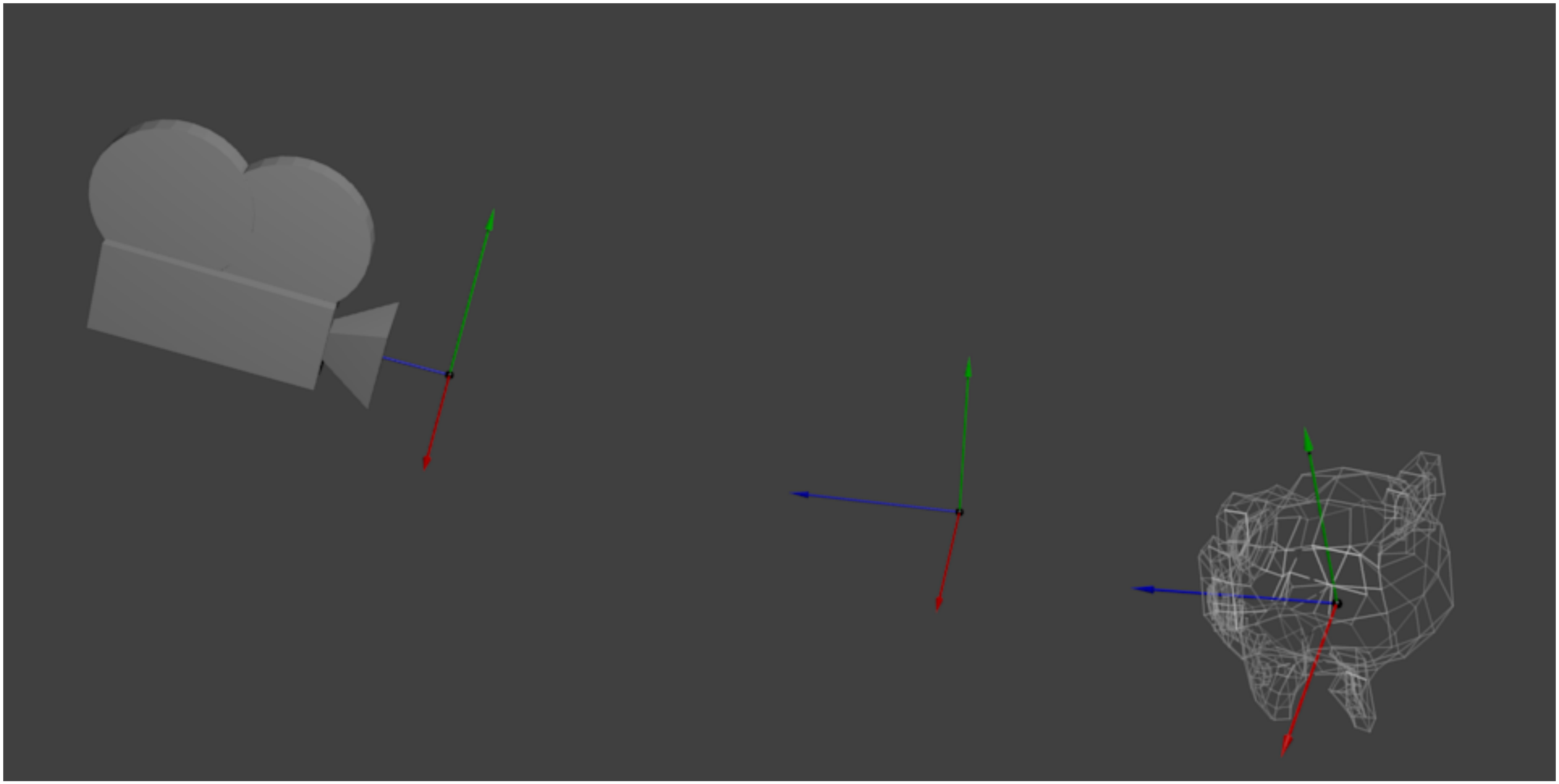


gluLookAt

LookAt(eye, at, up)



Camera Coordinate Frame



Camera Space

Right hand coordinate system

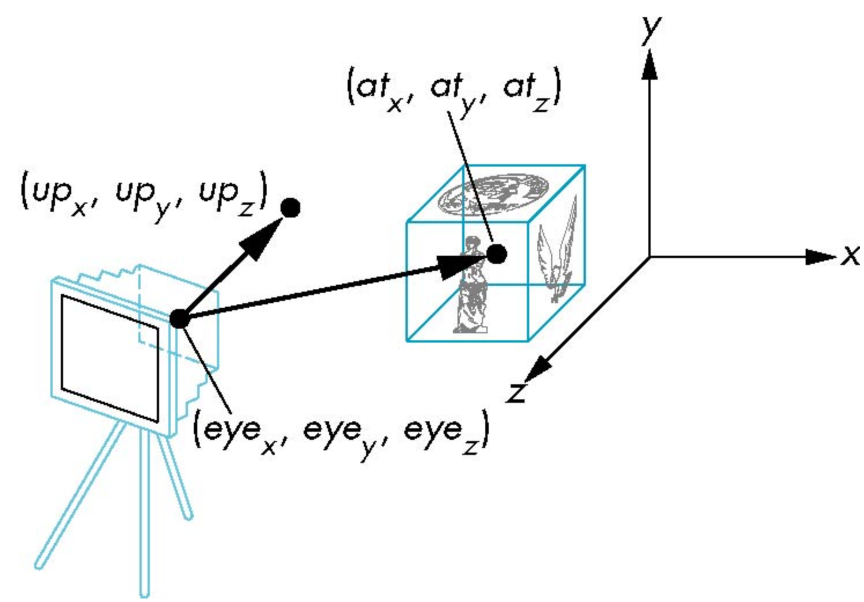
$$\vec{n} = at - eye$$

$$\vec{n} = \frac{\vec{n}}{\|\vec{n}\|}$$

$$\vec{u} = up \times \vec{n}$$

$$v = \vec{n} \times \vec{u}$$

$$\boldsymbol{V} = \begin{pmatrix} u_x & u_y & u_z & -eye \cdot \mathbf{u} \\ v_x & v_y & v_z & -eye \cdot \mathbf{v} \\ n_x & n_y & n_z & -eye \cdot \mathbf{n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Old Style

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    …
}
```

New World

- Create a view matrix

```
view = glm::lookAt(glm::vec3(0.0, 2.0, 2.0), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0,
1.0, 0.0));
```

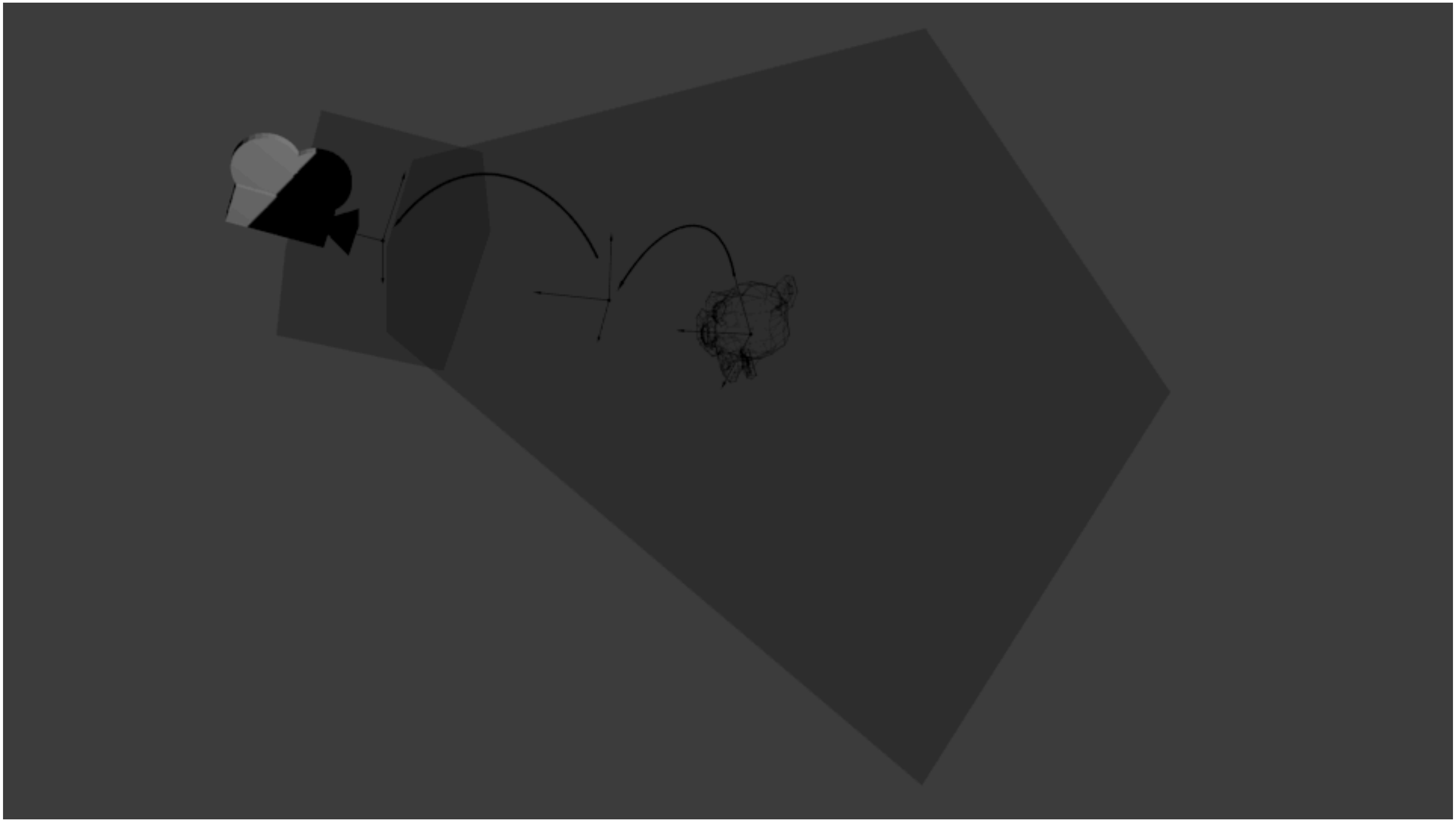
- Combine with modeling matrices

```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, angle, glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::scale(model, scale_size, scale_size, scale_size);

glm::mat4 modelview = view * model;
```

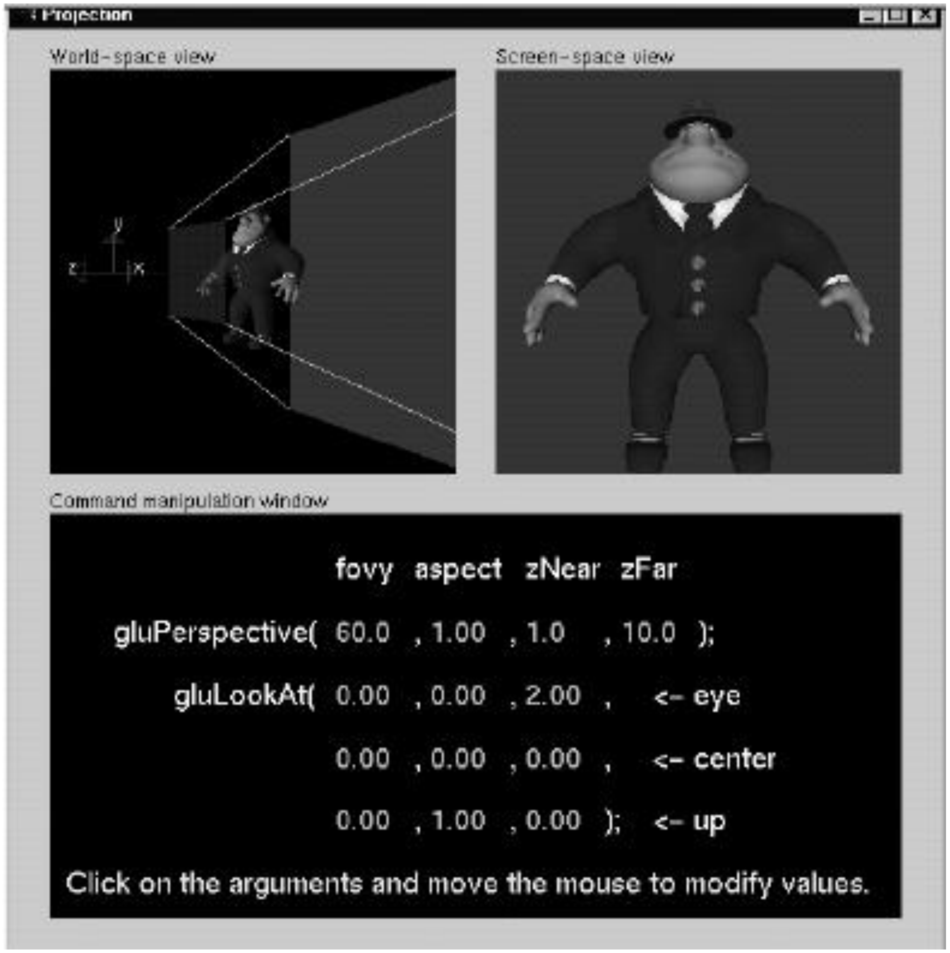
Working with Old World

```
glMatrixMode(GL_MODELVIEW);
 glLoadMatrixf(&modelview[0][0]);

// begin to draw your geometry
…
```

Projection Matrices



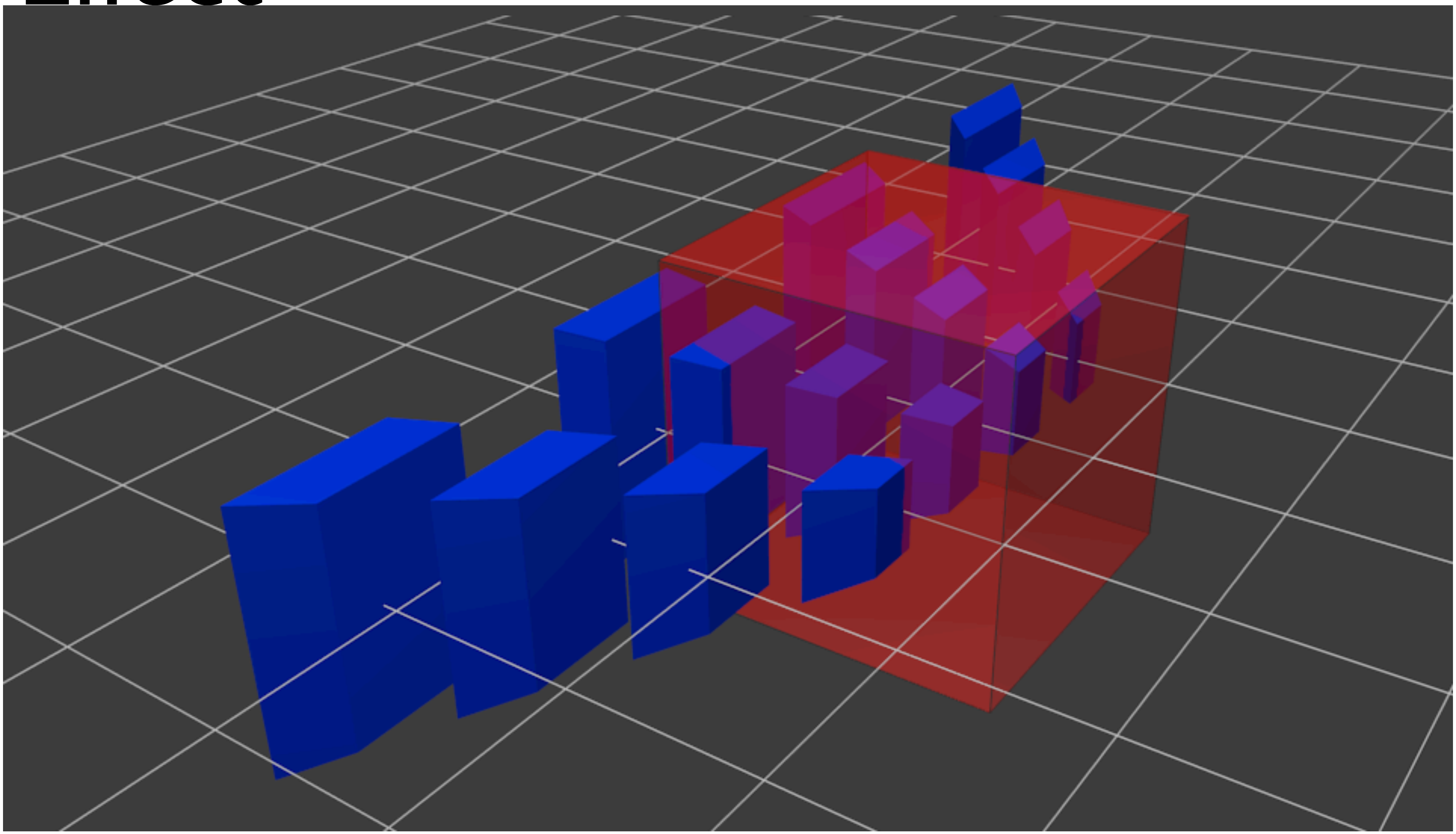
Demo


Projection
World-space view
Screen-space view
Command manipulation window
fovy aspect zNear zFar
gluPerspective(60.0 , 1.00 , 1.0 , 10.0);
gluLookAt(0.00 , 0.00 , 2.00 , <- eye
0.00 , 0.00 , 0.00 , <- center
0.00 , 1.00 , 0.00); <- up
Click on the arguments and move the mouse to modify values.

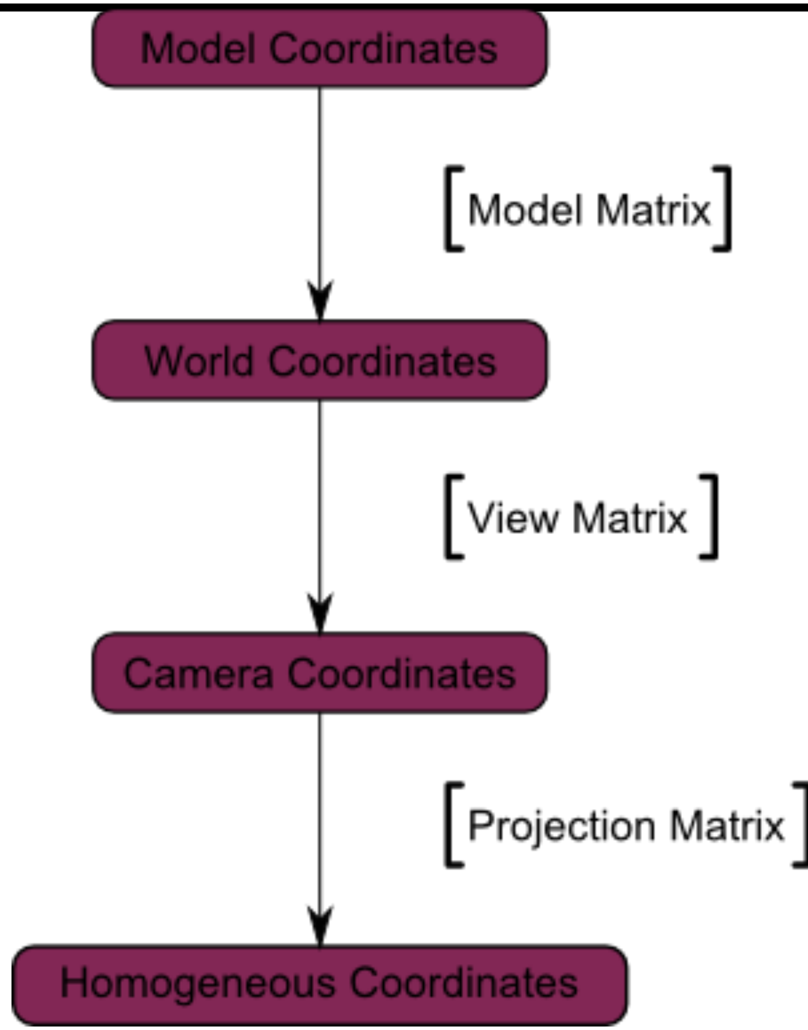
In Code

```
// Generates a really hard-to-read matrix, but a normal, standard 4x4 matrix nonetheless

glm::mat4 projectionMatrix = glm::perspective(

    FoV,          // The horizontal Field of View, in degrees : the amount of "zoom".
                  // Think "camera lens". Usually between 90° (extra wide) and 30° (quite zoomed in)
    4.0f / 3.0f,  // Aspect Ratio. Depends on the size of your window.
                  //Notice that 4/3 == 800/600 == 1280/960, sounds familiar ?
    0.1f,         // Near clipping plane. Keep as big as possible, or you'll get precision issues.

    100.0f        // Far clipping plane. Keep as little as possible.
);
```

Effect



In Diagrams


Model Coordinates
Model Matrix
World Coordinates
View Matrix
Camera Coordinates
Projection Matrix
Homogeneous Coordinates

More Code

C++ : compute the matrix

```
glm::mat4 MVPmatrix = projection * view * model;
// Remember : inverted !
```

```
// GLSL : apply it
transformed_vertex = MVP * in_vertex;
```

Combined

Generate Matrix

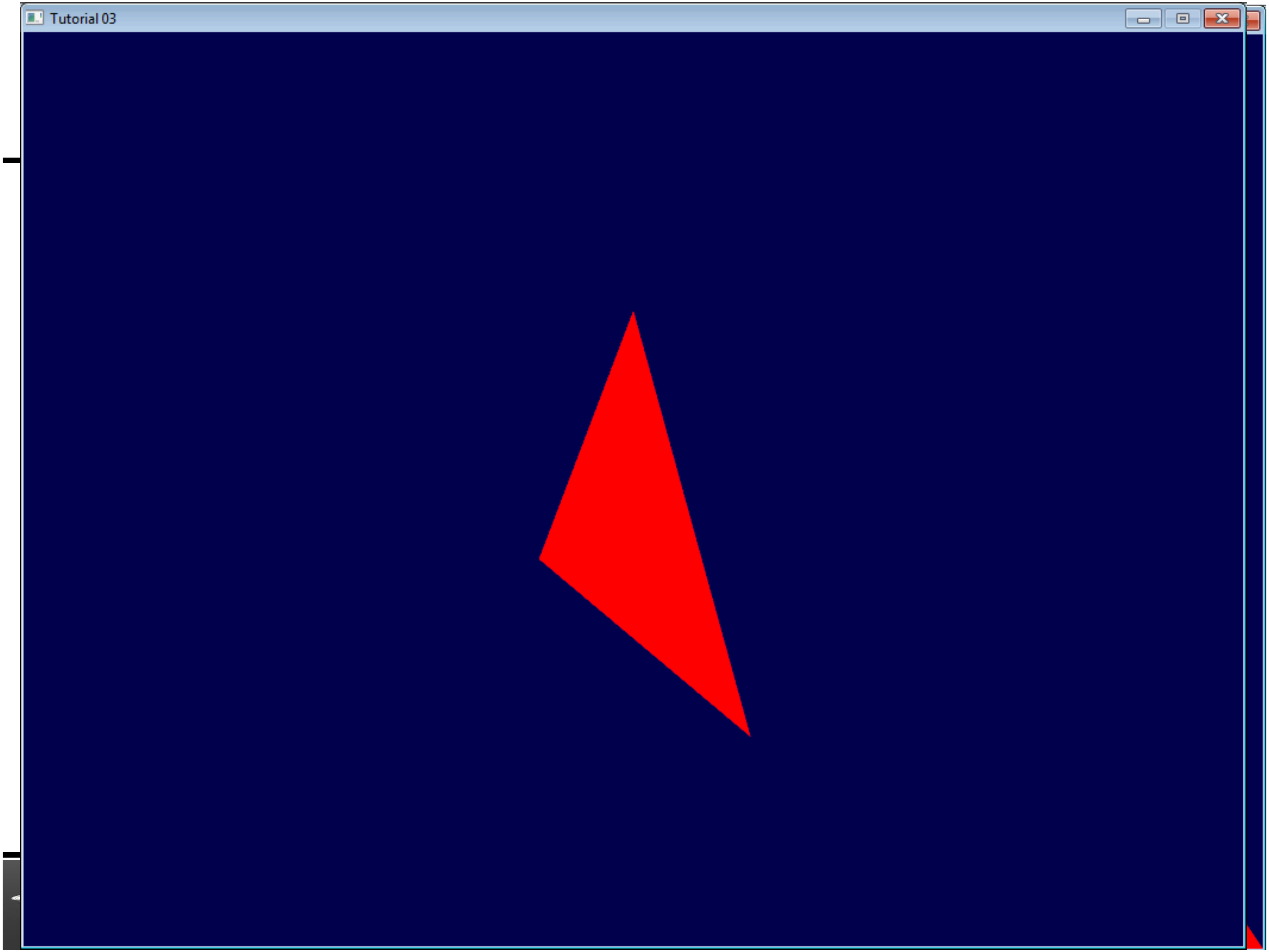
```
// Projection matrix : 45°
//Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);

// Camera matrix
glm::mat4 View       = glm::lookAt(
    glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
    glm::vec3(0,0,0), // and looks at the origin
    glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
);
// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model      = glm::mat4(1.0f);  // Changes for each model !
// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP        = Projection * View * Model;
// Remember, matrix multiplication is the other way around
```

GLSL Takes Over

```
// Get a handle for our "MVP" uniform.
// Only at initialisation time.
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
// For each model you render, since the MVP will be different
// (at least the M part)

glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

Use It

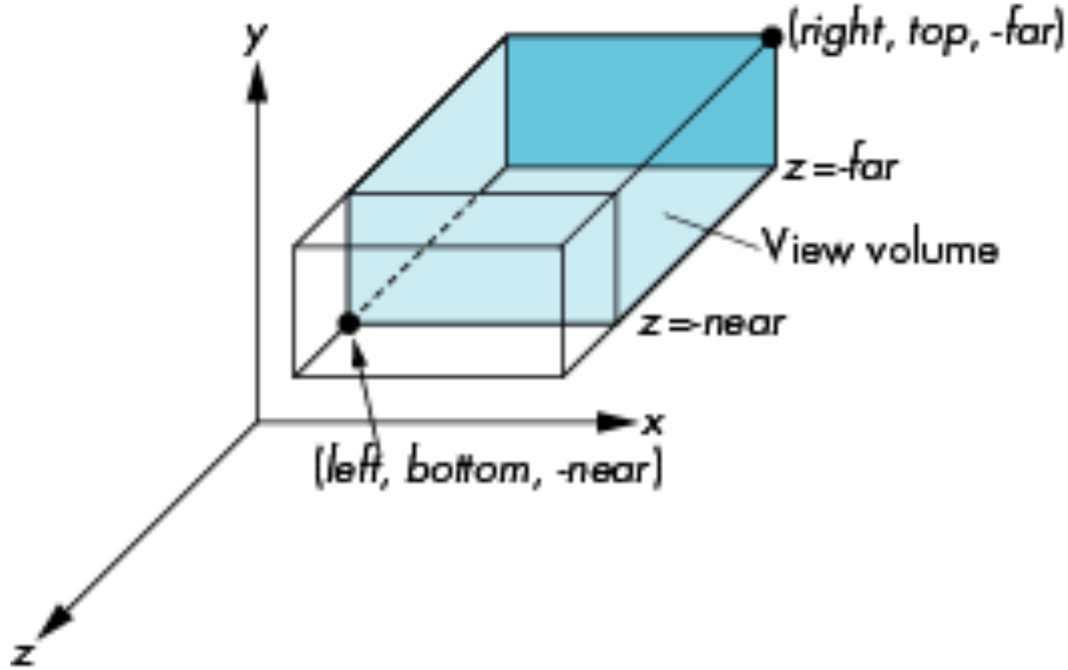
```
in vec3 vertexPosition_modelspace;
uniform mat4 MVP;

void main(){
// Output position of the vertex, in clip space : MVP * position

    vec4 v = vec4(vertexPosition_modelspace,1);

// Transform an homogeneous 4D vector, remember ?
    gl_Position = MVP * v;

}
```


Tutorial 03

Old Style

OpenGL Orthogonal Viewing

`Ortho(left,right,bottom,top,near,far)`

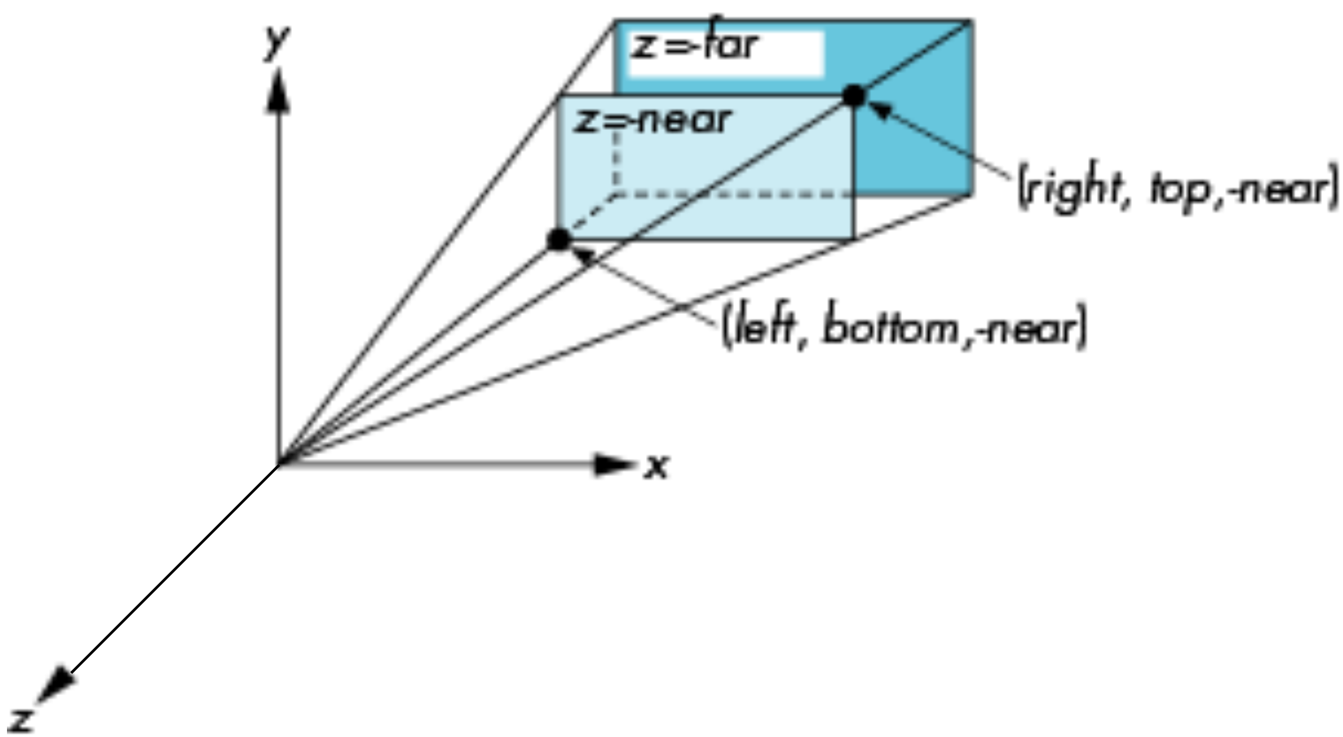


`near` and **`far`** measured <u>from</u> camera

THE OHIO STATE UNIVERSITY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

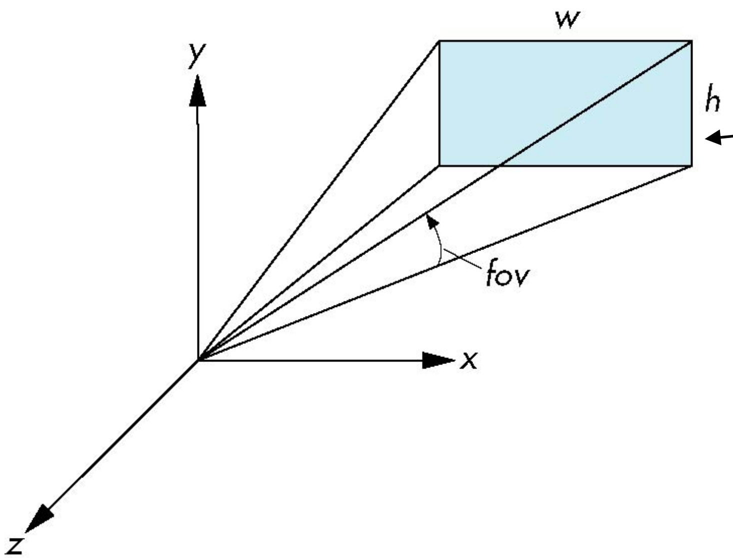
OpenGL Perspective

Frustum(left,right,bottom,top,near,far)



Using Field of View

- With Frustum it is often difficult to get the desired view
- Perpective(fovy, aspect, near, far) often provides a better interface



front plane

aspect = w/h

Old Style

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJETION);
    glLoadIdentity();
    gluPerspective(fove, aspect, near, far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,1,0,0,0,0,1,0);
    my_display();    // your display routine
}
```

Can Still GLM

- Set up the projection matrix

```
glm::mat4 projection = glm::mat4(1.0f);
projection = glm::perspective(60.0f,1.0f,.1f,100.0f);
```

- Load the matrix to GL_PROJECTION

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(&projection[0][0]);
```

Next