Viewing and OpenGL

EDA 221
Jacob Munkberg

Today

- Camera setup
- Viewing and Projection
- Modern OpenGL
  - Walkthrough of simple program

Setup Camera Matrix

- LookAt function
  - Takes eye position, a position to look at and an up vector
  - Constructs the **View** matrix, i.e., a matrix that transforms geometry (in world space) into the camera's coordinate system (camera space)

```
mat4 View =
            LookAt(eye.x,eye.y,eye.z, // Camera position
                     0.0,0.0,0.0,         // Center of interest
                     0.0f,1.0f,0.0f);   // Up-vector
```

Task at Hand

- Setup an OpenGL camera
- Find matrix that transforms an arbitrary camera to the origin, looking along the negative z axis

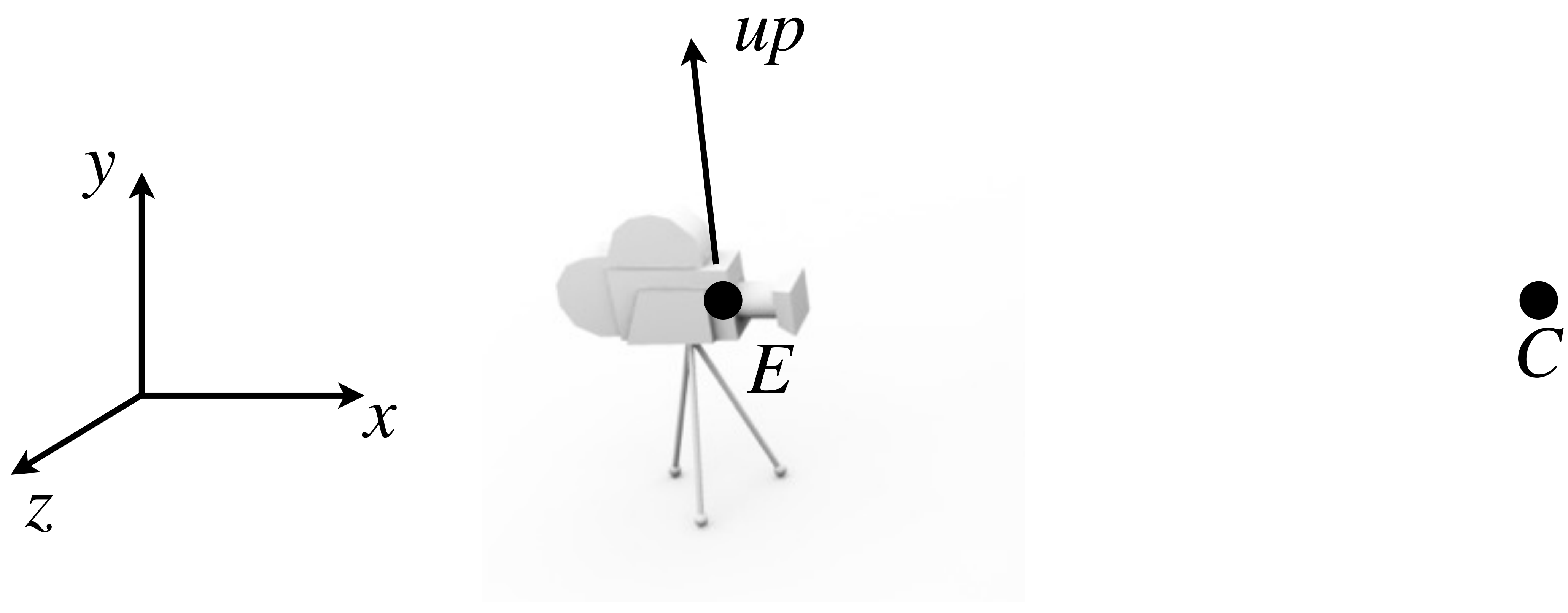
OpenGL convention

- In OpenGL: right-hand coordinate system, looking down *-z*.

Camera Placement

Derivation from Ravi Ramamoorthi

- Specify camera position (E), center of interest (C) and up-vector (up)



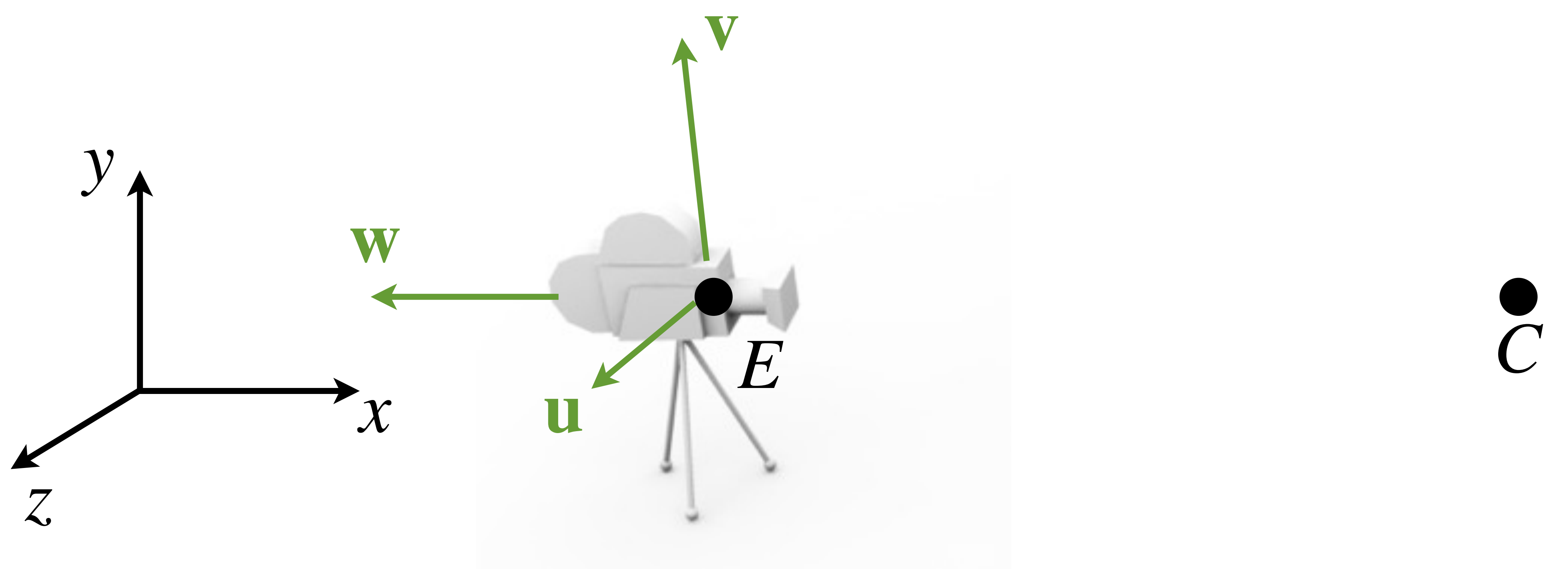
Find orthonormal basis

Derivation from Ravi Ramamoorthi

- OpenGL standard: camera looks along negative z. Choose $\mathbf{w}$ = - $(C$-$E)$

$$\mathbf{a} = E - C \qquad \mathbf{b} = up$$

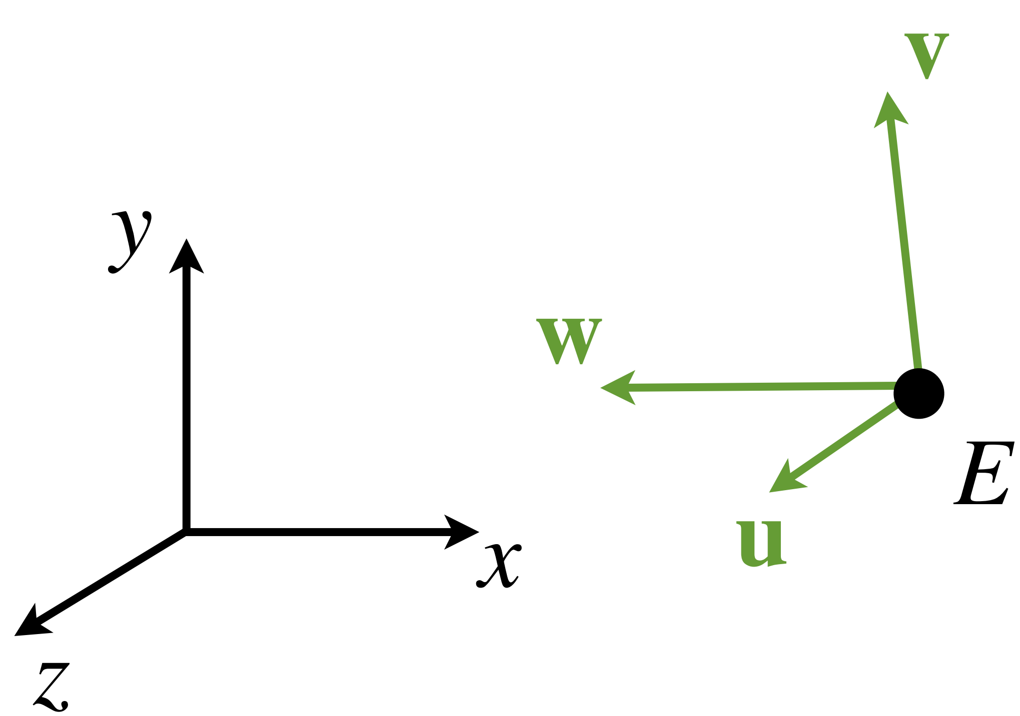
$$\mathbf{w} = \frac{\mathbf{a}}{|\mathbf{a}|} \qquad \mathbf{u} = \frac{\mathbf{b} \times \mathbf{w}}{|\mathbf{b} \times \mathbf{w}|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$



Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Now, we look for matrix that transforms basis $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ to $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$
- Translation and rotation

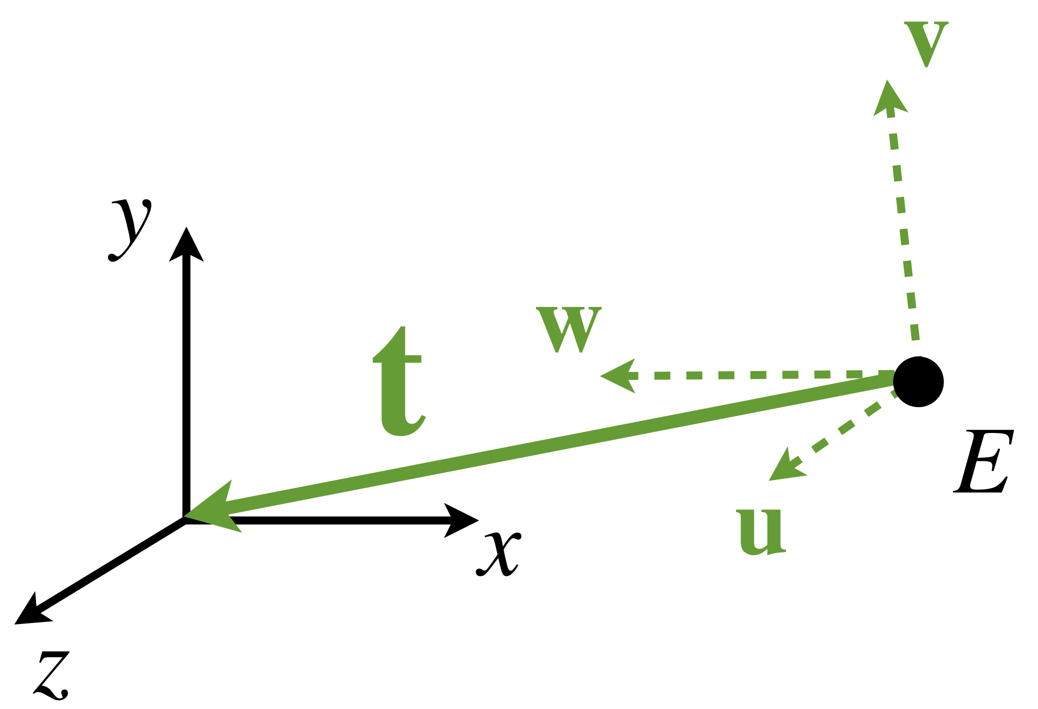


Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Translate **uvw** frame so that the origin align with the **xyz** frame

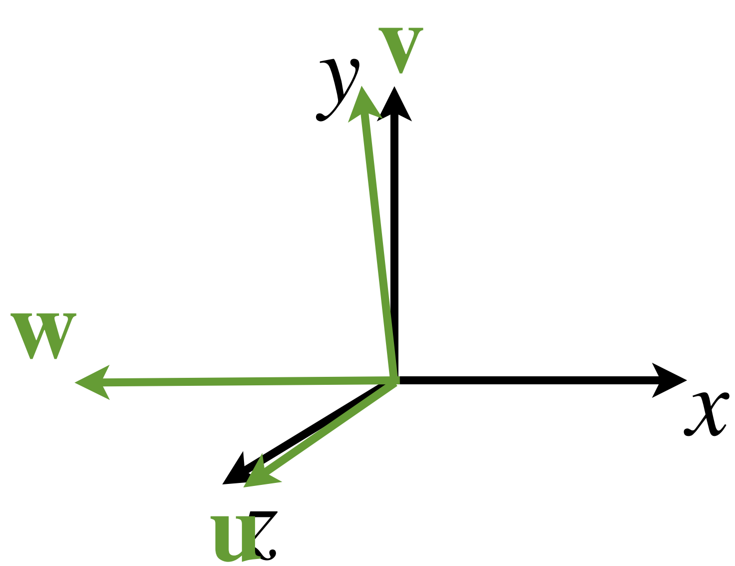
$$\mathbf{t} = \begin{bmatrix} -E_x \\ -E_y \\ -E_z \end{bmatrix}$$



Find orthonormal basis

Derivation from Ravi Ramamoorthi

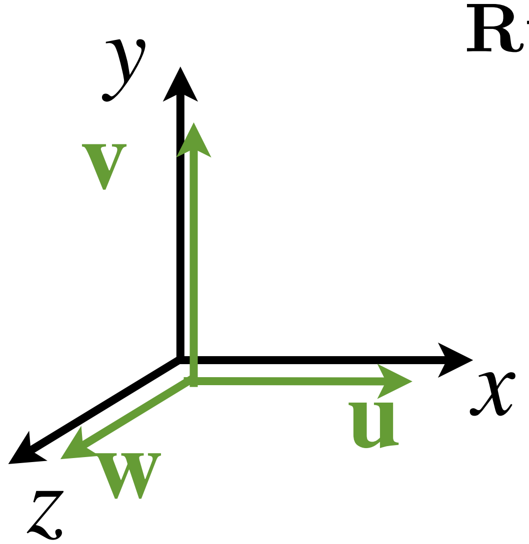
- Translate **uvw** frame so that the origin align with the **xyz** frame



Find orthonormal basis

Derivation from Ravi Ramamoorthi

- Then rotate $\mathbf{uvw}$ system so that the three axis align, $\mathbf{u}$ // $\mathbf{x}$, $\mathbf{v}$ // $\mathbf{y}$ and $\mathbf{w}$ // $\mathbf{z}$
- Rotation matrix given by $\mathbf{R} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix}$
- R rotates vectors $\mathbf{uvw}$ to $\mathbf{xyz}$



$$\mathbf{Ru} = \begin{bmatrix} - & \mathbf{u} & - \\ - & \mathbf{v} & - \\ - & \mathbf{w} & - \end{bmatrix} \begin{bmatrix} | \\ \mathbf{u} \\ | \end{bmatrix} = \begin{bmatrix} \mathbf{u} \cdot \mathbf{u} \\ \mathbf{v} \cdot \mathbf{u} \\ \mathbf{w} \cdot \mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

$$\mathbf{Rv} = \mathbf{y}, \quad \mathbf{Rw} = \mathbf{z}$$

Camera Placement

Derivation from Ravi Ramamoorthi

- Combine the two transforms
- Move to center, and apply rotation

$$M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate — Move to center

Workflow

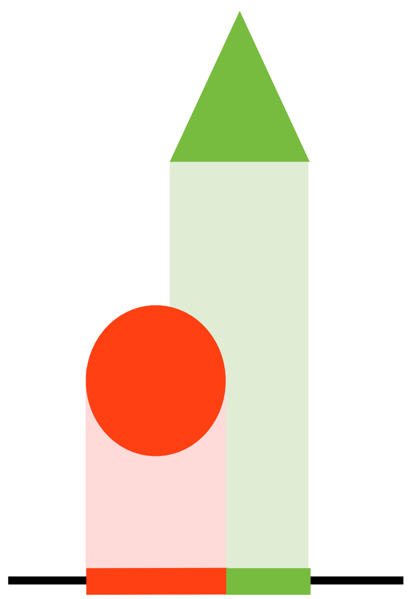
- OpenGL geometry workflow
  - Place camera in scene
  - Find **View** transform that moves camera to origin, looking along $-z$.
  - Place geometry in scene using **Model** (or **World**) transform
  - Setup camera **Projection** matrix (3D->2D)
  - Apply **ModelViewProjection** matrix to all geometry in the scene in vertex shader

Projection

- From 3D to a 2D image
  - Orthographic projection
  - Perspective projection
- Lines map to lines
  - Projective transform **does not** preserve parallel lines, angles or distances!
  - Demo: Nate Robins Tutors http://user.xmission.com/~nate/tutors.html

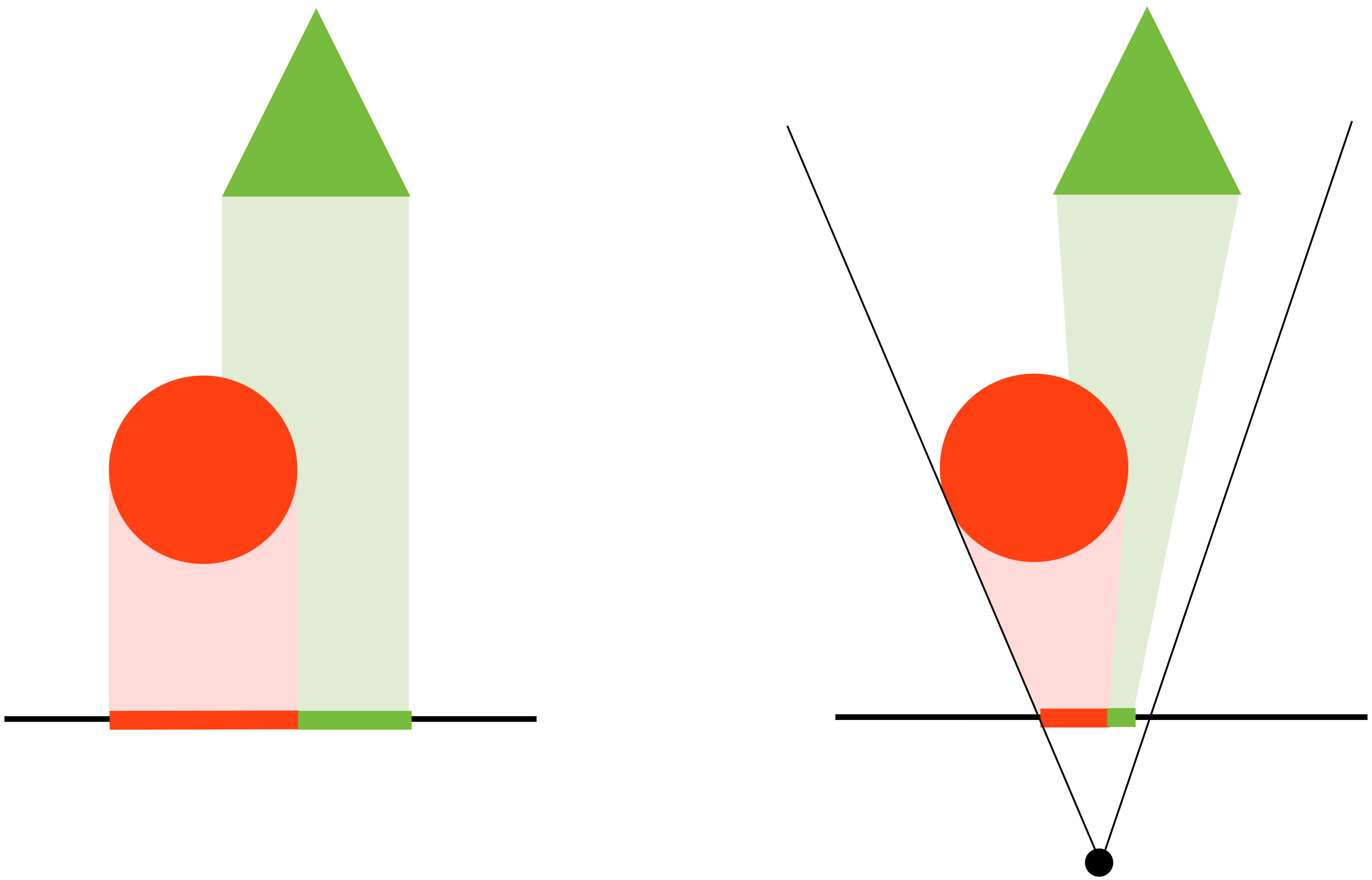
Orthographic Projection

- Drop one coordinate
  - Project onto *xy* plane: $(x,y,z) \rightarrow (x,y)$
  - Parallel lines remain parallel
  - In homogeneous coordinates:

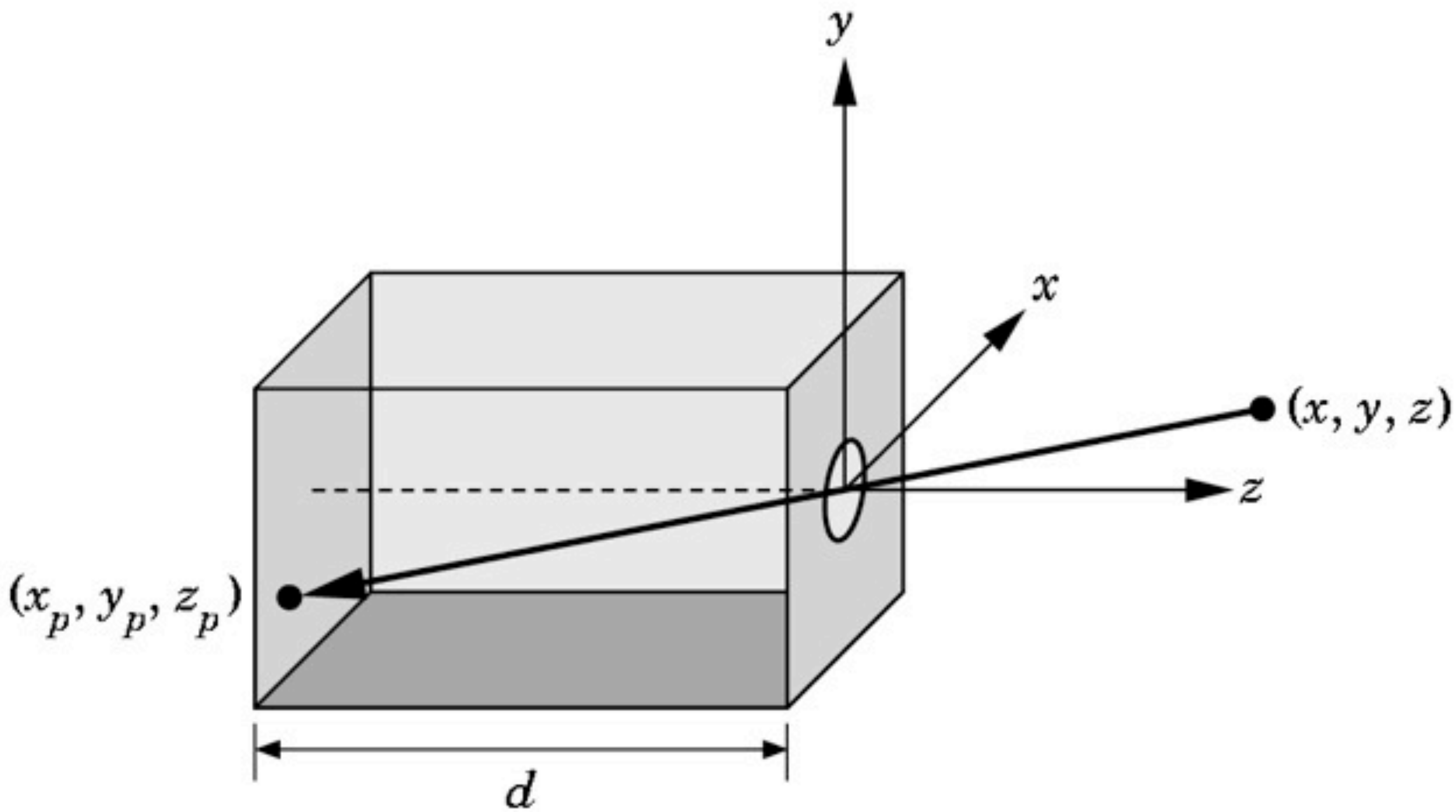


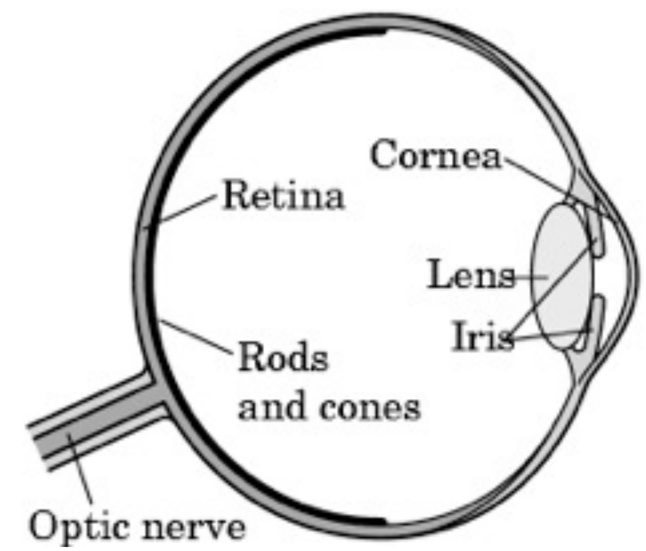
$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Orthographic vs Perspective



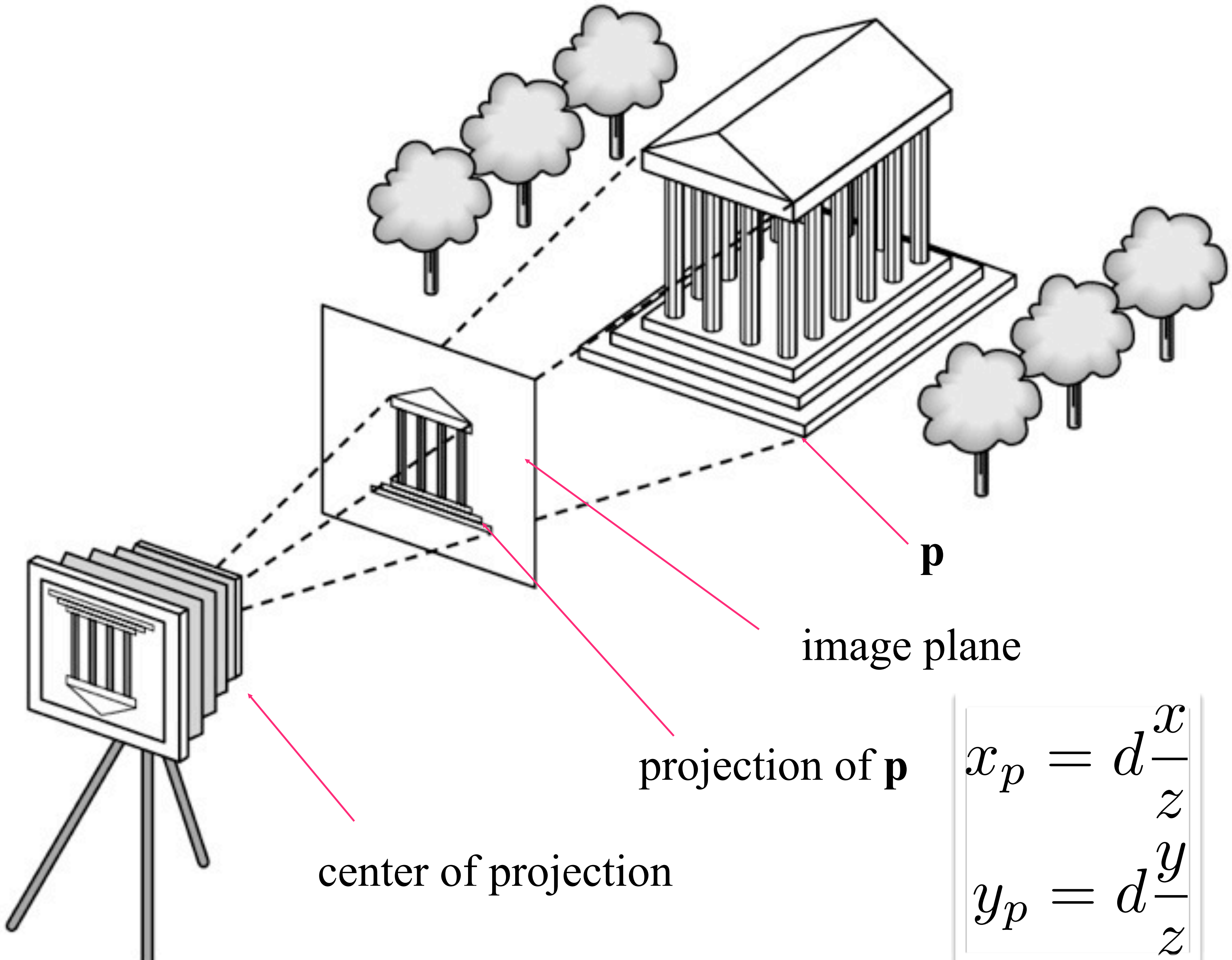
Pinhole Camera





- Projection of a 3D point (x,y,z) on image plane: $x_p = -d\frac{x}{z}, \quad y_p = -d\frac{y}{z}$

Synthetic Camera Model



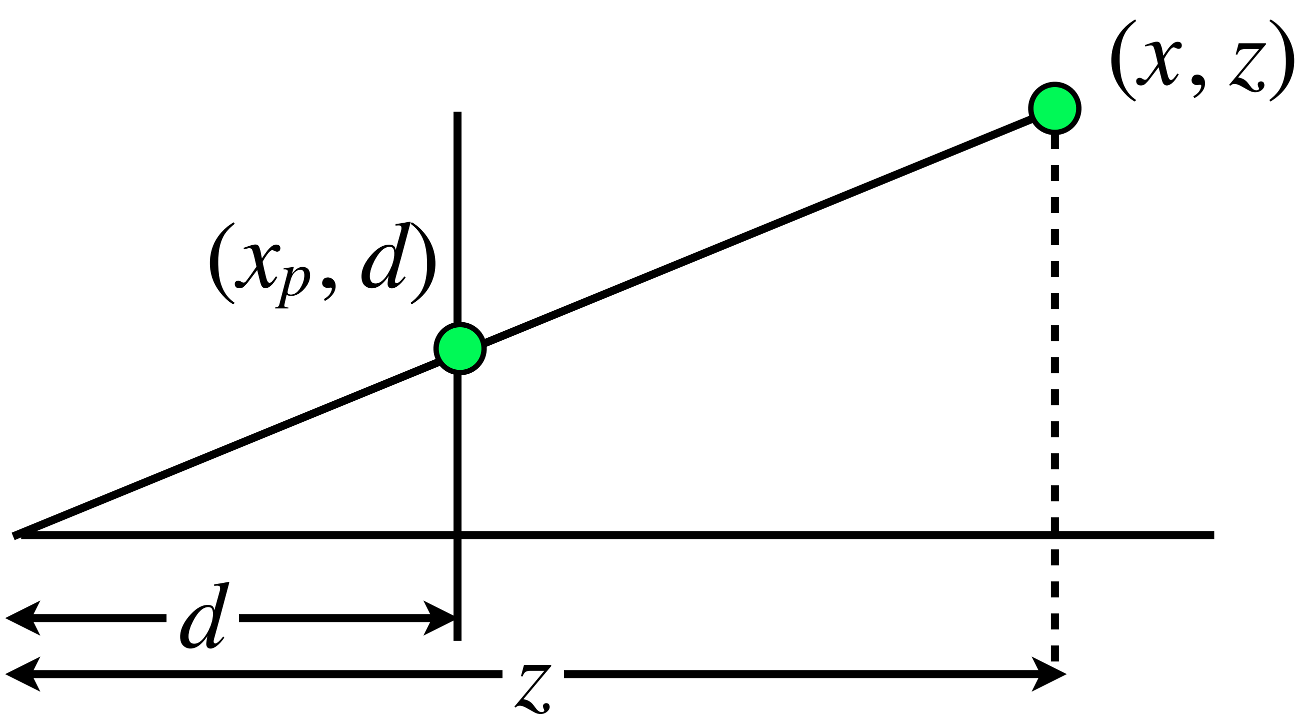
$$x_p = d\frac{x}{z}$$

$$y_p = d\frac{y}{z}$$

Perspective Projection

- More realistic model - objects far away are smaller after projection

$$(x, y, z) \rightarrow (d\frac{x}{z}, d\frac{y}{z})$$



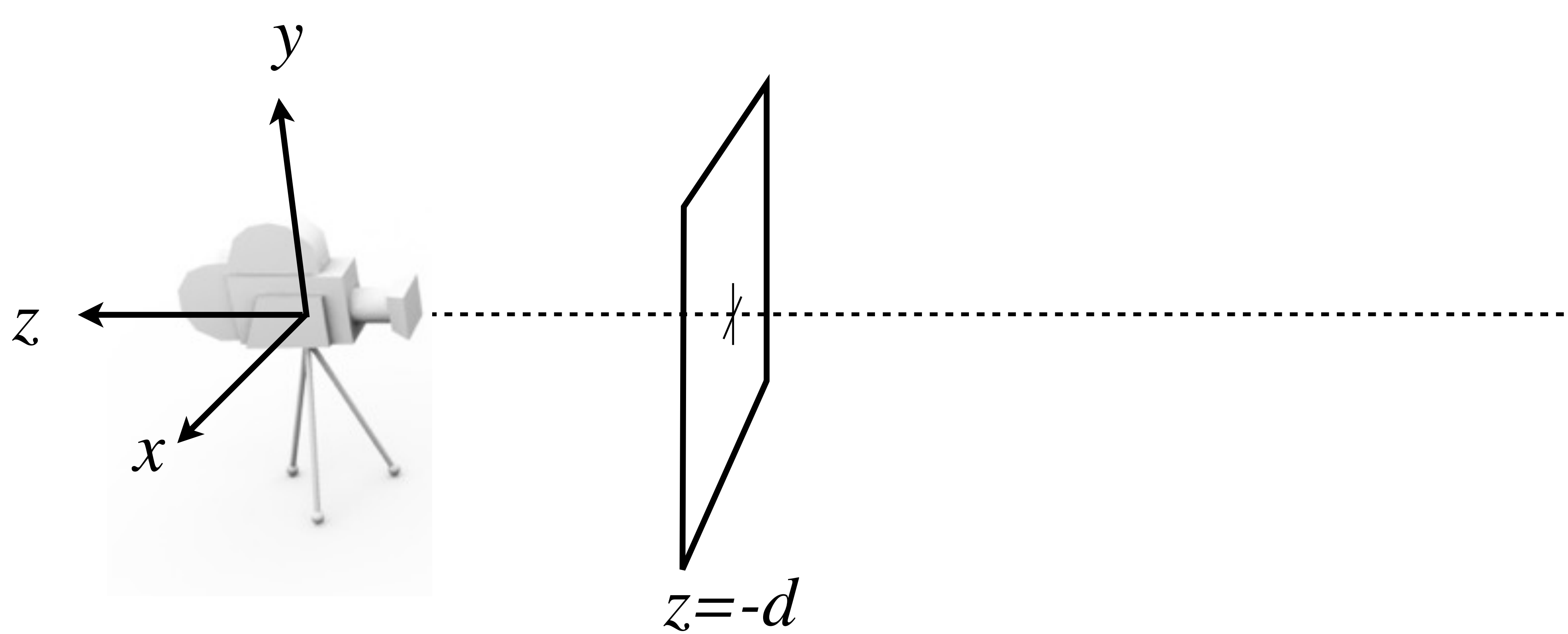
Equal triangles

$$\frac{x_p}{d} = \frac{x}{z}$$

$$x_p = d\frac{x}{z}$$

OpenGL convention

- In OpenGL: right-hand coordinate system, looking down $-z$.
  - The image plane is placed at $z = -d$
  - Visible geometry has negative z-values



Homogeneous Coordinates

- Homogeneous coordinates only defined up to scale

$$(wx, wy, wz, w) = (x, y, z, 1)$$

- Normalize by dividing with w
- Vector: $\mathbf{v} = (x, y, z, 0)$
  - "Point at infinity", pure direction
- Exploit this representation to express projection

Perspective Projection on Matrix Form

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{bmatrix}$$

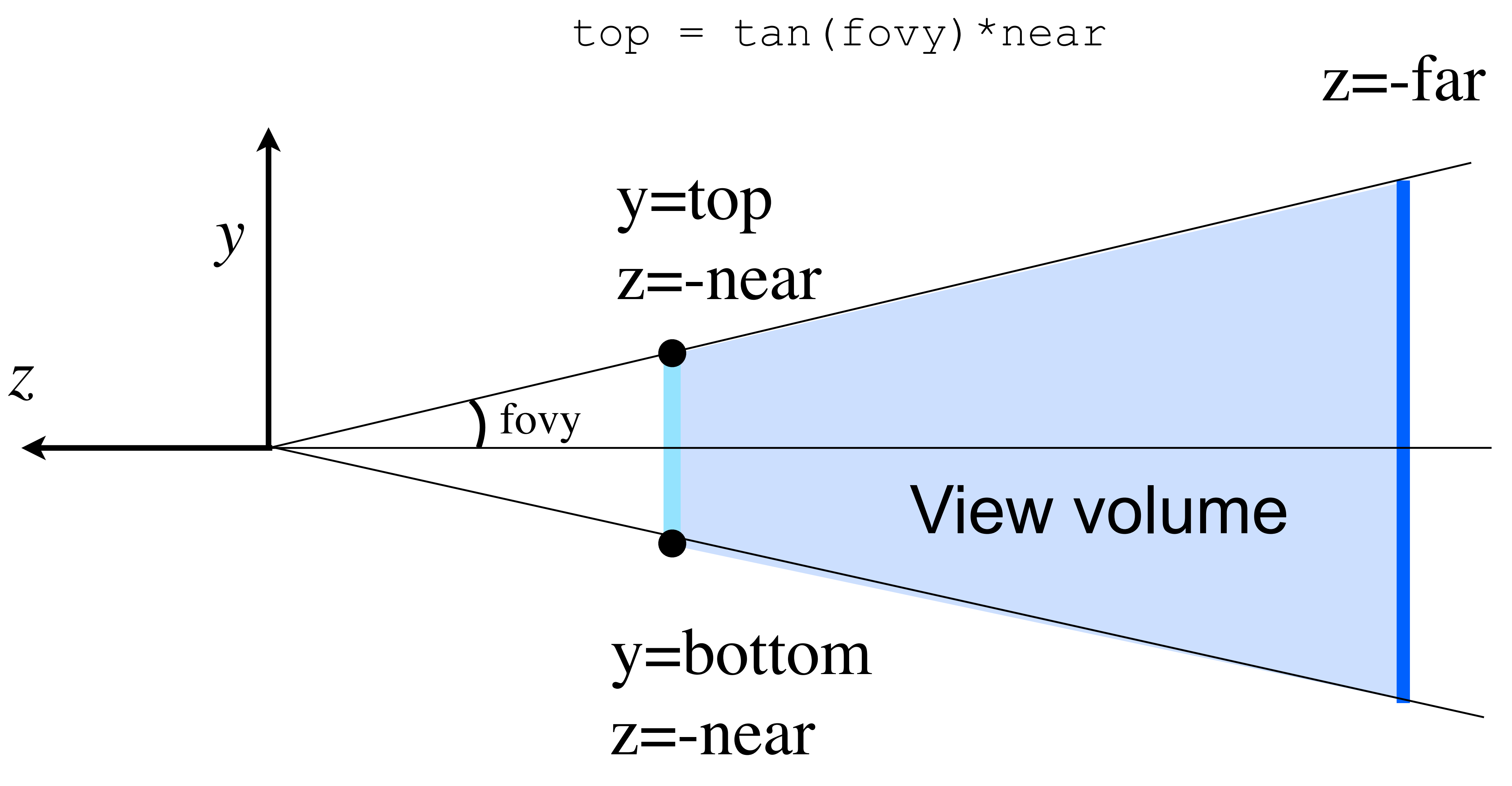
$$\begin{bmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{bmatrix} \rightarrow \begin{bmatrix} -\frac{dx}{z} \\ -\frac{dy}{z} \\ -d \\ 1 \end{bmatrix}$$

projected point on image plane $z = -d$

Divide by: $-\frac{z}{d}$

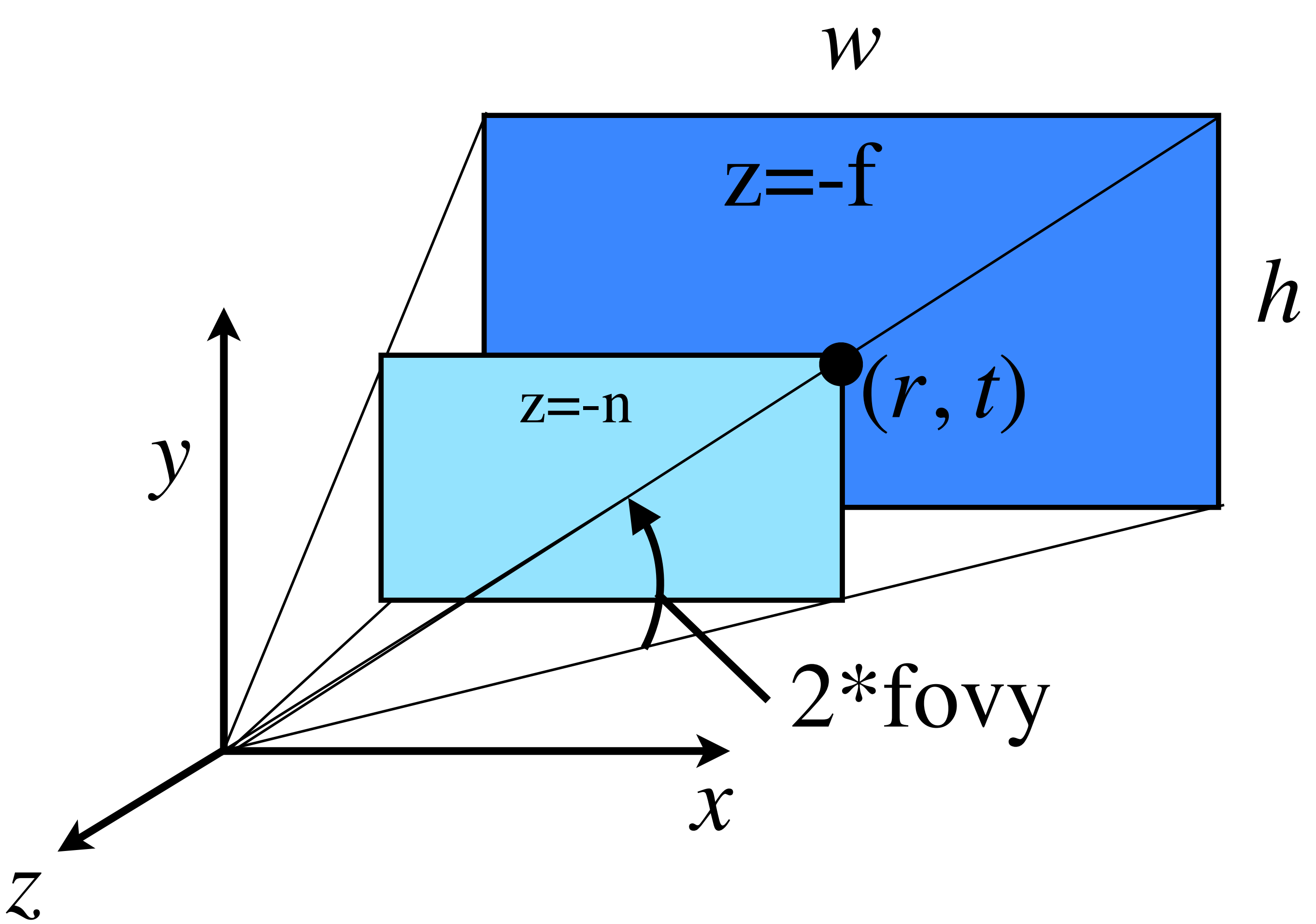
Camera in OpenGL

- Perspective camera setup



OpenGL Projection Matrix

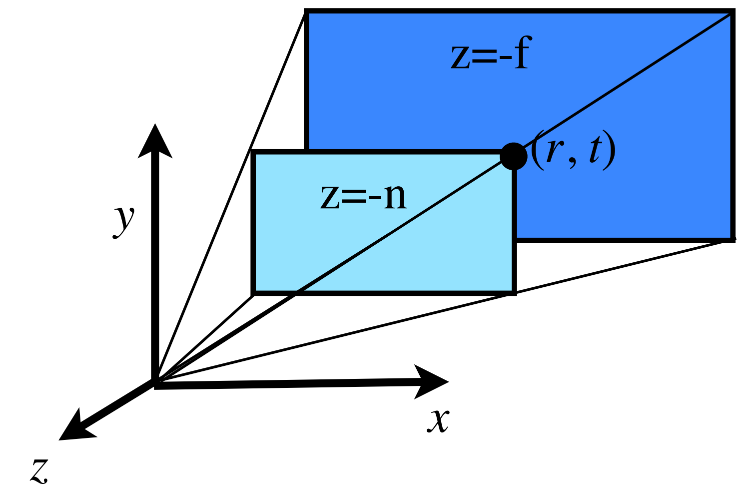
```
mat4 proj = Perspective(fovy,aspect,n,f);
```




```
aspect = w/h
t = tan(fovy)*n
r = t*aspect
```

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Examples



Point at upper right corner at near plane

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} r \\ t \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} n \\ n \\ -n \\ n \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

divide by w

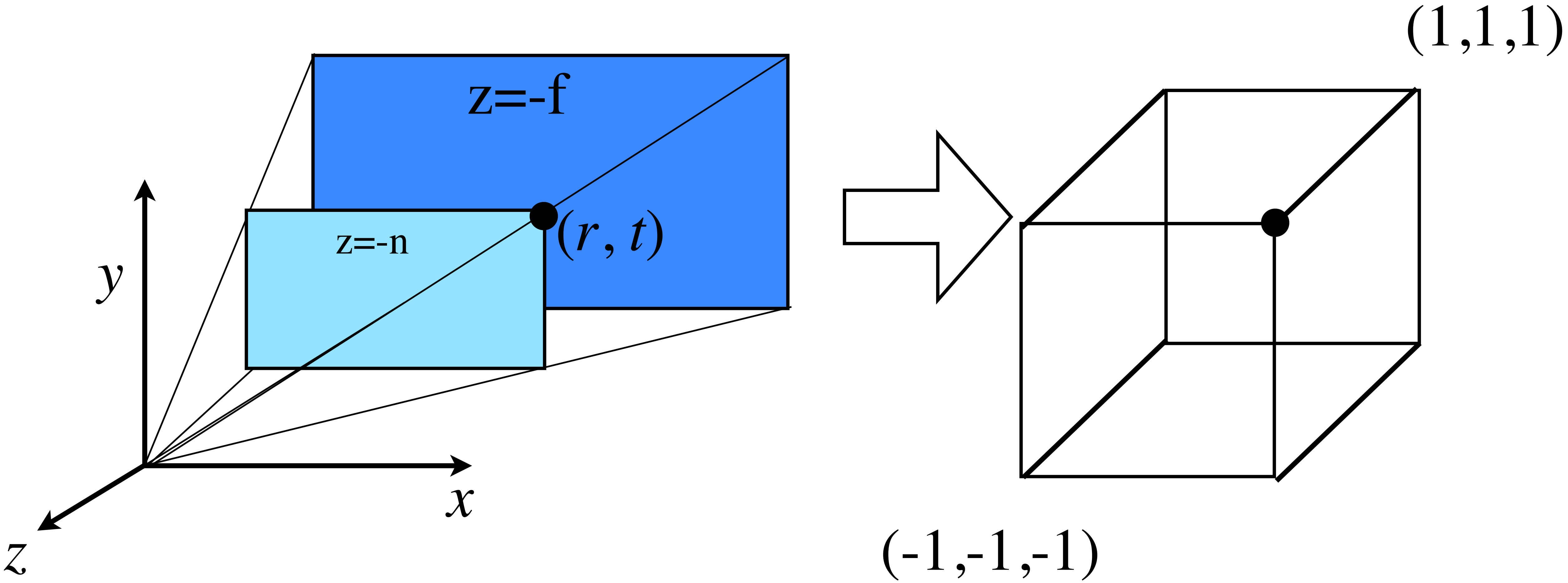
Point along view direction (-z) at far plane

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ f \\ f \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

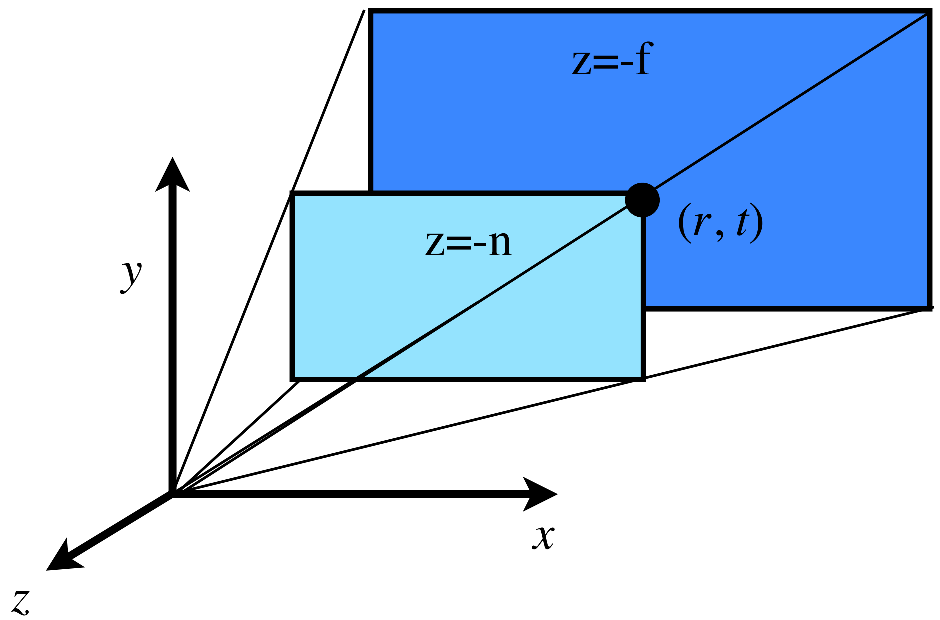
divide by w

OpenGL Projection Matrix

- View frustum volume maps to a cube



New Coordinate Spaces



divide by w

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \\ 1 \end{bmatrix}$$

Projection Matrix | Camera space | Clip space | NDC

Normalized Device Coords

Classification of Transforms

Translation

Rotation

Rigid Body preserves angles and distances

Uniform Scaling

Similarity preserves angles

Non-Uniform Scaling

Shear

Reflection

Affine preserves parallel lines

Perspective

Projective preserves lines

OpenGL

What is OpenGL?

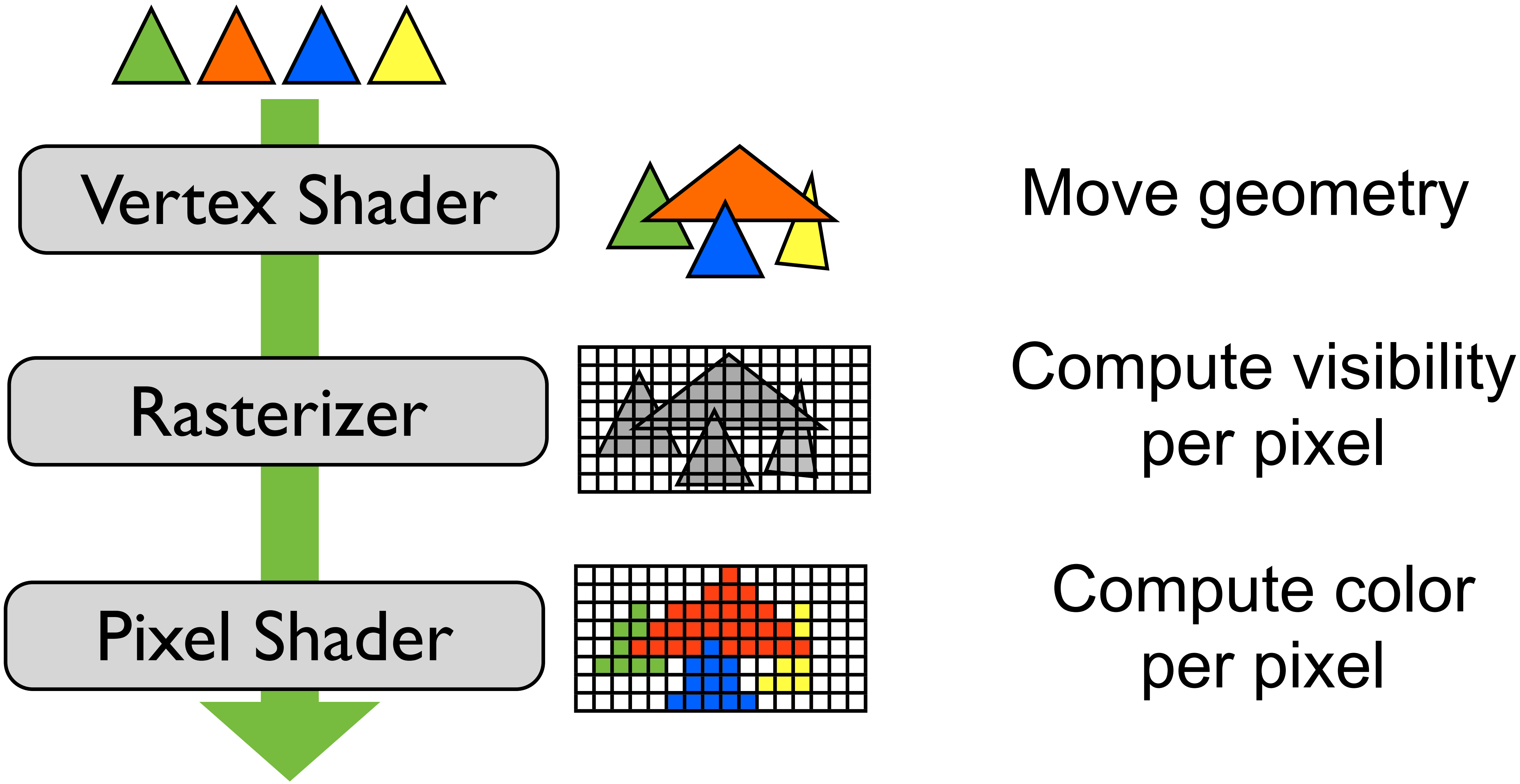
- OpenGL is a computer graphics rendering API
- High level API to graphics hardware
- Abstracts the graphics pipeline
- State machine
  - Input can either change the state or produce visible output

Flavors of GL

- OpenGL
  - Desktop & laptop
- OpenGL ES
  - Phones & Tablets
  - Focus on energy efficiency (heat, battery life)
- WebGL
  - JavaScript implementation of OpenGL ES
  - Works in modern web-browsers

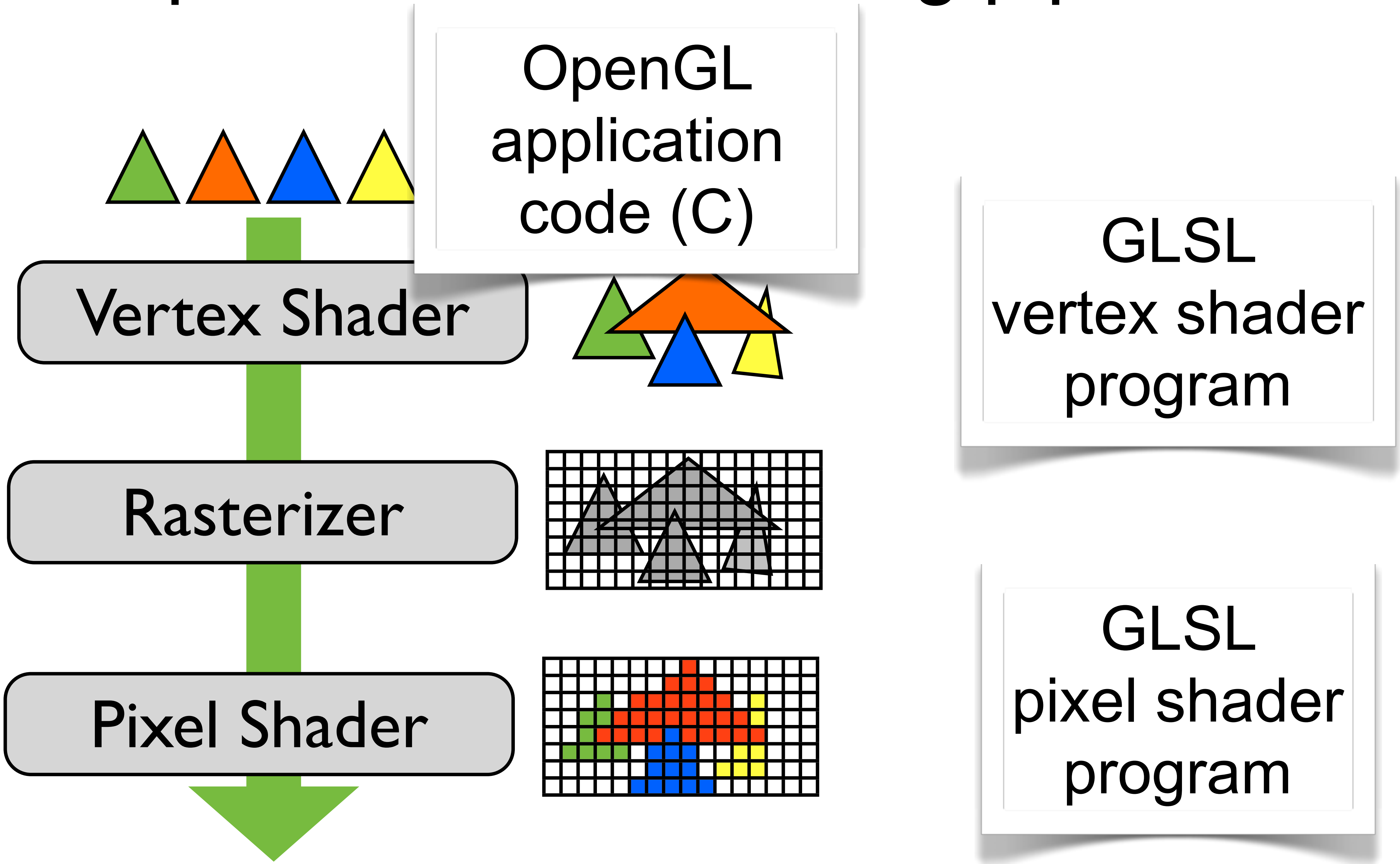
Graphics Hardware

- Pipeline that accelerates the costly tasks of rendering



Graphics Hardware

- Expresses the rendering pipeline



OpenGL Programming

- Create shaders
- Create buffers and load data into them
  - Vertices, normals, transformation matrices, textures
- Connect buffers with shader variables
- Render to an on-screen window
  - Platform specific, or use GLUT

Platform Independence

- Avoid OS window specific code
  - OpenGL render into a window, need to communicate with native windowing system
  - **GLUT**: open source lib for windowing ops.
- Different linkage mechanisms between OS:es
  - Library functions may look different on different platforms
  - **GLEW**: open source library hides this

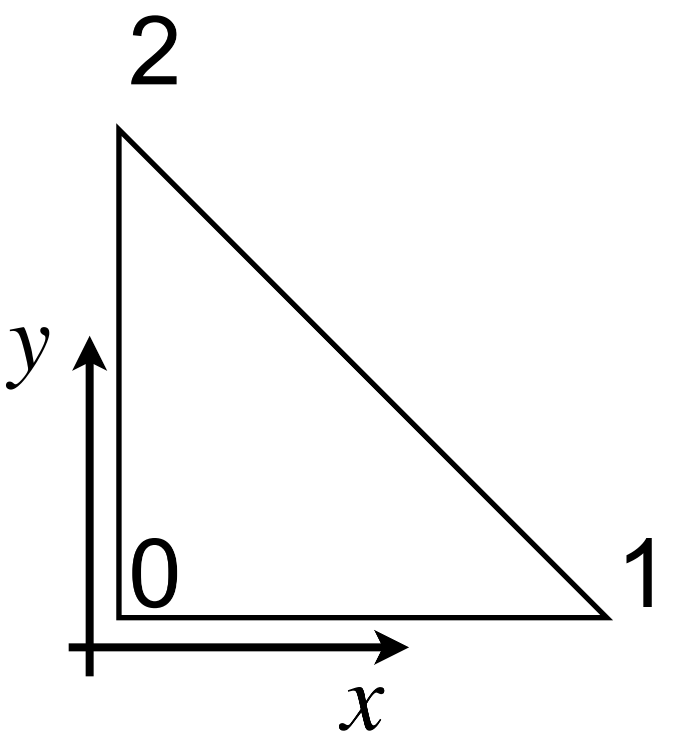
Example setup code

- Render a triangle
  - Create triangle vertices
  - Create Vertex Buffer Object to hold vertex data
  - Write shaders that:
    - Positions the triangle (vertex shader)
    - Color the triangle (pixel shader)

Create Geometry

```
struct vec4
{
    float x;
    float y;
    float z;
    float w;
};

vec4 points[3] =
{
    vec4(0,0,0,1), // vec 0
    vec4(1,0,0,1), // vec 1
    vec4(0,1,0,1), // vec 2
};
```



Vertex Array Object (VAO)

- Store all data of a geometric object
  - Holds one or more buffers describing the object

```
GLuint vaoID;
gGenVertexArrays(1, &vaoID); // generate vertex array
                             // object name(s)

glBindVertexArray(vaoID);    // bind a specific vertex array
                             // i.e., render a specific object
```

Vertex Buffer Object

- Store vertex data in GL buffers

```
GLuint bufferID;
gGenBuffers(1, &bufferID); // generate vertex array
                           // object name(s)

glBindBuffer(GL_ARRAY_BUFFER, bufferID);    // vertex array

glBufferData(GL_ARRAY_BUFFER, sizeof(points),
             points, GL_STATIC_DRAW);         // fill buffer
                                              // with data
```

```
vec4 points[3] =
{
    vec4(0,0,0,1),
    vec4(1,0,0,1),
    vec4(0,1,0,1),
};
```

Connect vertex array to shader

- Application vertex data enters GL pipeline through vertex shader
  - Load shaders
  - Get entry point in shader
  - Set pointer to uploaded buffer data

```
in vec4 vPosition;
uniform mat4 MVP; //ModelViewProj

void main()
{
   gl_Position = MVP * vPosition;
}
```

```
GLuint posID = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( posID );
glVertexAttribPointer( posID, 4, GL_FLOAT, GL_FALSE, 0,0);
```

Passing variables to shaders

- Need to associate a shader variable with an OpenGL data source
  - Example:

```
GLint  mat_idx = glGetUniformLocation( program, ”MVP” );
GLfloat mvp[16] = ...
glUniformMatrix4fv(mat_idx, 1, GL_TRUE, mvp);
```

```
in vec4 vPosition;
uniform mat4 MVP; //ModelViewProj

void main()
{
  gl_Position = MVP * vPosition;
}
```

Draw Geometry

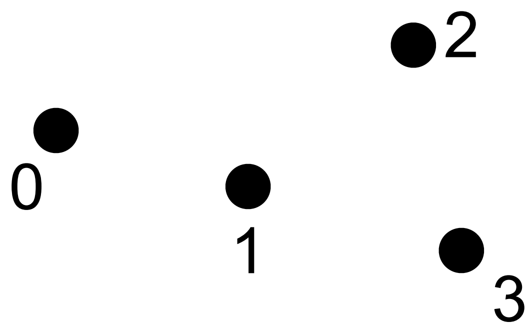
- Tell OpenGL to start rendering the triangle

number of indices

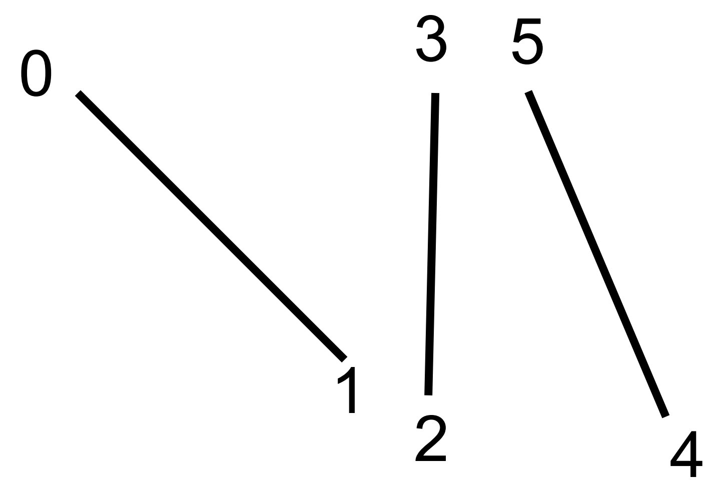
primitive type start index

```
glDrawArrays( GL_TRIANGLES, 0, 3 );
```

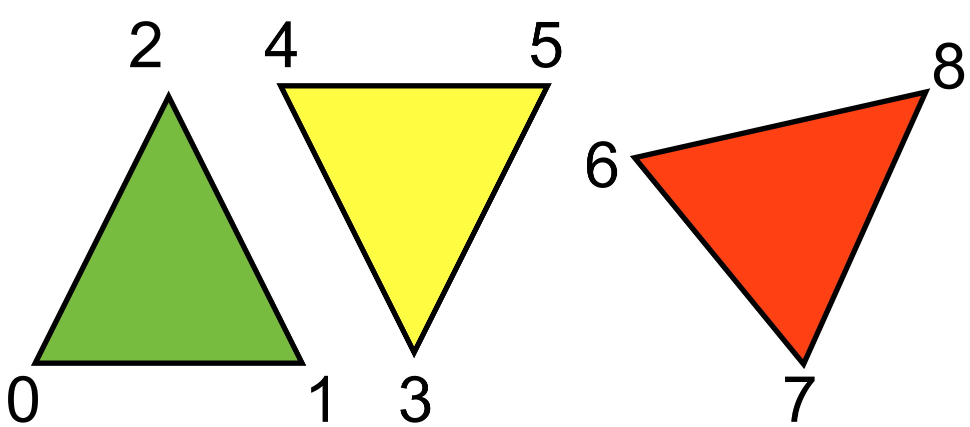
Build primitives from vertices



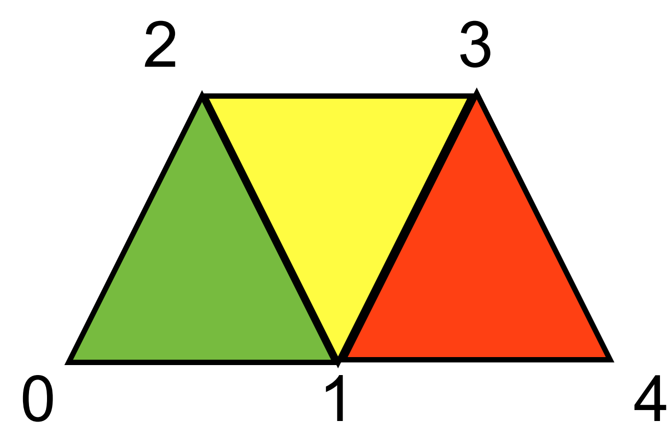
GL_POINTS



GL_LINES



GL_TRIANGLES



GL_TRIANGLE_STRIP

Transforms

- User constructs matrices in application code
- Pass them to the vertex shader
  - ```
    GLint mat_idx = glGetUniformLocation( program, "MVP" );
    GLfloat mvp[16] = ...
    glUniformMatrix4fv(mat_idx, 1, GL_TRUE, mvp);
    ```
- Vertex shader applies matrices
  - Last matrix applied is the projection matrix
  - Output is a clip space position
    ```

User input

- Callbacks for mouse and keyboard
  - See Assignment 4 for example code
  - Feel free to modify/add additional controls
  - GLUT has convenient callbacks for keyboard & mouse interactions

Getting shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program that runs on the graphics card

Shader Setup

```
GLuint initShaders()
{
   GLuint vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
   GLuint pixelShaderID = glCreateShader(GL_FRAGMENT_SHADER);

   const char* vs = textFileRead("simple.vert");
   const char* fs = textFileRead("simple.frag");
   glShaderSource(vertexShaderID, 1, &vs,NULL);
   glShaderSource(pixelShaderID, 1, &fs,NULL);
   delete [] vs;
   delete [] fs;

   glCompileShader(vertexShaderID);
   glCompileShader(pixelShaderID);

   GLuint program = glCreateProgram();
   glAttachShader(program,pixelShaderID);
   glAttachShader(program,vertexShaderID);

   glLinkProgram(program);
   glUseProgram(program);

   return program;
}
```

A complete OpenGL program in GLUT

Main

```
int main(int argc, char **argv)
{
   GLenum err = glewInit();  // Init GLEW
   if (GLEW_VERSION_3_0) { printf("GL version 3 supported \n"); }

   glutInit(&argc, argv); // Init GLUT
   glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
   glutInitWindowPosition(100,100);
   glutInitWindowSize(512,512);
   glutCreateWindow("GLSL Test");

   // Set GLUT callbacks
   glutDisplayFunc(render);
   glutIdleFunc(render);
   glutReshapeFunc(resize);
   glutKeyboardFunc(processKeys);
   glutMouseFunc(processMouse);
   glutMotionFunc(processMouseActiveMotion);

   init();       // Create geometry and shaders
   glutMainLoop();
   cleanup();

   return 0;
}
```

Init - Setup Geometry

```
void init()
{
    glClearColor(1.0,1.0,1.0,1.0);
    // Initalize shaders
    gShaderProgramID  = initShaders();

    // Create geometry (one triangle)
    vec3 vertices[3];
    vertices[0] = vec3( -0.5f, -0.5f, 1.0f);
    vertices[1] = vec3(  0.5f, -0.5f, 1.0f);
    vertices[2] = vec3( -0.5f,  0.5f, 1.0f);

    // Create a vertex array object
    glGenVertexArrays( 1, &gVaoID );
    glBindVertexArray( gVaoID );

    // Create and initialize a buffer object
    glGenBuffers( 1, &gVboID );
    glBindBuffer( GL_ARRAY_BUFFER, gVboID );
    glBufferData( GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW );

     // Initialize the vertex position attribute from the vertex shader
    GLuint pos = glGetAttribLocation( gShaderProgramID, "vPosition" );
    glEnableVertexAttribArray( pos );
    glVertexAttribPointer( pos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
}
```

Resize

```
// If the size of the window changed,
// call this to update the GL matrices
void resize(int w, int h)
{
    if(h == 0) h = 1; // Prevent a divide by zero

    // Calculate the projection matrix
    float aspect = ((float)w) / h;
    float fovy = 45.0f;
    float near = 0.01f;
    float far  = 10.0f;
    gProjectionMatrix = Perspective(fovy, aspect, near, far);

    glViewport(0, 0, w, h); // Set the viewport to be the entire window
}
```

Render

```
void render()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Calculate the view matrix
    vec3 at(0.0,0.0,0.0);
    vec3 up(0.0,1.0,0.0);
    mat4 View = LookAt(gEyePos, at, up );

    // Compute world matrix
    mat4 World = ...;

    // Compute ModelViewProjection matrix
    mat4 MVP = gProjectionMatrix*View*World;

    // Pass the modelview projection matrix to the shader
    GLuint mvpID = glGetUniformLocation(gShaderProgramID,"MVP");
    glUniformMatrix4fv(mvpID, 1, GL_TRUE, (GLfloat*)MVP.getFloatArray());

    // draw a triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glutSwapBuffers();
}
```

Input Handling

```
// Mouse and keyboard handling
void processKeys(unsigned char key, int x, int y)
{
  switch (key) {
     case 27:
        exit(0);
        break;
     case 'w': case 'W':
        gEyePos.z -= 0.1;
        break;
     case 's': case 'S':
        gEyePos.z += 0.1;
        break;
     default:
        break;
  }
}
```