3D Graphics Programming

T163 - Game Programming

Fall 2020

Week 2

VAOs/VBOs

Transformations & Animation

OpenGL Libraries

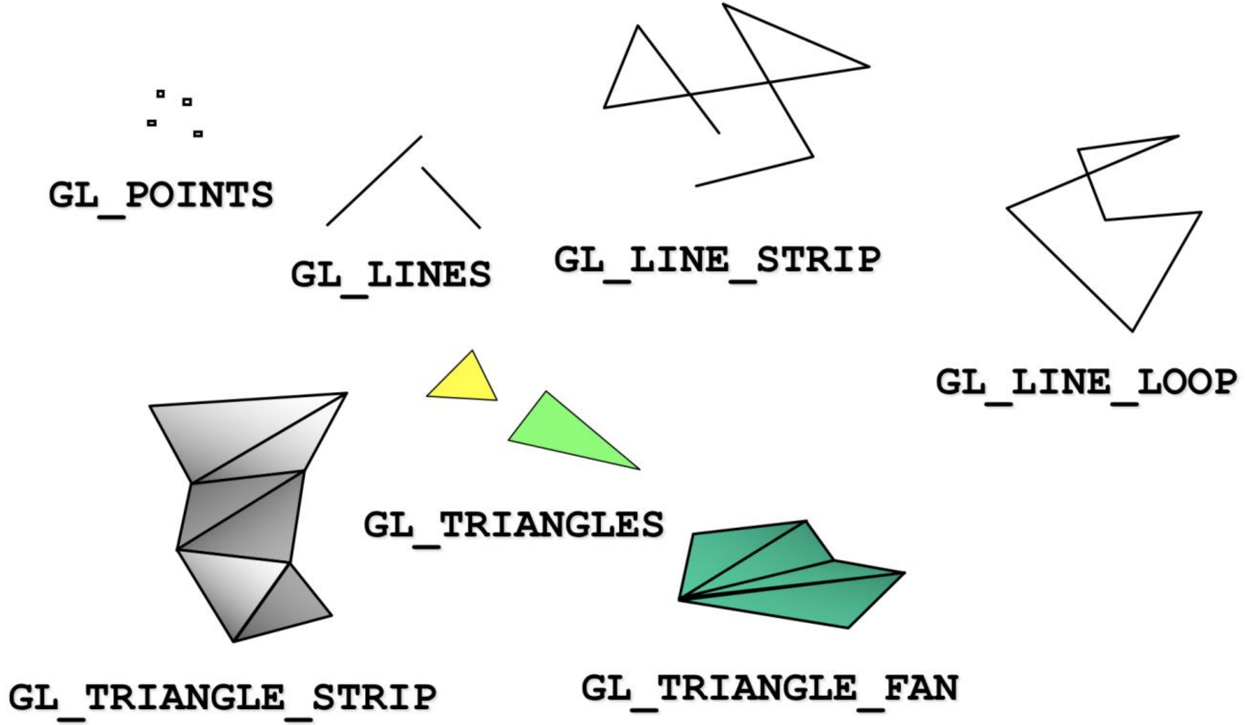
- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)

- GLEW - Open**GL E**xtension **W**rangler Library

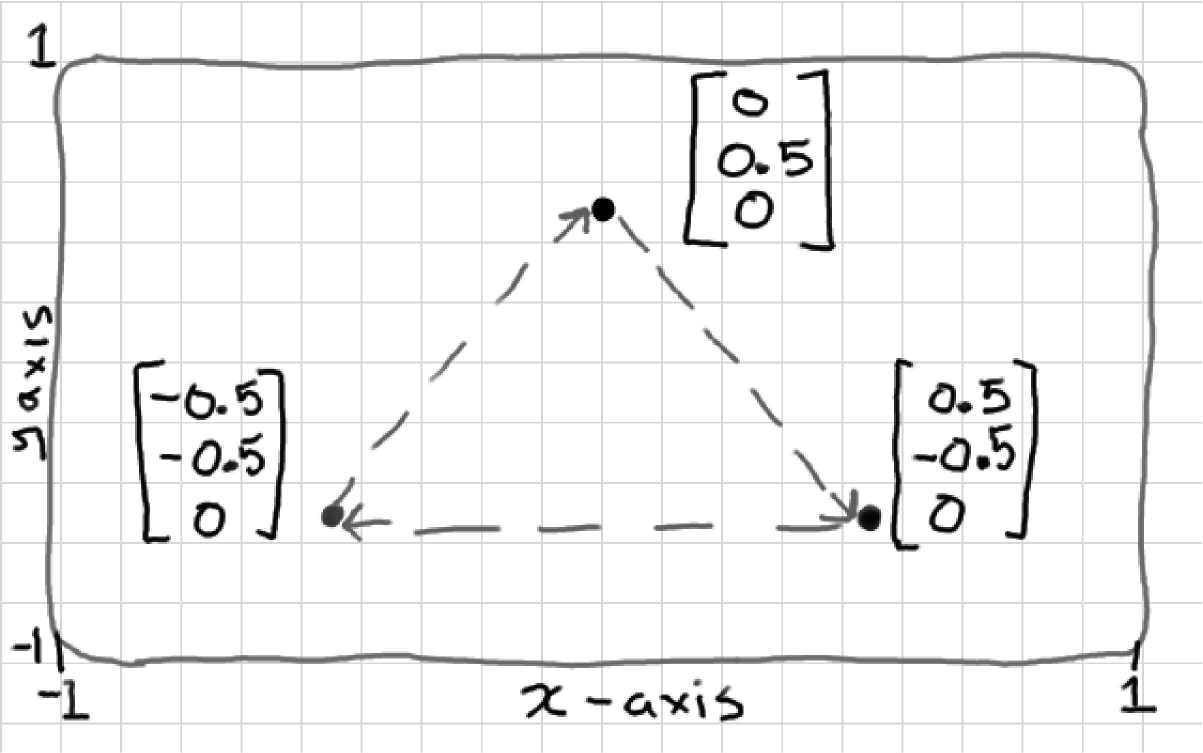
- GLUT - Open**GL U**tility **T**oolkit
  - Provides functionality common to all window systems
  - Open a window
  - Get input from mouse and keyboard

- GLM - Open**GL M**athematics Library

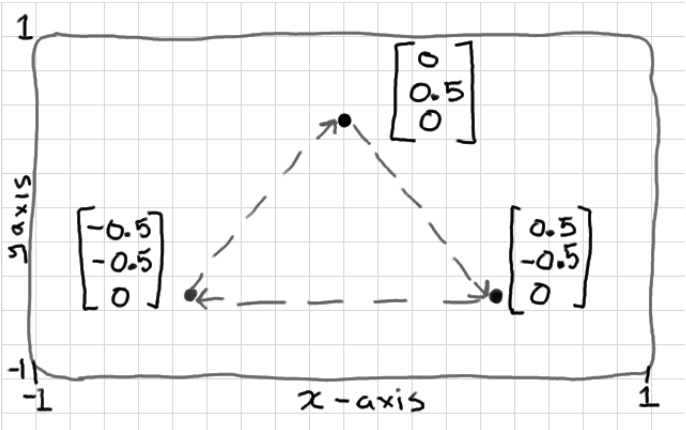
OpenGL Primitives



Describing Shapes



Describing Shapes




```
float points[] = {
    0.0f,  0.5f,  0.0f,
    0.5f, -0.5f,  0.0f,
   -0.5f, -0.5f,  0.0f
};
```

Adding Colors

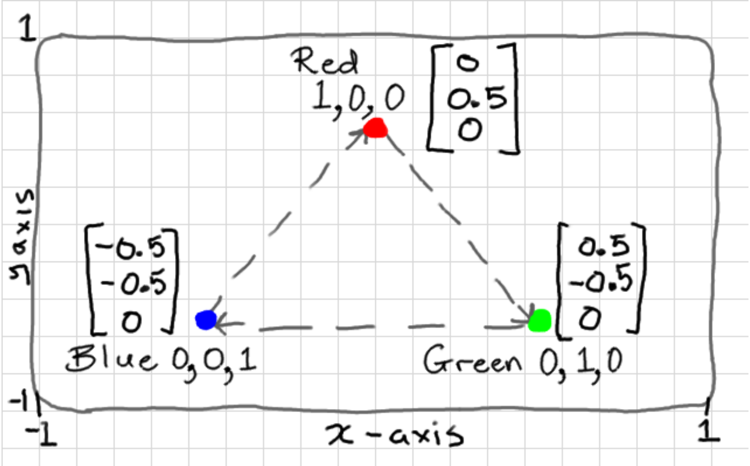
- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes

- Let's associate a color with each vertex

- Set up an array of same size as positions

- Send to GPU as a vertex buffer object

Adding Colors




```
float points[] = {
    0.0f,  0.5f,  0.0f,
    0.5f, -0.5f,  0.0f,
   -0.5f, -0.5f,  0.0f
};
```

```
float colours[] = {
  1.0f, 0.0f,  0.0f,
  0.0f, 1.0f,  0.0f,
  0.0f, 0.0f,  1.0f
};
```

Shader Basics

- Vertex Shader
  - The main purpose of a vertex shader is to transform points (x, y, and z coordinates) into different points
- Fragment Shader
  - The main purpose of a fragment shader is to calculate the color of each pixel that is drawn
- VAOs and VBO
  - Vertex Array Object / Vertex Buffer Object
  - VAOs and VBOs are used to take data from your C++ program and send it through to the shaders for rendering

VAOs

- VAOs are the link between the VBOs and the shader variables
- VAOs describe what type of data is contained within a VBO, and which shader variables the data should be sent to.

- One VAO contains the description of one object, which may consist of multiple VBOs

```
glGenVertexArrays(1, &gVAO);
glBindVertexArray(gVAO);
```

VAOs

- One VAO contains the description of one object, which may consist of multiple VBOs
- In our example we need to bind each VBO to our one VAO

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colours_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

VAOs

```
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

- The first parameter of glVertexAttribPointer() asks for an index
  - This is going to map to the indices in our vertex shader, so we need to give each attribute here a unique index
  - I will give my points index 0, and the colours index 1.
  - We will make these match up to the variables in our vertex shader later
  - If you accidentally leave both indices at 0 (easy enough to do when copy-pasting code), then your colours will be read from the position values so x→red y→green and z→blue

- Both buffers contain arrays of floating point values, hence GL_FLOAT, and each variable has 3 components each, hence the 3 in the second parameter
  - If you accidentally get this parameter wrong (quite a common mistake), then the vertex shaders will be be given variables made from the wrong components (e.g. position x,y,z gets values read from a y,z,x)

VAOs

- You only need to define the VAO once for each object, when creating the object
- There is no need to repeat this code inside the rendering loop
- Just keep track of the VAO index for each type of mesh that you create

VBOs

- VBOs are "buffers" of video memory – just a bunch of bytes containing any kind of binary data you want

```
glGenBuffers(1, &gVBO);
glBindBuffer(GL_ARRAY_BUFFER, gVBO);
```

VBOs

- To have colors we would need a VBO for the points and another for the colors

```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float),
                              points, GL_STATIC_DRAW);
```

```
GLuint colours_vbo = 0;
glGenBuffers(1, &colours_vbo);
glBindBuffer(GL_ARRAY_BUFFER, colours_vbo);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float),
                              colours, GL_STATIC_DRAW);
```

Note: We have omitted the glVertexAttribPointer methods to just show the two buffers

Filling the VBO

- In the glBindBuffer method, we send the array data to the buffer, like our points and colours below

```
float points[] = {
   0.0f,  0.5f,  0.0f,
   0.5f, -0.5f,  0.0f,
  -0.5f, -0.5f,  0.0f
};
```

```
float colours[] = {
  1.0f, 0.0f,  0.0f,
  0.0f, 1.0f,  0.0f,
  0.0f, 0.0f,  1.0f
};
```

VBO Final Step

- We created a vertex array object, and described two attribute "pointers" within it
  - Unfortunately, attributes are disabled by default in OpenGL
  - We use a function called glEnableVertexAttribArray() to enable each one

- This function only affects the currently bound vertex array object
  - This means that when we do this now, it will only affect our attributes, above
  - We will need to bind every new vertex array and repeat this procedure for those too

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Vertex Shader

- Now that we have our buffers set, we can send that data to the vertex shader
- Shaders have “inputs” and “outputs”

```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_colour;

out vec3 colour;

void main() {
  colour = vertex_colour;
  gl_Position = vec4(vertex_position, 1.0);
}
```

Fragment Shader

- The output of the vertex shader becomes an input for the fragment shader

```
in vec3 colour;
out vec4 frag_colour;

void main() {
  frag_colour = vec4(colour, 1.0);
}
```

Draw Loop

- We will have a display function in our OpenGL programs
- Serves as a render loop

```
glClear(GL_COLOR_BUFFER_BIT);
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Culling

- This gives a hint to GL so that it can throw away the hidden "back" or inside faces of a mesh
- This should remove half of the vertex shaders, and half of the fragment shader instances from the GPU - allowing you to render things twice as large in the same time
- It's not appropriate all of the time

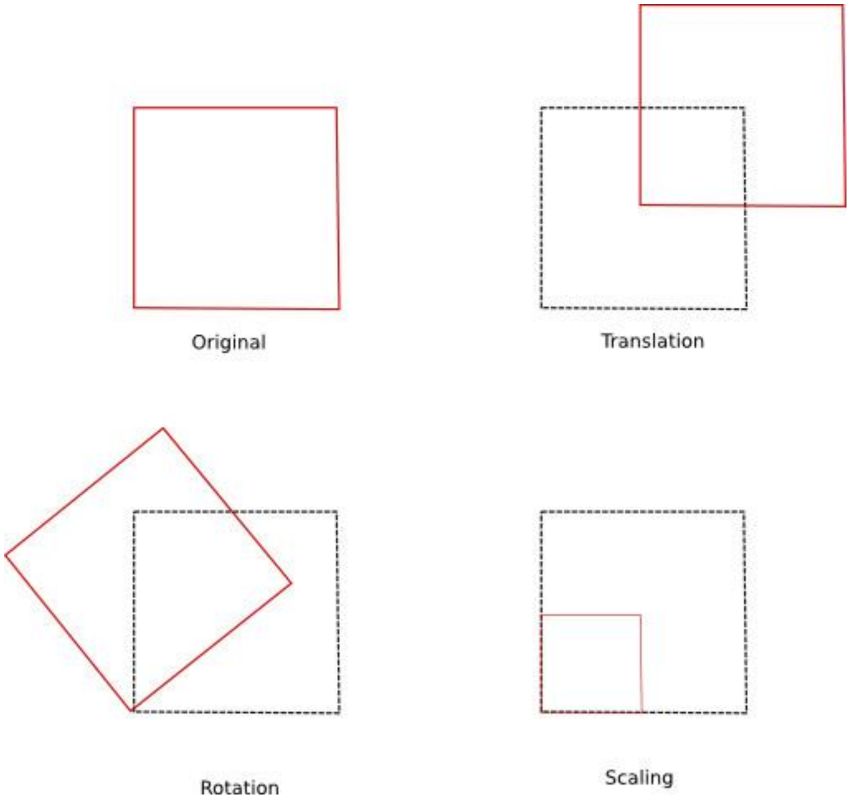
```
glEnable(GL_CULL_FACE); // cull face
glCullFace(GL_BACK); // cull back face
glFrontFace(GL_CW); // GL_CCW for counter clock-wise
```

Week 2

Transformations & Animation

Fall 2020

Transformations



Matrices

- Identity Matrix

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

- Scaling Matrix

```
glm::mat4 myScalingMatrix = glm::scale(glm::mat4(1.0f),glm::vec3(0.5f));
```

Matrices

- Applying a 45 degree rotation:

```
// Rotation around Oz with 45 degrees
glm::mat4 myRotationMatrix = glm::rotate(glm::mat4(1.0f),
glm::radians(45.0f), glm::vec3(1.0));
```

- Applying a translation:

```
// Translation
glm::mat4 TranslationMatrix = translate(mat4(), myTranslation);
```

Matrices

- Model Matrix
  - Combining it all together:

```
glm::mat4 ModelMatrix;
ModelMatrix = glm::mat4(1.0f)
ModelMatrix = glm::translate(ModelMatrix, translation);
ModelMatrix = glm::rotate(ModelMatrix, glm::radians(rotationAngle), rotationAxis);
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(scale));
```

  - Going to shader:

```
// Get a reference of the location in the shader
GLint model = glGetUniformLocation(shaderProgram, "Model" );
// Link the matrix with the shader uniform variable
glUniformMatrix4fv(model, 1, GL_FALSE, &ModelMatrix[0][0]);
```

Matrices

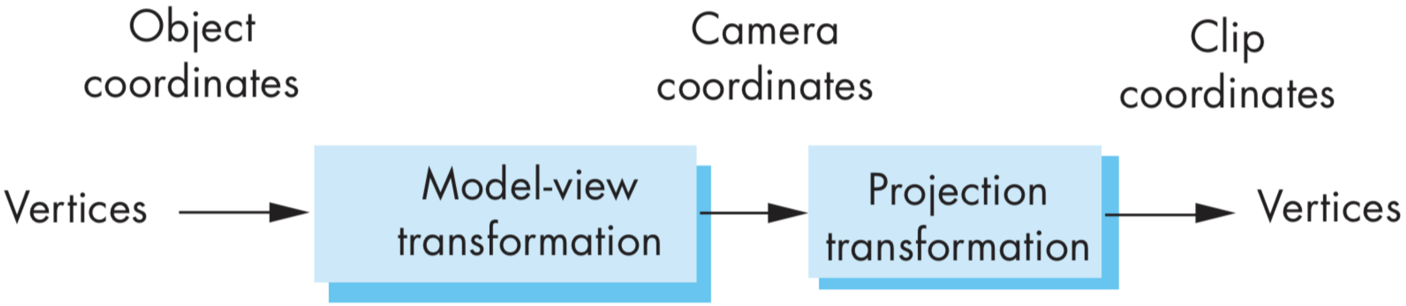
- Vertex Shader
  - Sending transformations as a uniform

```
in vec4 position;

uniform mat4 Model;

void main() {
  gl_Position = Model * position;
}
```

View Transformations



Model Matrix (Local to World)

- The matrix that contains every translations, rotations or scaling, applied to an object is named the **model matrix** in OpenGL

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

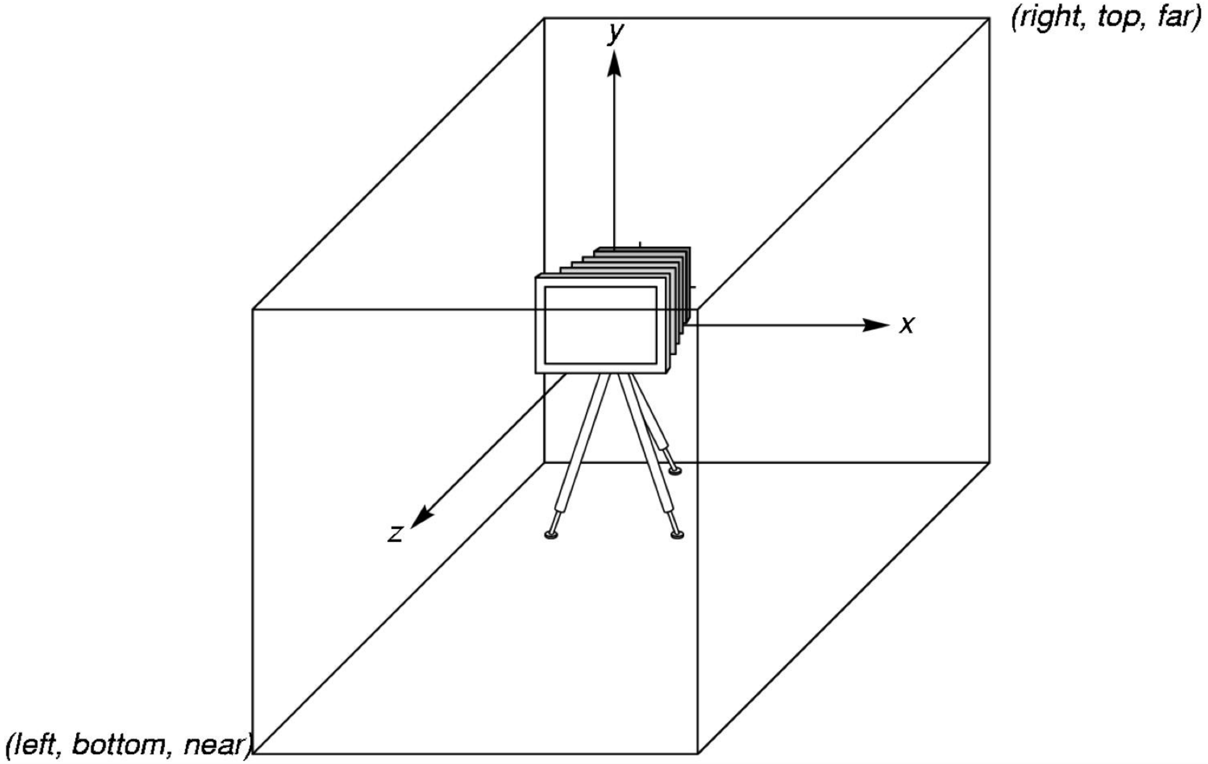
$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View Matrix (World to View)

- The view matrix in OpenGL controls the way we look at a scene
  - The eye, or the position of the viewer
  - The center, or the point where we the camera aims
  - The up, which defines the direction of the up for the viewer

- The defaults in OpenGL are: the eye at (0, 0, -1); the center at (0, 0, 0) and the up is given by the positive direction of the Oy axis (0, 1, 0).

OpenGL Camera



Projection Matrix (View to Screen)

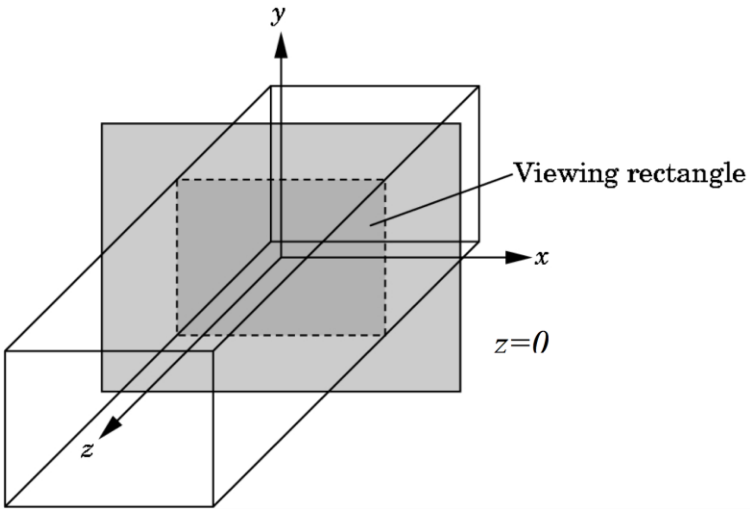
orthographic projection matrix

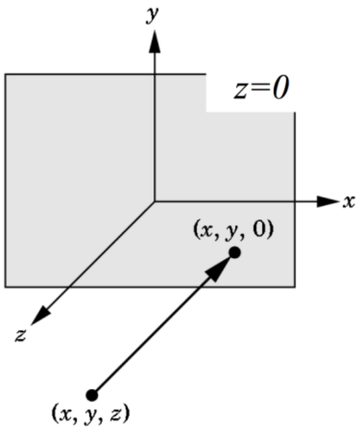
$$P = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

perspective projection matrix

$$P = \begin{bmatrix} \frac{2\cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2\cdot near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\cdot far\cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Orthographic View





Week 2

Lab Activities

Week 2 Lab

- For the lab, see Hooman's material (with video)
- OpenGL examples covered:
  - More fractal examples (cool factor over 9000!)
  - More shapes
  - Transforms

Week 2

End