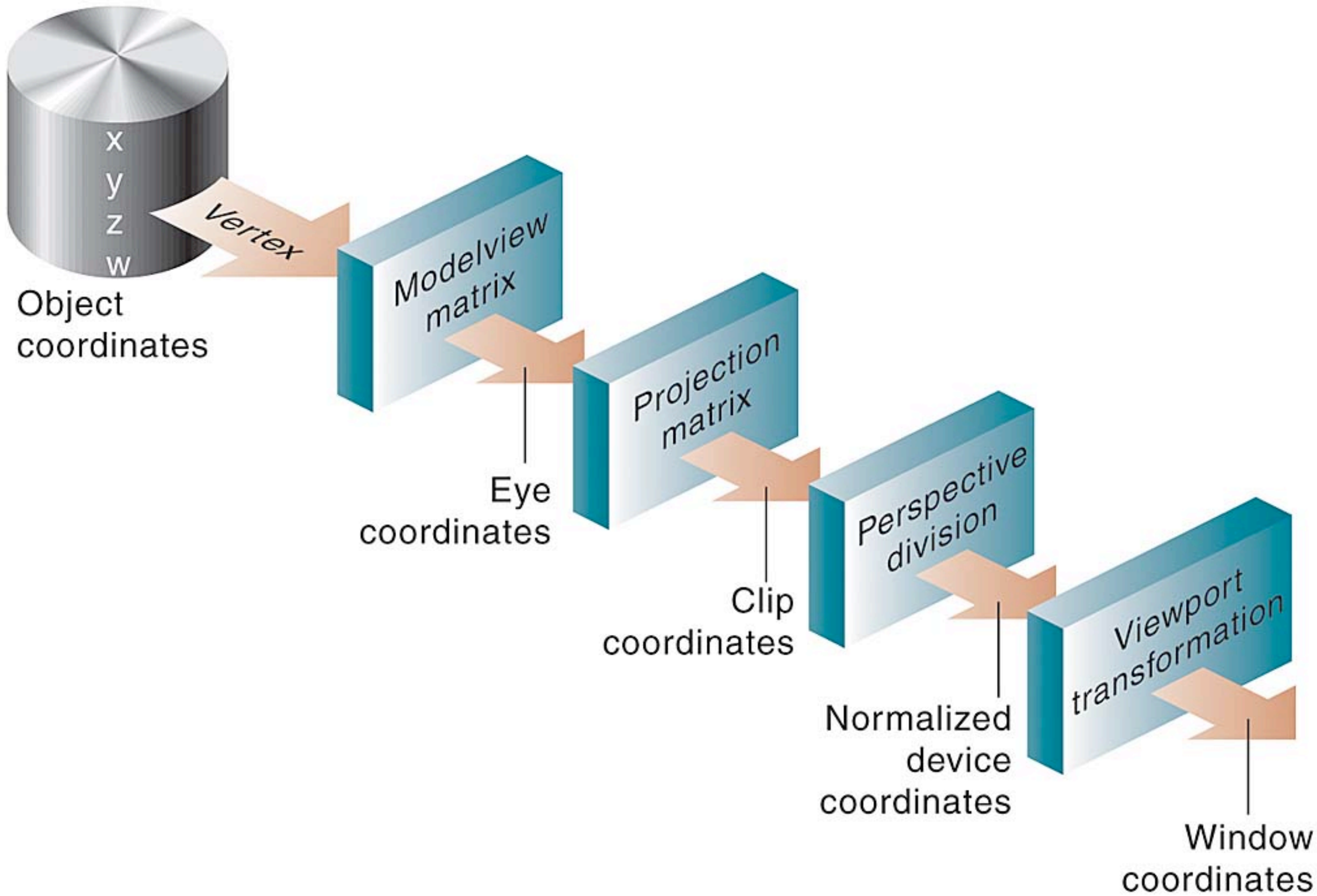
Overview

By end of the week:

- Know the basics of git
- Make sure we can all compile and run a C++/ OpenGL program
- Understand the OpenGL rendering pipeline
- Understand how matrices are used for geometric transformations
- Understand how the projection from 3D to 2D is encoded in a matrix
- Load and use an image texture

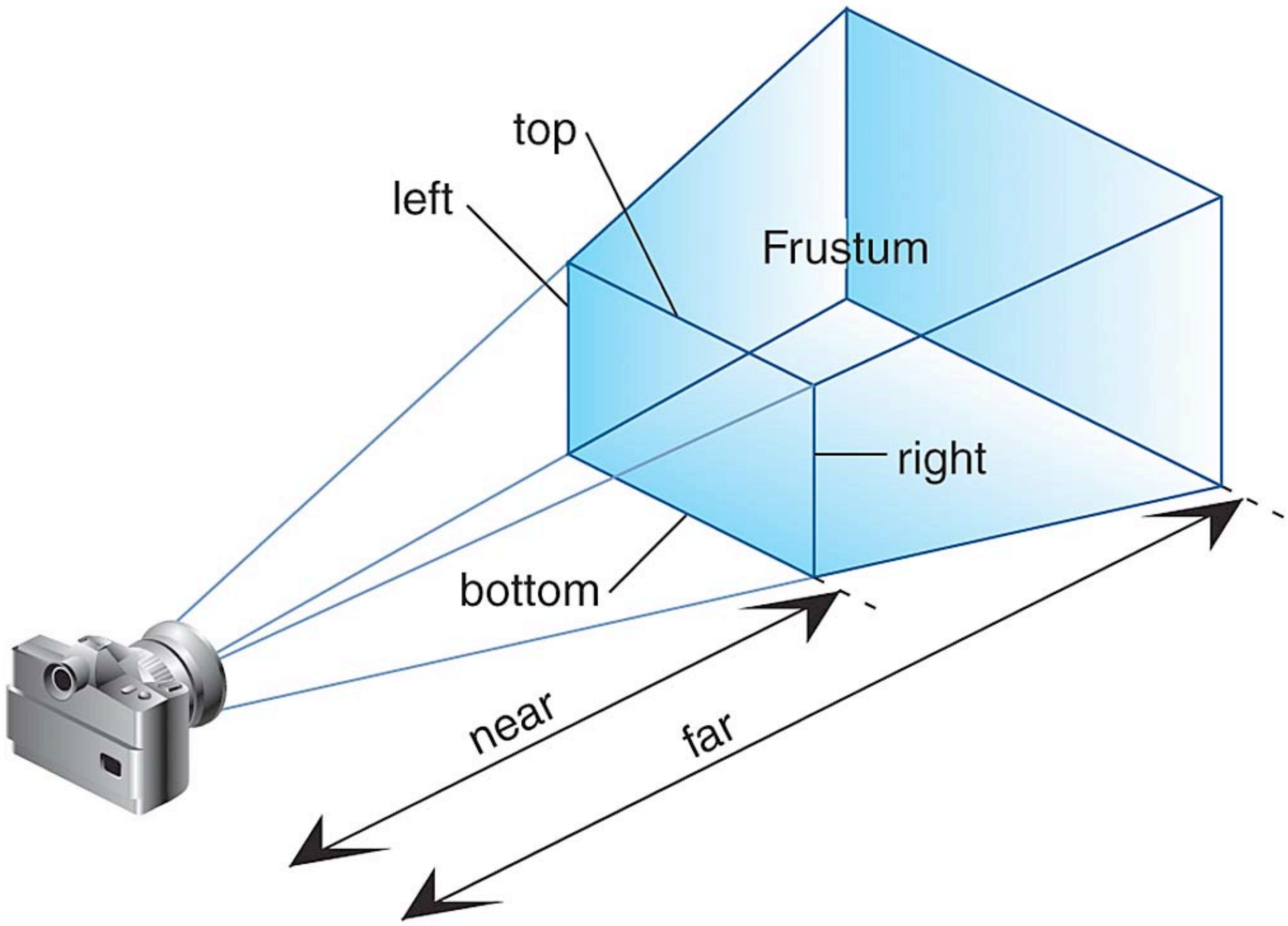
OpenGL – Vertex Transformation

Moving a point in 3D space to a 2D screen…



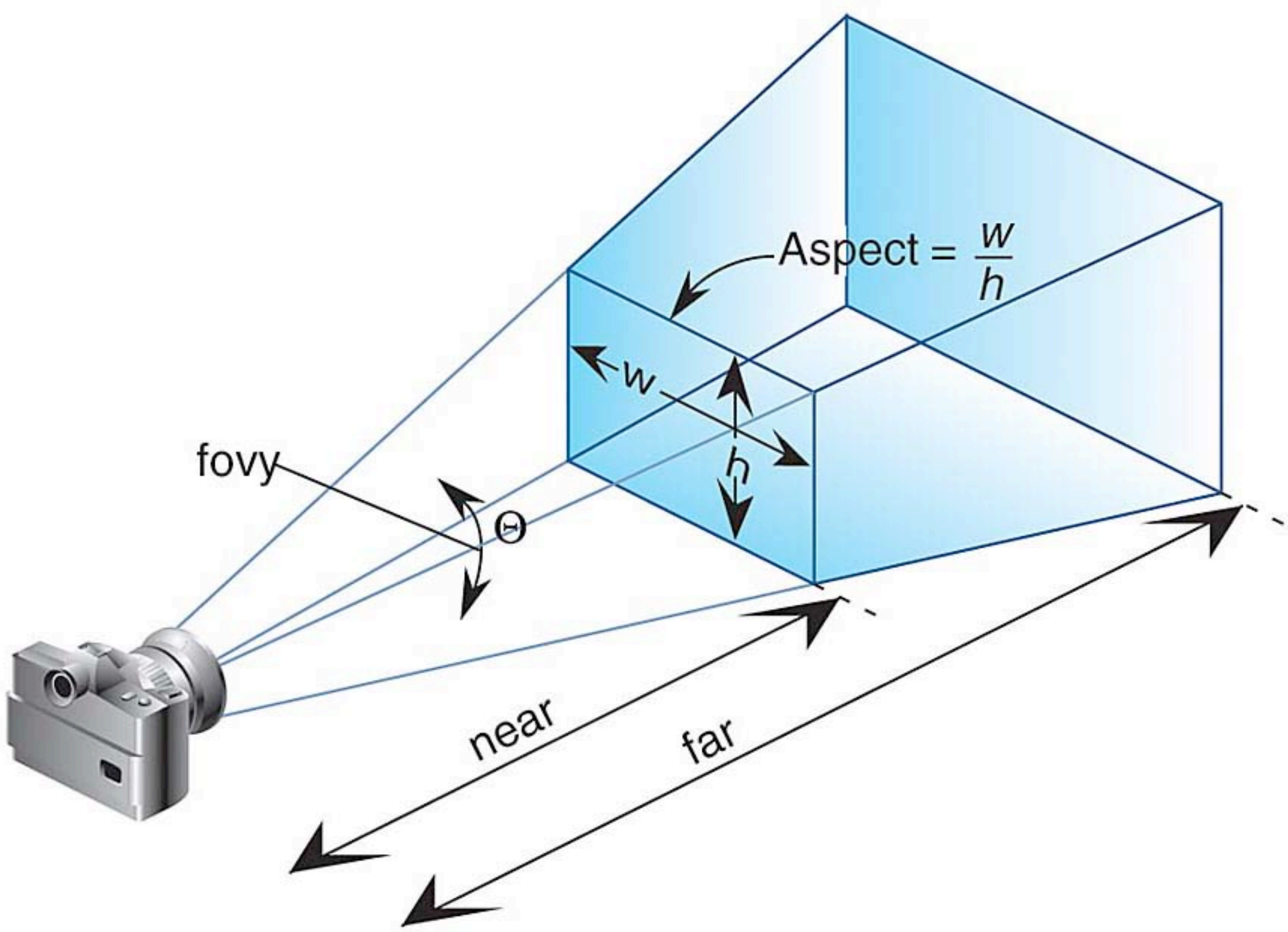
Clip Coordinates

The view frustum is defined from the point of view of the camera.



Clip Coordinates

Defining the view frustum using a perspective transformation.



Projection Matrix

The Projection Matrix defines how much of the world is seen by the camera. It encodes the following information:

The near plane and the far plane: The range of depth in the world that the camera can see.

The field of view angle that the camera sees in the y direction.

The aspect ratio of the screen which the world will be projected on.

Projection Matrix

The near plane and far plane define the distance along the z axis from the camera origin. The near plane needs to be a distance > 0 and the far plane needs to be < infinity. Common values are .1 and 100, but it depends on how you decide to position things in the world.

The field of view, or "fovy", defines the angle in the y direction

The aspect ratio (width/height) of the screen bounds thus defines the clipping in the x axis.

These values are used to define the view "frustum" in terms of 6 values, the left, right, top, bottom, near, and far bounds of the world.

The projection matrix transforms the view "frustum" into a unit cube.

Projection Matrix

The actual Projection Matrix looks likes this:

$$\begin{pmatrix} 2n / (r - l), & 0, & -(r + l) / (r - l), & 0 \\ 0 & 2n / (t - b), & (t + b) / (t - b), & 0 \\ 0 & 0, & -(f + n) / (f - n), & -(2fn) / (f - n) \\ 0 & 0, & 1, & 0 \end{pmatrix}$$

Where n and f are the near and far planes,

t and b are defined by the fovy

And l and r are further defined by the aspect ratio

Useful GLM methods

glm::mat4 proj = glm::**perspective**(60.0, width/height, 0.1, 100.0);

//creates a symmetrical perspective projection matrix

//arg 1,2,3,4 = fovy, aspect ratio, near plane, far plane

glm::vec3 camera_pos = vec3(0,0,-2);

glm::vec3 camera_look_at = vec3(0,0,0);

glm::vec3 camera_up = vec3(0,1,0);

glm::mat4 view = glm::**lookAt**(camera_pos , camera_look_at , camera_up);

//pos = position of camera in world space

//look_at = position camera is looking at; defines "view vector" emenating out from the camera

//up = the orientation of the camera around the view vector

Example: Transforming a vertex

To transform our 3D point from object coordinates into 2D window coordinates we do the following operations:

Given a vertex $\mathbf{v}_o$ in object coordinates (x_o, y_o, z_o, w_o), where w_o is always 1.

Put the object point into eye coordinates by multiplying it by the MODELVIEW matrix **M** (which concatenates the transformation from object coordinates → world coordinates → eye coordinates)…

$$\mathbf{v}_e = \mathbf{M}\mathbf{v}_o$$

Put the vertex into clip coordinates by multiplying it by the PROJECTION matrix P

$$\mathbf{v}_c = \mathbf{P}\mathbf{v}_e$$

Put the vertex into normalized device coordinates by dividing by the w_c value of $\mathbf{v}_c$.

$$\mathbf{v}_d = (x_c / w_c, y_c / w_c, z_c / w_c)$$

Put the vertex into screen space by scaling x_c and y_c by the width and height of the screen.

$$\mathbf{v}_p = (\text{width}/2 + (x_d * \text{width}/2), \text{height}/2 + (y_d * \text{height}/2))$$

Textures

Loading textures by hand is kind of a pain. OpenGL environments generally provide helper methods. We're using Cocoa/iOS methods (for Apple) or FreeImage (for Windows and Linux) which handles most of this.

A texture is just an array of data, can be used for images, depth maps, luminance maps, etc

1. enable textures and generate texture ids
2. bind a specific texture id
3. load image from disk
4. put it into a texture object – usually 2D, RGBA format
5. set texture attributes (eg, linear filtering, clamping)

Textures

Textures are copied directly onto the video card, so drawing them is "hardware-accelerated"

First, we call our helper method to load, say, a JPEG into a buffer of bytes, say a variable called "imgPixelData"

```
glEnable(GL_TEXTURE_2D);
glGenTextures(1, texID); //bind 1 textures to IDs
glBindTexture(GL_TEXTURE_2D, texID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(texID, 0, GL_RGBA, imgWidth, imgHeight, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, imgPixelData);
glBindTexture(GL_TEXTURE_2D, 0); //unbind texture
```

Textures - openGL

```
program.bind(); {
    //pass in uniform data ... one of which will be a pointer to a texture
    glUniform1i(program.uniform(”u_tex_id"), 0);

    glActiveTexture(GL_TEXTURE0) //the number here must match the ID above!

    glBindTexture(GL_TEXTURE_2D, texID) { //bind the texture
        //now pass in vertex data ...
        glBindVertexArray( vao ); {
            glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, 0);
        } glBindVertexArray( 0 );
    } glBindTexture(GL_TEXTURE_2D, 0); //unbind texture

} program.unbind();
```

Textures – vertex shader

```
uniform mat4 proj;
uniform mat4 view;
uniform mat4 model;
in vec4 vertexPosition;
in vec3 vertexTexCoord;
out vec2 texCoord;

void main() {
    texCoord = vertexTexCoord.xy;
    gl_Position = proj * view * model * vertexPosition;
}
```

Texture – fragment shader

```
uniform sampler2D u_tex_id;

in vec2 texCoord;
out vec4 outputFrag;

void main(){
  outputFrag = texture(u_tex_id, texCoord);
}
```

Homework package #1

I will send the first homework out tonight or tomorrow.

Will be due on Monday 9/15 in the evening (11:59pm).

1. A small sized programming project that makes use of the basic OpenGL / GLSL we've learned this week (and will cover next week)

2. Some smaller programming examples

3. A series of (hopefully) simple problem solving questions that you could do by hand

I'll announce details via Piazza...