CSE 167 Discussion #2

Nosrasteratu

Rasterizing points

- Let’s say the point *p* is a point in 3d space, represented by the vector:

$$\begin{bmatrix} 2.5 \\ 1.5 \\ -1 \\ 1 \end{bmatrix}$$

- We want to draw this on screen (Rasterization)
  - 3d scene description → 2d image

Rasterizing points

- Two issues though!
  - Our screen is a 2d coordinate system, and the point is in 3d!
  - Where in the screen is (2.5, 1.5, -1)?
- We have a mismatch in coordinate systems.

$$p' = D \cdot P \cdot C^{-1} \cdot M \cdot p$$

Coordinate Systems

- Object space: What we call our original 3d coordinate system
  ???
  ???
  ???
- **Image space: The 2d coordinate system of the display window**

$$p' = D \cdot p$$

Image Space

- x0 and x1 are 0 and window width respectively
- y0 and y1 are 0 and window height respectively

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix} \frac{x_1 - x_0}{2.0} & 0 & 0 & \frac{x_0 + x_1}{2.0} \\ 0 & \frac{y_1 - y_0}{2.0} & 0 & \frac{y_0 + y_1}{2.0} \\ 0 & 0 & \frac{1.0}{2.0} & \frac{1.0}{2.0} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$p' = D \cdot p$$

Coordinate Systems

- Object space: What we call our original 3d coordinate system
- **World space: After transforming (translation, scaling, rotation)**
  ???
  ???
- Image space: The 2d coordinate system of the display window

$$p' = D \cdot M \cdot p$$

World Space

- Rotation then translation (do it this way)

$$\begin{bmatrix}1 & 0 & 0 & t.x\\ 0 & 1 & 0 & t.y\\ 0 & 0 & 1 & t.z\\ 0 & 0 & 0 & 1\end{bmatrix} * \begin{bmatrix}x_c.x & y_c.x & z_c.x & 0\\ x_c.y & y_c.y & z_c.y & 0\\ x_c.z & y_c.z & z_c.z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$= \begin{bmatrix}(1*x_c.x+0*x_c.y+0*x_c.z+t.x*0) & (1*y_c.x+0*y_c.y+0*y_c.z+t.y*0) & (1*z_c.x+0*z_c.y+0*z_c.z+t.z*0) & (1*0+0*0+0*0+t.x*1)\\ (0*x_c.x+1*x_c.y+0*x_c.z+t.x*0) & (0*y_c.x+1*y_c.y+0*y_c.z+t.y*0) & (0*z_c.x+1*z_c.y+0*z_c.z+t.z*0) & (0*0+1*0+0*0+t.y*1)\\ (0*x_c.x+0*x_c.y+1*x_c.z+t.x*0) & (0*y_c.x+0*y_c.y+1*y_c.z+t.y*0) & (0*z_c.x+0*z_c.y+1*z_c.z+t.z*0) & (0*0+0*0+1*0+t.z*1)\\ (0*x_c.x+0*x_c.y+0*x_c.z+1*0) & (0*y_c.x+0*y_c.y+0*y_c.z+1*0) & (0*z_c.x+0*z_c.y+0*z_c.z+1*0) & (0*0+0*0+0*0+1*1)\end{bmatrix}$$

$$= \begin{bmatrix}x_c.x & y_c.x & z_c.x & t.x\\ x_c.y & y_c.y & z_c.y & t.y\\ x_c.z & y_c.z & z_c.z & t.z\\ 0 & 0 & 0 & 1\end{bmatrix}$$

World Space

- Translation then rotation

$$\begin{bmatrix} x_c.x & y_c.x & z_c.x & 0 \\ x_c.y & y_c.y & z_c.y & 0 \\ x_c.z & y_c.z & z_c.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & t.x \\ 0 & 1 & 0 & t.y \\ 0 & 0 & 1 & t.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} (x_c.x*1 + y_c.x*0 + z_c.x*0 + 0*0) & (x_c.x*0 + y_c.x*1 + z_c.x*0 + 0*0) & (x_c.x*0 + y_c.x*0 + z_c.x*1 + 0*0) & (x_c.x*t.x + y_c.x*t.y + z_c.x*t.z + 0*1) \\ (x_c.y*1 + y_c.y*0 + z_c.y*0 + 0*0) & (x_c.y*0 + y_c.y*1 + z_c.y*0 + 0*0) & (x_c.y*0 + y_c.y*0 + z_c.y*1 + 0*0) & (x_c.y*t.x + y_c.y*t.y + z_c.y*t.z + 0*1) \\ (x_c.z*1 + y_c.z*0 + z_c.z*0 + 0*0) & (x_c.z*0 + y_c.z*1 + z_c.z*0 + 0*0) & (x_c.z*0 + y_c.z*0 + z_c.z*1 + 0*0) & (x_c.z*t.x + y_c.z*t.y + z_c.z*t.z + 0*1) \\ (0*1 + 0*0 + 0*0 + 1*0) & (0*0 + 0*1 + 0*0 + 1*0) & (0*0 + 0*0 + 0*1 + 1*0) & (0*t.x + 0*t.y + 0*t.z + 1*1) \end{bmatrix}$$

$$= \begin{bmatrix} x_c.x & y_c.x & z_c.x & (x_c.x*t.x + y_c.x*t.y + z_c.x*t.z) \\ x_c.y & y_c.y & z_c.y & (x_c.y*t.x + y_c.y*t.y + z_c.y*t.z) \\ x_c.z & y_c.z & z_c.z & (x_c.z*t.x + y_c.z*t.y + z_c.z*t.z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Model Matrix

- Usually denoted *M*, for Model
- We've defined the member variable toWorld for both **cube** and **OBJObject**.

```
void Cube::spin(float deg)
{
    this->angle += deg;
    if (this->angle > 360.0f || this->angle < -360.0f) this->angle = 0.0f;
    // This creates the matrix to rotate the cube
    this->toWorld = glm::rotate(glm::mat4(1.0f), this->angle / 180.0f * glm::pi<float>(), glm::vec3(0.0f, 1.0f, 0.0f));
}
```

- What's wrong with this method?

$$p' = D \cdot M \cdot p$$

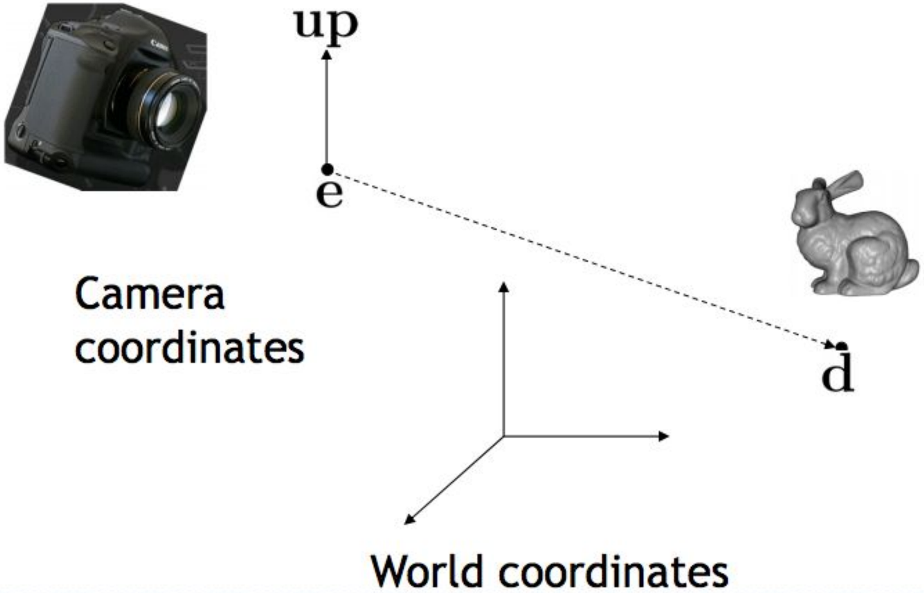
Coordinate Systems

- Object space: What we call our original 3d coordinate system
- World space: After transforming (translation, scaling, rotation)
- **Camera space: How the world looks, centered around camera**
  ???
- Image space: The 2d coordinate system of the display window

$$p' = D \cdot C^{-1} \cdot M \cdot p$$

Camera Space

- Construct from center of projection **e**, look at **d**, up-vector **up**:



Inverse Camera Matrix

- We're actually interested in finding the inverse C^{-1}

$$R^{-1} = R^T = \begin{bmatrix} x_c.x & x_c.y & x_c.z & 0 \\ y_c.x & y_c.y & y_c.z & 0 \\ z_c.x & z_c.y & z_c.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -e.x \\ 0 & 1 & 0 & -e.y \\ 0 & 0 & 1 & -e.z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C^{-1} = (T * R)^{-1} = R^{-1} * T^{-1}$$

Inverse Camera Matrix

- How did we do it in openGL?

```
// Move camera back 20 units so that it looks at the origin (or else it's in the origin)
glTranslatef(0, 0, -20);
```

- Do we have to construct this manually?
- There's a glm function that does this for us!

```
C_inverse = glm::lookAt(e, d, up)
```

$$p' = D \cdot C^{-1} \cdot M \cdot p$$

Coordinate Systems

- Object space: What we call our original 3d coordinate system
- World space: After transforming (translation, scaling, rotation)
- Camera space: How the world looks, centered around camera
- **Projection space: How the world looks with perspective**
- Image space: The 2d coordinate system of the display window

$$p' = D \cdot P \cdot C^{-1} \cdot M \cdot p$$

Projection Space

- Perspective projection matrix(P) will create perspective transform, and put our Camera Space into the canonical view volume

$$P_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1.0}{aspect*tan(FOV/2.0)} & 0 & 0 & 0 \\ 0 & \frac{1.0}{tan(FOV/2.0)} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2.0*near*far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective Projection

- How did we do it in openGL?

```
// Set the perspective of the projection viewing frustum
gluPerspective(60.0, double(width) / (double)height, 1.0, 1000.0);
```

- Do we have to construct this manually?
- There’s a glm function that does this for us!

```
P = glm::perspective(glm::radians(60.0f),
                     (float) width / (float) height,
                     1.0f, 1000.0f);
```

$$p' = D \cdot P \cdot C^{-1} \cdot M \cdot p$$

Interactive Graphics

- When/where do we change these for project 1?
  - Object: Doesn't change! Inherent to object
  - World: When the object transforms (e.g. cube's spin, keyboard input)
  - Camera: We're not moving the camera for this project
  - Projection: What happens when user changes the aspect ratio?
  - Image: What happens when the viewport becomes larger or smaller?

$$P_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1.0}{aspect*tan(FOV/2.0)} & 0 & 0 & 0 \\ 0 & \frac{1.0}{tan(FOV/2.0)} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2.0*near*far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix} \frac{x_1-x_0}{2.0} & 0 & 0 & \frac{x_0+x_1}{2.0} \\ 0 & \frac{y_1-y_0}{2.0} & 0 & \frac{y_0+y_1}{2.0} \\ 0 & 0 & \frac{1.0}{2.0} & \frac{1.0}{2.0} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Resizing

- What happens when we resize the window?
  - aspect, *x1, x0, y1, y0* change
  - Have to update P and D.
- What about pixel buffer?

$$P_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1.0}{aspect*tan(FOV/2.0)} & 0 & 0 & 0 \\ 0 & \frac{1.0}{tan(FOV/2.0)} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2.0*near*far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix} \frac{x_1-x_0}{2.0} & 0 & 0 & \frac{x_0+x_1}{2.0} \\ 0 & \frac{y_1-y_0}{2.0} & 0 & \frac{y_0+y_1}{2.0} \\ 0 & 0 & \frac{1.0}{2.0} & \frac{1.0}{2.0} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Resizing

- Resizing should change the total number of pixels
  - We provide code for this

```
// Called whenever the window size changes
void resizeCallback(GLFWwindow* window, int width, int height)
{
    window_width = width;
    window_height = height;
    delete[] pixels;
    pixels = new float[window_width * window_height * 3];
}
```

Some Code Tips

- OpenGL expects column major, not row major.
- How do we end up drawing to the buffer?

```
// glDrawPixels writes a block of pixels to the framebuffer
glDrawPixels(window_width, window_height, GL_RGB, GL_FLOAT, pixels);
```

Some Code Tips

- Manually changing the `pixels` array is tedious
  - `drawPoint(int x, int y, float r, float g, float b)`
    - At the index [x][y] of our pixel buffer, let's set our color to be r, g, b
    - ```
      int offset = y*width*3 + x*3;
      pixels[offset]   = r;
      pixels[offset+1] = g;
      pixels[offset+2] = b;
      ```
      ```

Some Code Tips

```
Foo bar = Foo();
```

- Foo constructor called
- Foo object created
- Data copied over from temporary Foo() object into bar
- ~Foo() destructor called on temporary Foo object

```
Foo *bar = new Foo();
```

- Foo constructor called
- Foo object created
- Address of Foo() object that was created is saved in bar

Some Code Tips

- **DO NOT** load your OBJ files from disk every frame
- How can we efficiently implement switching between OBJ models without having a series of if statements?
  - What if we had a Drawable* and we change what it points to whenever we need to switch between models?
- Multiple OBJ objects?