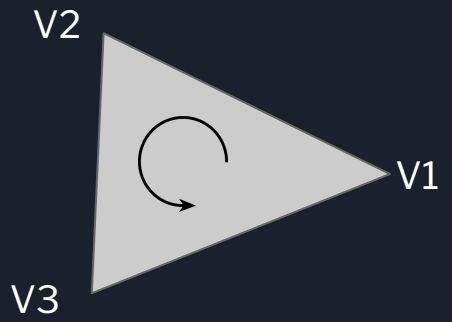
Discussion 4
CSE 167

Outline

- Parsing Faces
- Mouse Interaction
- Virtual Trackball
- Shaders

Parsing Faces

- Face Lines in obj file format: f v1 // vn1 v2 // vn2 v3 //vn3
  - Three integers represent a triangle created by our vertices at index x, y, and z, forming a face!
- For each face, we want to store these three 3 indices inside a ivec3 or uvec3



- When parsing, check for...
  - the delimiters ‘//’
  - Index 1 represents the first index of our vertex (Subtract each index by 1 when storing faces)
- After parsing, pass in to EBO in the constructor and use glDrawElements with GL_TRIANGLES in draw() (**Review Discussion 3**)

Mouse Interaction

glfwSetMouseButtonCallback(window, mouse_button_callback);

- Checks if mouse has been pressed or released

```
void mouse_button_callback(GLFWwindow* window, int button, int action,
int mods){

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {

        // some function/statements

    }

}
```

Mouse Interaction

glfwSetCursorPosCallback(window, cursor_pos_callback);

- Receives the cursor position, measured in screen coordinates but relative to the top-left corner of the window content area

```
void cursor_position_callback(GLFWwindow* window, double
xpos, double ypos) {

    double pos_x = xpos; double pos_y = ypos;

}
```

Mouse Interaction

glfwSetScrollCallback(window, scroll_callback);

- Receives receives two-dimensional scroll offsets.

```
void scroll_callback(GLFWwindow* window, double xoffset,
double yoffset){

    double x_off = xoffset; double y_off = yoffset;

}
```

Mouse Interaction

Set up call back functions in main.cpp file, in the setup_callbacks(GLFWwindow* window) method

- `glfwSetMouseButtonCallback(window, mouse_button_callback);`
- `glfwSetCursorPosCallback(window, cursor_pos_callback);`
- `glfwSetScrollCallback(window, scroll_callback);`

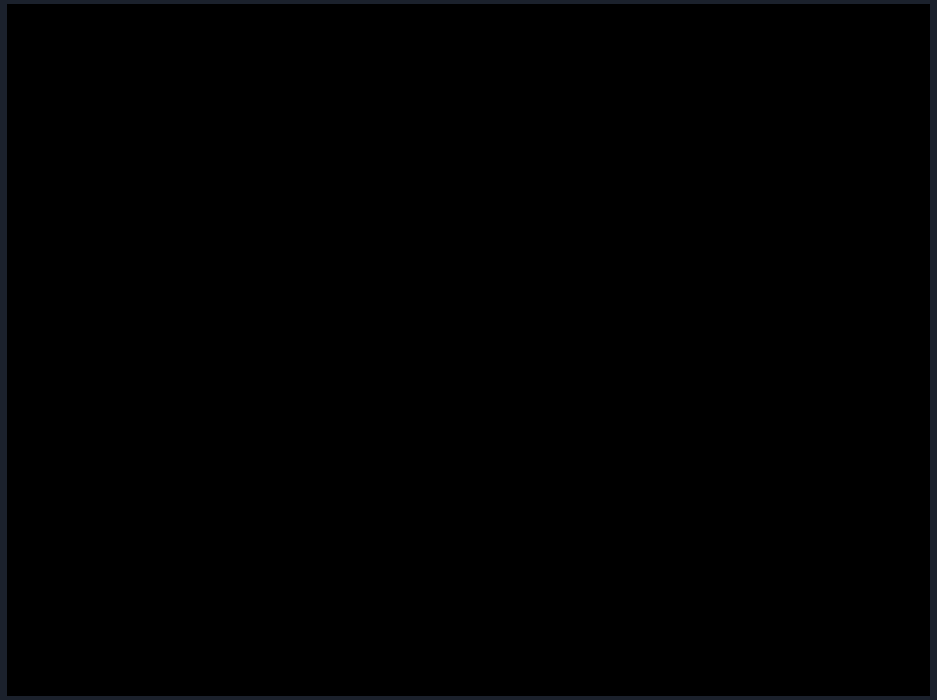
```
void setup_callbacks(GLFWwindow* window)
{
    // Set the error callback.
    glfwSetErrorCallback(error_callback);

    // Set the window resize callback.
    glfwSetWindowSizeCallback(window, Window::resizeCallback);

    // Set the key callback.
    glfwSetKeyCallback(window, Window::keyCallback);
}
```

https://www.glfw.org/docs/latest/input_guide.html

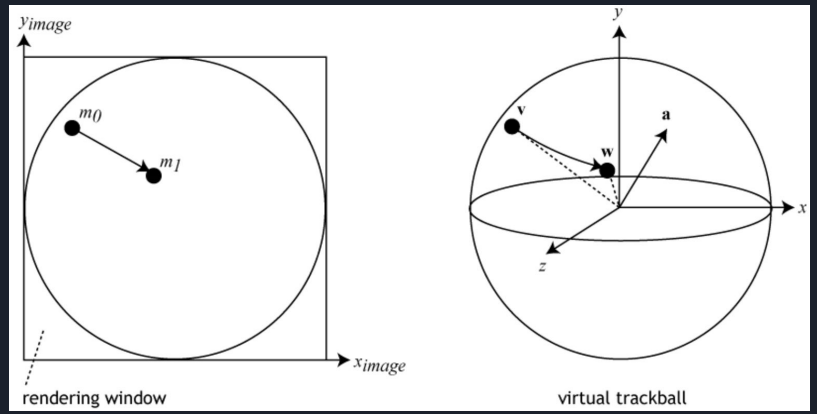
Virtual Trackball



Virtual Trackball

- Taking two different 2D screen positions and mapping them into two 3D vectors
  - m_0: Mouse position in the **previous** frame
  - m_1: Mouse position in the **current** frame

- Based on these 3D vectors you find the angle and axis to rotate your model
  - Angle: angle between these two vectors
  - Axis: perpendicular to both of these vectors

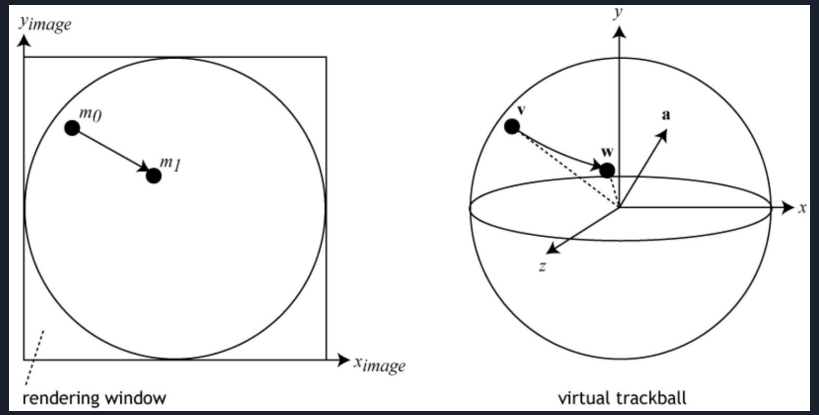


Virtual Trackball - Mapping 2D to 3D

```
Vec3f CSierpinskiSolidsView::trackBallMapping(CPoint point)
{

        Vec3f v;
        float d;
        v.x = (2.0*point.x - windowSize.x) / windowSize.x;
        v.y = (windowSize.y - 2.0*point.y) / windowSize.y;
        v.z = 0.0;
        d = v.Length();
        d = (d<1.0) ? d : 1.0;
        v.z = sqrtf(1.001 - d*d);
        v.Normalize(); // Still need to normalize, since we only capped d, not v.
        return v;

}
```



Virtual Trackball

1. On mouse click, note current mouse position
2. On mouse move, get new mouse position
3. Map 2D position to 3D positions on a virtual trackball
4. Ensure 3D velocity exceeds a small threshold
5. Calculate the axis of rotation (cross product)
6. Calculate angle of rotation (approximated by scalar times velocity)
7. Compute and apply rotation matrix to object's model matrix (LEFT multiply!)
8. Update current mouse position

- See full tutorial here (Note differing libraries used)

Shaders - GLSL Review

- Types: bool, int, uint, float, double
  - can be vectors (e.g vec2, ivec3, etc.)
    - access elements in vector with component names (xyzw, rgba, stpq) or by 0-indexed subscript notation (e.g., v[0])
    - swizzling can extract and reorder vector data (e.g., v.zyx)
- Matrices: mat2, mat3, mat4, mat4x3, etc
  - Access columns of matrices by subscript notation (e.g., m[0])
- Can use Arrays and Structs as well!
- Keywords:
  - layout: specifies where in storage a variable comes from
  - in: input to the shader
  - out: output from the shader
  - uniform: global data (can be used in any shader)

Shaders - Vertex Shader

```
1  #version 330 core
2  layout (location = 0) in vec3 position;
3  layout (location = 1) in vec3 normal;
4
5  uniform mat4 projection;
6  uniform mat4 view;
7  uniform mat4 model;
8
9  out vec3 normalOutput;
10 out vec3 posOutput;
11
12 void main()
13 {
14     gl_Position = projection * view * model * vec4(position, 1.0);
15
16     ...
17 }
```

TODO: Transform vertices and normals from local coordinate to world coordinate before passing it to fragment shaders.

Warning: please read here on **normal matrix** to avoid transforming normals incorrectly.

Shaders - Vertex Shader

```
2  layout (location = 0) in vec3 position;
3  layout (location = 1) in vec3 normal;
```

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

- In PointCloud.cpp, the first parameter of glVertexAttribPointer should be the same as the location number in the shader.

```
5  uniform mat4 projection;
6  uniform mat4 view;
7  uniform mat4 model;
```

```
glUniformMatrix4fv(glGetUniformLocation(shader, "view"), 1, false, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(shader, "projection"), 1, false, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shader, "model"), 1, GL_FALSE, glm::value_ptr(model));
```

- In draw() of PointCloud.cpp, the argument into glUniformLocation should have the same name as the parameters in the shader.

Shaders - Fragment Shader

- normalOutput and posOutput are the output from Vertex Shader.
- You should pass light attributes (e.g. color) as glUniforms to specify the attributes of light source.
- fragColor is the final color of the pixel coming out of the shader.
- TODO: Use phong lighting and linear attenuation to calculate fragment color here.

```
1  #version 330 core
2
3  in vec3 normalOutput;
4  in vec3 posOutput;
5
6  uniform vec3 lightAttr1;
7  uniform vec3 lightAttr2;
8  ...
9
10 out vec4 fragColor;
11
12 void main()
13 {
14     vec3 ambient = ...
15
16     vec3 diffuse = ...
17
18     vec3 specular = ...
19
20     fragColor = ...
21 }
```

Shaders - Loading and Using Shaders

```
shaderProgram = LoadShaders("shaders/shader.vert", "shaders/shader.frag");
```

- Load and compile shader in Window::initializeProgram()
  - Store the output to identify the shader program

```
currObj->draw(view, projection, shaderProgram);
```

- Pass the desired shader into the draw method of an object

```
glUseProgram(shader);
```

- The draw method will set the current shader program

Any questions?