CSE 167

Discussion 03 ft. Glynn
10/16/2017

Announcements

- Midterm next Thursday(10/23)
  - Sample midterms are up
- Project 1 late grading until this Friday
  - You will receive 75% of the points you've earned

Contents

- Trackball mouse control
  - Demo
  - Math
  - Implementation details
- Modern OpenGL
  - Shaders
  - VAO/VBO/EBO
- Some stuff about parsing faces

Trackball mouse control: demo

-

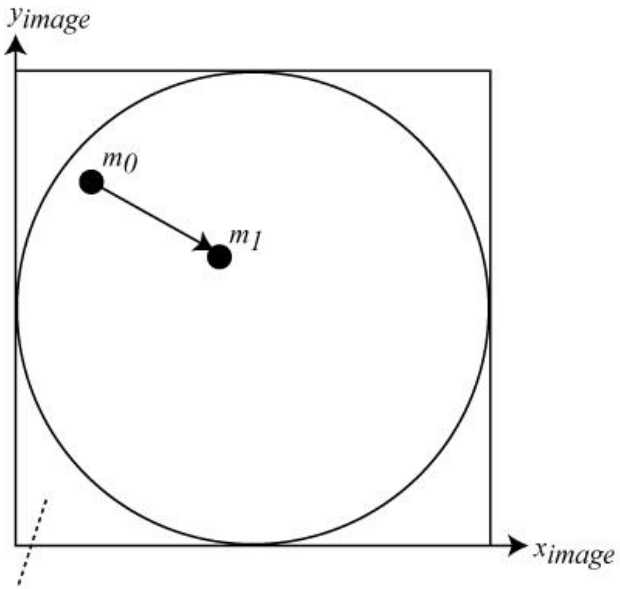
Trackball mouse control: math I

- How to map 2D cursor position (x, y) back to 3D space? Chalkboard time!

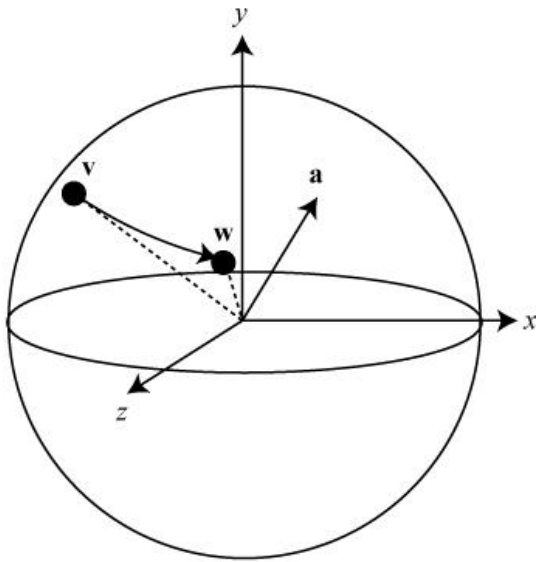
```
Vec3f trackBallMapping(CPoint point)
{
    Vec3f v;
    float d;
    v.x = (2.0*point.x - windowSize.x) / windowSize.x;
    v.y = (windowSize.y - 2.0*point.y) / windowSize.y;
    v.z = 0.0;
    d = v.Length();
    d = (d<1.0) ? d : 1.0;
    v.z = sqrtf(1.001 - d*d);
    v.Normalize();
    return v;
}
```

Trackball mouse control: math II

- How to define rotation axis, angle and speed? Chalkboard time!

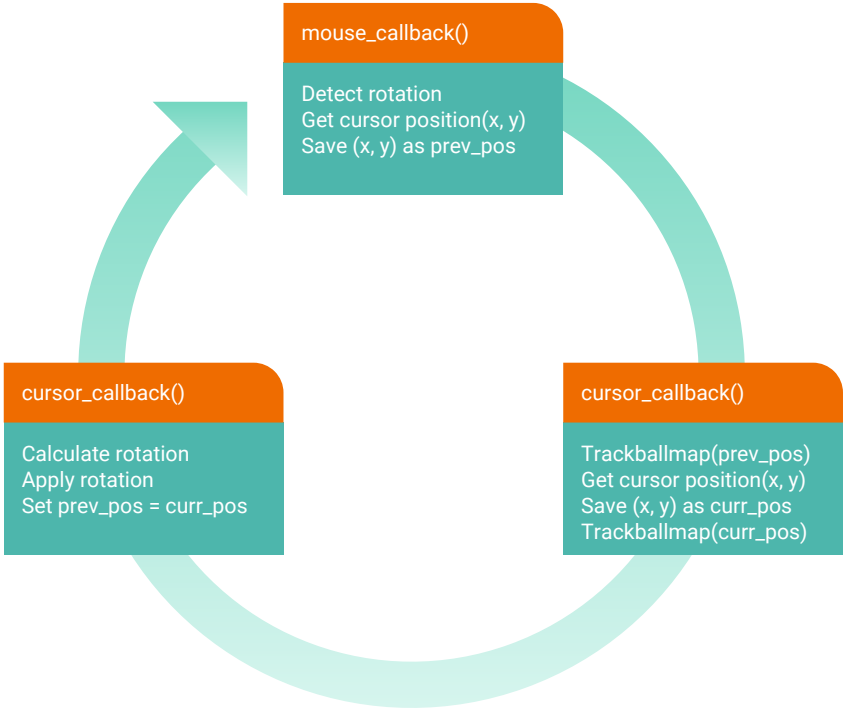


rendering window



virtual trackball

Trackball mouse control: implementation details



Trackball mouse control: implementation details

- http://www.glfw.org/docs/latest/input_guide.html#input_mouse
- Get mouse input
  - main.cpp: `glfwSetMouseButtonCallback(window, mouse_button_callback);`
  - window.cpp: define `void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)`
- Get cursor position
  - main.cpp: `glfwSetCursorPosCallback(window, cursor_pos_callback);`
  - window.cpp: define `static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)`

Modern OpenGL: shaders

- Modern OpenGL pipeline

Vertices → Geometry → Rasterization

Vertex Buffer Object (VBO)

- Vertex Shader

- Fragment Shader

Modern OpenGL: shaders

- Vertex shader: handles processing of individual vertices

```
#version 330 core
// NOTE: Do NOT use any version older than 330! Bad things will happen!

// This is an example vertex shader. GLSL is very similar to C.
// You can define extra functions if needed, and the main() function is
// called when the vertex shader gets run.
// The vertex shader gets called once per vertex.

layout (location = 0) in vec3 position;

// Uniform variables can be updated by fetching their location and passing values to that location
uniform mat4 projection;
uniform mat4 modelview;

// Outputs of the vertex shader are the inputs of the same name of the fragment shader.
// The default output, gl_Position, should be assigned something. You can define as many
// extra outputs as you need.
out float sampleExtraOutput;

void main()
{
   // OpenGL maintains the D matrix so you only need to multiply by P, V (aka C inverse), and M
   gl_Position = projection * modelview * vec4(position.x, position.y, position.z, 1.0);
   sampleExtraOutput = 1.0f;
}
```

Modern OpenGL: shaders

```
void Cube::draw(GLuint shaderProgram)
{
	// Calculate the combination of the model and view (camera inverse) matrices
	glm::mat4 modelview = Window::V * toWorld;
	// We need to calculate this because modern OpenGL does not keep track of any matrix other than the viewport (D)
	// Consequently, we need to forward the projection, view, and model matrices to the shader programs
	// Get the location of the uniform variables "projection" and "modelview"
	uProjection = glGetUniformLocation(shaderProgram, "projection");
	uModelview = glGetUniformLocation(shaderProgram, "modelview");
	// Now send these values to the shader program
	glUniformMatrix4fv(uProjection, 1, GL_FALSE, &Window::P[0][0]);
	glUniformMatrix4fv(uModelview, 1, GL_FALSE, &modelview[0][0]);
	// Now draw the cube. We simply need to bind the VAO associated with it.
	glBindVertexArray(VAO);
	// Tell OpenGL to draw with triangles, using 36 indices, the type of the indices, and the offset to start from
	glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
	// Unbind the VAO when we're done so we don't accidentally draw extra stuff or tamper with its bound buffers
	glBindVertexArray(0);
}
```

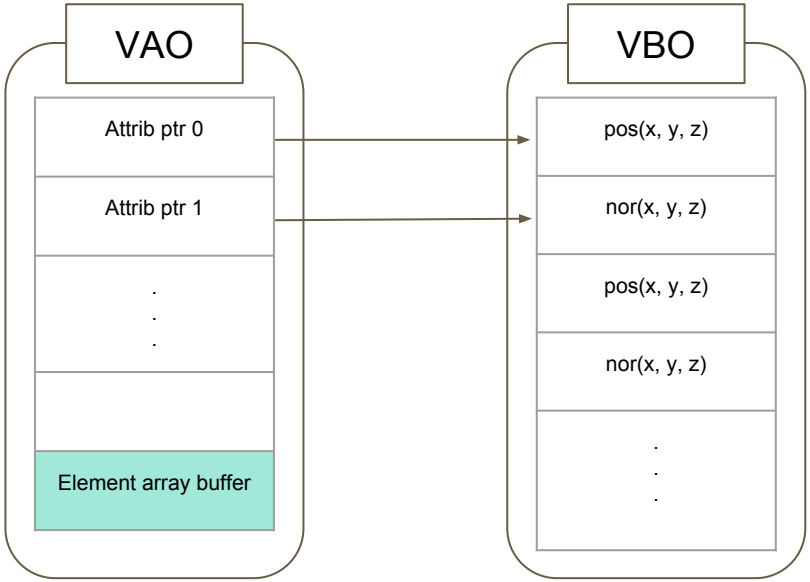
Modern OpenGL: shaders

- Fragment shader: handles processing of fragments created by rasterization

```
#version 330 core
// This is a sample fragment shader.

// Inputs to the fragment shader are the outputs of the same name from the vertex shader.
// Note that you do not have access to the vertex shader's default output, gl_Position.
in float sampleExtraOutput;

// You can output many things. The first vec4 type output determines the color of the fragment
out vec4 color;

void main()
{
    // Color everything a hot pink color. An alpha of 1.0f means it is not transparent.
    color = vec4(1.0f, 0.41f, 0.7f, sampleExtraOutput);
}
```

Modern OpenGL: VAO, VBO, EBO

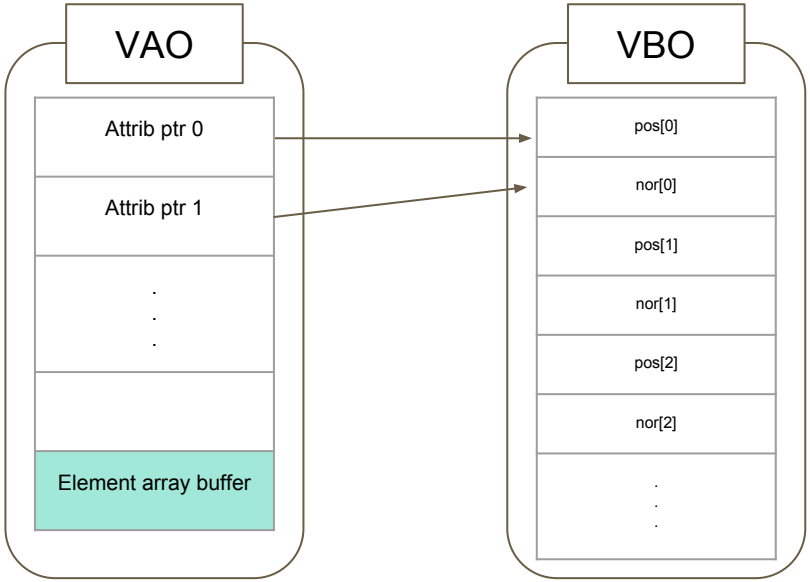
Single VBO, glm::vec3




```
Cube::Cube()
{
        toWorld = glm::mat4(1.0f);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0,// This first parameter x should be the same as the number passed
into the line "layout (location = x)" in the vertex shader. In this case, it's 0. Valid values are 0 to
GL_MAX_UNIFORM_LOCATIONS.
                3, // This second line tells us how any components there are per vertex. In this
case, it's 3 (we have an x, y, and z component)
                GL_FLOAT, // What type these components are
                GL_FALSE, // GL_TRUE means the values should be normalized. GL_FALSE
means they shouldn't
                3 * sizeof(GLfloat), // Offset between consecutive indices. Since each of our
vertices have 3 floats, they should have the size of 3 floats in between
                (GLvoid*)0); // Offset of the first vertex's component. In our case it's 0 since we
don't pad the vertices array with anything.

...

}
```

Modern OpenGL: VAO, VBO, EBO

Single VBO, float

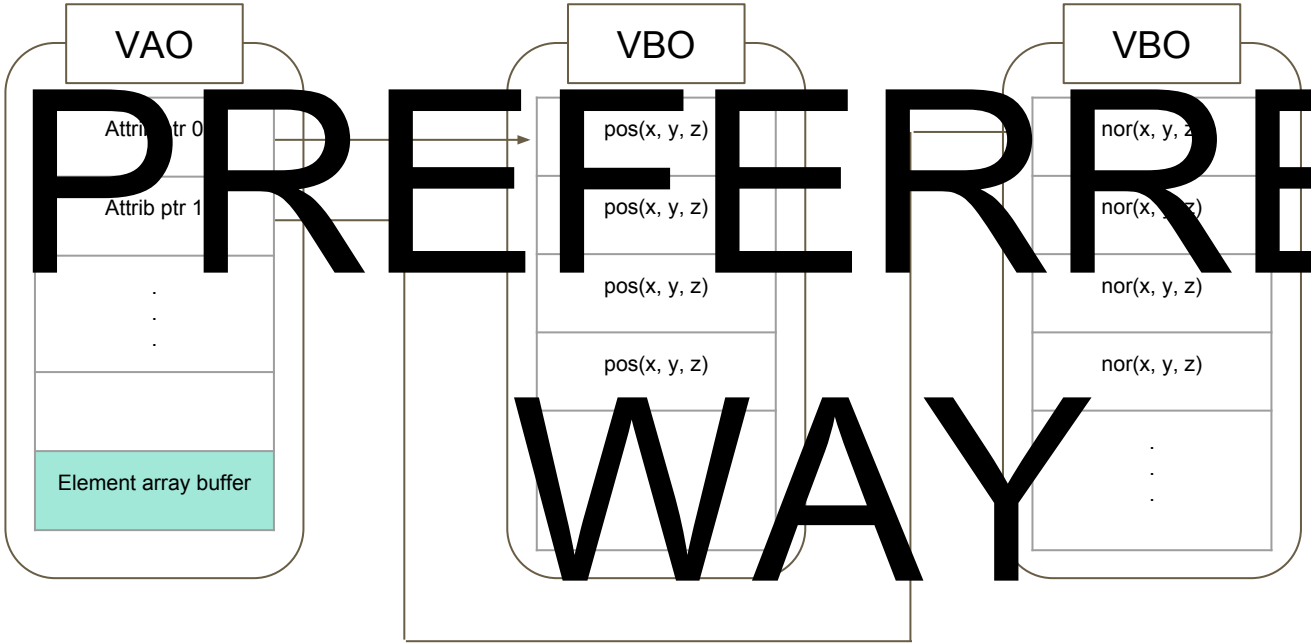



```
Cube::Cube()
{
        toWorld = glm::mat4(1.0f);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0,// This first parameter x should be the same as the number passed
into the line "layout (location = x)" in the vertex shader. In this case, it's 0. Valid values are 0 to
GL_MAX_UNIFORM_LOCATIONS.
                3, // This second line tells us how any components there are per vertex. In this
case, it's 3 (we have an x, y, and z component)
                GL_FLOAT, // What type these components are
                GL_FALSE, // GL_TRUE means the values should be normalized. GL_FALSE
means they shouldn't
                ?, // Offset between consecutive indices. Since each of our vertices have 3
floats, they should have the size of 3 floats in between
                (GLvoid*)0); // Offset of the first vertex's component. In our case it's 0 since we
don't pad the vertices array with anything.

...

}
```

Modern OpenGL: VAO, VBO, EBO

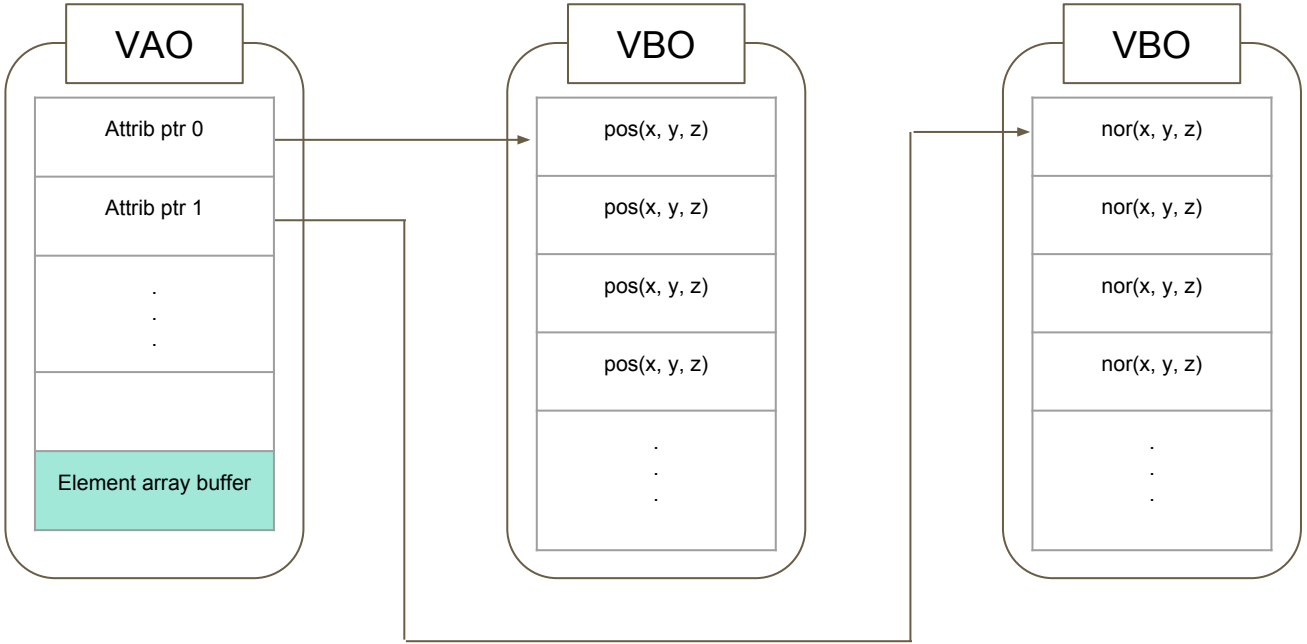
THE PREFERRED WAY

Multiple VBOs, glm::vec3



Modern OpenGL: VAO, VBO, EBO

Multiple VBOs, glm::vec3



Modern OpenGL: VAO, VBO, EBO

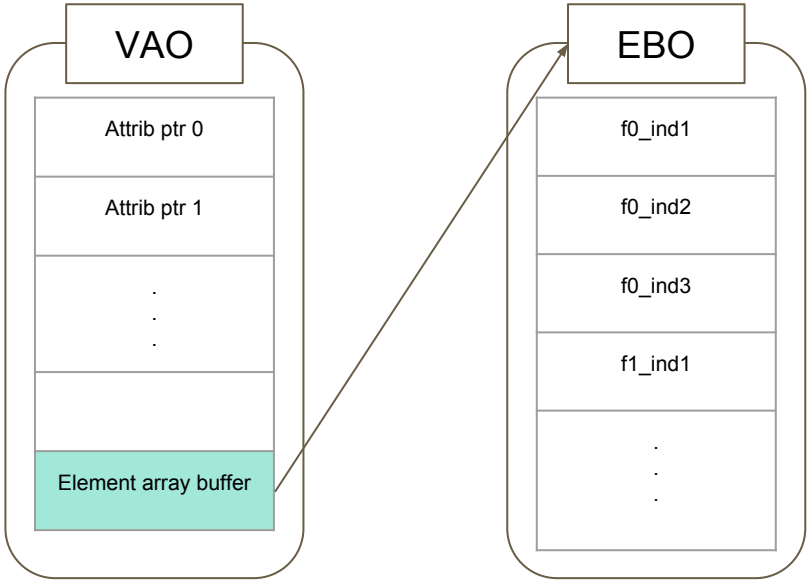
Multiple VBOs, glm::vec3

```
Cube::Cube()
{
        toWorld = glm::mat4(1.0f);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        // (1) Generate VBO for normals, e.g. VBO2
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, …);

        // (2) Bind VBO2
        // (3) Send buffer data of VBO2

        // (4) Enable vertex attribute array with position 1
        // (5) Define vertex attribute pointer for position 1

        ...
}
```

```
#version 330 core
// NOTE: Do NOT use any version older than 330! Bad things will happen!

layout (location = 0) in vec3 position;
// Create layout for location = 1

// Uniform variables can be updated by fetching their location and passing values
to that location
uniform mat4 projection;
uniform mat4 modelview;

// Outputs of the vertex shader are the inputs of the same name of the fragment
shader.
// The default output, gl_Position, should be assigned something. You can define
as many
// extra outputs as you need.
out float sampleExtraOutput;

void main()
{
   // OpenGL maintains the D matrix so you only need to multiply by P, V (aka C
inverse), and M
   gl_Position = projection * modelview * vec4(position.x, position.y, position.z,
1.0);
   sampleExtraOutput = 1.0f;
}
```

Modern OpenGL: VAO, VBO, EBO

EBO




```
Cube::Cube()
{
        toWorld = glm::mat4(1.0f);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(...)

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindVertexArray(0);

}
```

Some stuff about parsing faces

- Indices start from 1 not 0!!!
- Indices are stored as unsigned int not glm::vec3!