CSE 167 Discussion #7

Control, control, you must lerp control

Bezier Curves

- The general form of a Bezier Curve can have any number of control points
- In general, the more control points used to generate a curve, the more accurately it can represent various non-polynomial curves

$$\dot{q} = \sum_{i=0}^{n} \left(\binom{n}{i} (t)^i (1-t)^{n-i} * \dot{p}_i \right)$$

Bezier Curves

- There is a fairly massive diminishing return in the computing world: the amount of accuracy gained from each additional control point is terribly small in comparison to the increase in compute time needed to evaluate the curve

$$\dot{q} = \sum_{i=0}^{n} \left(\binom{n}{i} (t)^i (1-t)^{n-i} * \dot{p}_i \right)$$

Bezier Curves

- So we make a compromise: use 4 control points and hope for the best!

$$\dot{q} = \sum_{i=0}^{n} \left(\binom{n}{i} (t)^i (1 - t)^{n-i} * \dot{p}_i \right)$$

Bezier Curves

- Starting with 4 control points $\{\dot{p}_0, \dot{p}_1, \dot{p}_2, \dot{p}_3\}$, and a time *t*, we can interpolate all of the control points at time *t* using this quite large, and potentially scary equation:

$$\dot{q} = \sum_{i=0}^{3} \left(\binom{3}{i} (t)^i (1-t)^{3-i} * \dot{p}_i \right)$$

Bezier Curves

- Note that what was n in the general equation has now been replaced with 3. This is due to us using 4 control points. n is always the number of control points - 1.

$$\dot{q} = \sum_{i=0}^{3} \left(\binom{3}{i} (t)^i (1-t)^{3-i} * \dot{p_i} \right)$$

Bezier Curves

- Remembering that combinations expand to:

$$\binom{3}{i} = \frac{3!}{(3-i)! \cdot i!}$$

$$\dot{q} = \sum_{i=0}^{3} \left(\binom{3}{i} (t)^i (1-t)^{3-i} * \dot{p}_i \right)$$

Feel the Bernstein

- At this point we notice that given some time *t* that the leading coefficient evaluates to a constant scalar, so we can replace it with a convenient function $C_i(t)$, the Bernstein Polynomial

$$C_i(t) = \frac{3!}{(3-i)! \cdot i!}(t)^i(1-t)^{3-i}$$

Bezier Curves

- Substituting back into our equation:

$$\dot{q} = \sum_{i=0}^{3} (C_i(t) * \dot{p}_i)$$

$$\dot{q} = C_0(t) * \dot{p}_0 + C_1(t) * \dot{p}_1 + C_2(t) * \dot{p}_2 + C_3(t) * \dot{p}_3$$

Bezier Curves

- This is nothing more than adding together 4 vectors that have each been multiplied by a scalar weight!

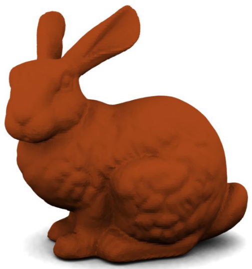
$$\begin{bmatrix} \dot{q}_x \\ \dot{q}_y \\ \dot{q}_z \end{bmatrix} = C_0(t) \begin{bmatrix} \dot{p}_{0x} \\ \dot{p}_{0y} \\ \dot{p}_{0z} \end{bmatrix} + C_1(t) \begin{bmatrix} \dot{p}_{1x} \\ \dot{p}_{1y} \\ \dot{p}_{1z} \end{bmatrix} + C_2(t) \begin{bmatrix} \dot{p}_{2x} \\ \dot{p}_{2y} \\ \dot{p}_{2z} \end{bmatrix} + C_3(t) \begin{bmatrix} \dot{p}_{3x} \\ \dot{p}_{3y} \\ \dot{p}_{3z} \end{bmatrix}$$

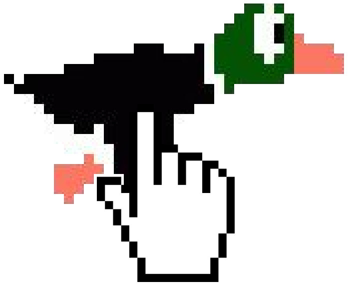
Bezier Curves

- Which is a matrix-vector product in disguise!:

$$\begin{bmatrix} \dot{q}_x \\ \dot{q}_y \\ \dot{q}_z \end{bmatrix} = \begin{bmatrix} \dot{p}_{0x} & \dot{p}_{1x} & \dot{p}_{2x} & \dot{p}_{3x} \\ \dot{p}_{0y} & \dot{p}_{1y} & \dot{p}_{2y} & \dot{p}_{3y} \\ \dot{p}_{0z} & \dot{p}_{1z} & \dot{p}_{2z} & \dot{p}_{3z} \end{bmatrix} \cdot \begin{bmatrix} C_0(t) \\ C_1(t) \\ C_2(t) \\ C_3(t) \end{bmatrix}$$

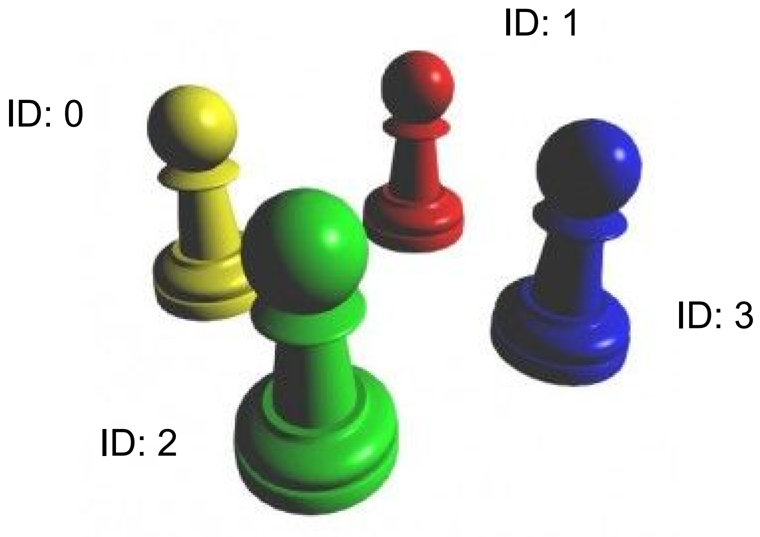
Selection





Selection Buffers

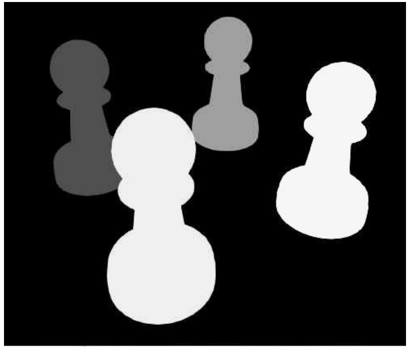
- Each selectable object in your scene will have an id



Selection Buffers

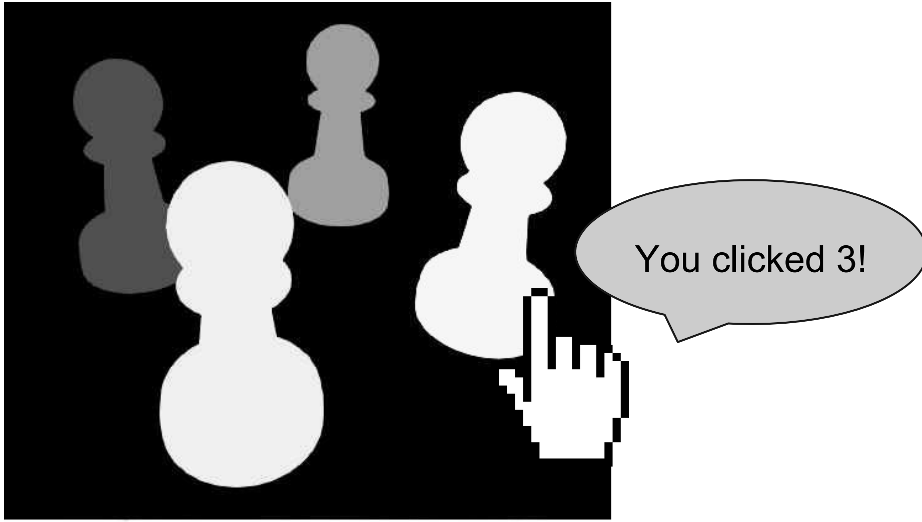
- On mouse click, re-render the scene with a selection shader, colored by the ID
- How? Use uniforms!

```
uniform uint id;
```



Selection Buffers

- Read the pixel color at that point
  → Retrieve the ID



A Selection Shader

Control.cpp

```
selectionDraw(GLuint shaderProgram)

{

    ...

    glPointSize(10.0f); // Make points larger for easier selection

    GLuint idLocation = glGetUniformLocation(shaderProgram, "id");

    glUniform1ui(idLocation, ID);

    ...

}
```

shader.frag

```
#version 330 core

uniform uint id;

out vec4 color;


void main()

{

    color = vec4(id/255.0f, 0.0f, 0.0f, 0.0f);

}
```

Window.cpp (pseudocode)

1. When mouse is clicked, draw all selectables with selection shader.
2. Read the pixel colored in by the shader
3. Recover ID from that pixel

A Selection Shader

shader.frag

```
#version 330 core
uniform uint id;
out vec4 color;

void main()
{
    color = vec4(id/255.0f, 0.0f, 0.0f, 0.0f);
}
```

Window.cpp

```
void Window::mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        for(auto & selectable : selectables)
        {
            selectable->selectionDraw(selectionShader);
        }
        unsigned char pix[4];
        glReadPixels(xpos, height - ypos, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, &pix);
        selected = selectable[(unsigned int) pix[0]];
        ...
}
```

Raycasting

- Shoot a ray from the camera towards the mouse
- Find the first object that intersects with the ray
  That object is now selected!
- A bit more math heavy way of selecting than selection buffer
- If you want to learn more, take CSE 168 or read this tutorial:
  http://antongerdelan.net/opengl/raycasting.html