CSE 167

Discussion 06 ft. Kaiser
11/13/2017

Announcements

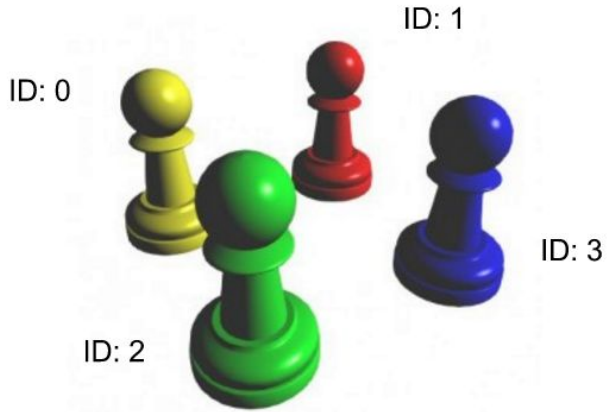
- Project 4 is due 11/27(Mon) 2PM
- Project 3 late grading is due this Friday

Contents

- Selection buffer
- Raycast
- Bezier curve
- Roller coaster physics

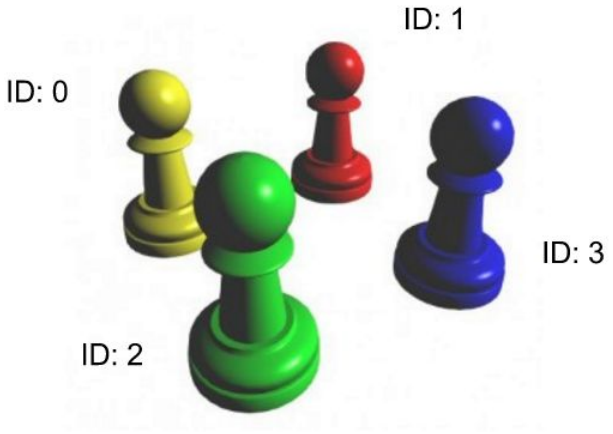
Selection buffer

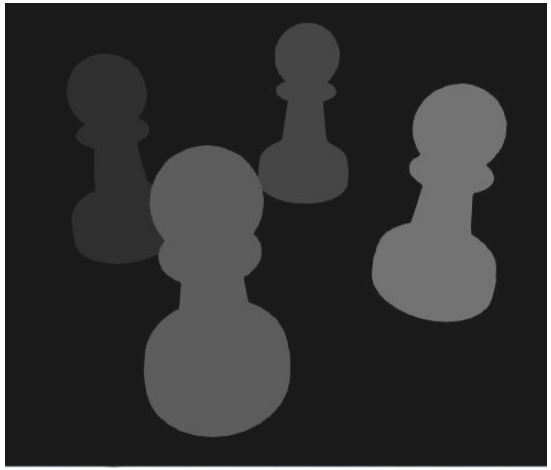
- How do we distinguish between control points? That is, how do we know which of the control points we clicked on?
  - Each clickable object in your scene should have a different ID, for example



Selection buffer

- When the mouse is clicked, redraw the scene with each object colored by their IDs
  - This can be achieved by uniform variables: uniform uint id





Selection buffer

- Let’s take a quick look at how this works in the shader
  - Note: glPointSize() *may* not work on some systems. On the bright side, you have code that draws big spheres in project 3 in case glPointSize() fails
  - You can assign any color you wish, so long as it’s different for each ID

Control.cpp

```
selectionDraw(GLuint shaderProgram)
{
    ...
    glPointSize(10.0f); // Make points larger for easier
selection
    GLuint idLocation = glGetUniformLocation(shaderProgram,
"id");
    glUniform1ui(idLocation, ID);
    ...
}
```

selection.frag

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

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

Selection buffer

- Now we can figure out the ID of the object you clicked on (if any) by reading the color values of the pixel the mouse is on top of

selection.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 = selectables[(unsigned int) pix[0]];
        ...
}
```

Selection buffer

- Implementation tips:
  - Reserve ‘0’ for background i.e. ‘0’ should be reserved for cases where nothing is selected. Alternatively, you can use ‘-1’ for background
  - Use only one color component for selection i.e. use R out of RGB
  - glReadPixels( ) for getting the pixel value
  - Reference tutorial

Ray cast

- Concept: shoot a ray from the camera to the point where mouse was clicked
- Along this ‘ray’, the hit will be with the nearest object

Bézier Curves

- Recall that a Bézier curve can have any number of control points
- The more control points, the more accurately it can represent non-polynomial curves (at the cost of computing time)
- The general formula for Bézier curves is

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

Bézier Curves

- Compromise: Use 4 control points and hope that the curve looks nice enough!
  - Of course, you can use any other number of control points
  - Not fewer than 4, because that looks ugly
- You'll see later why 4 is the preferred number of control points
  - Set n = 3 in the previous formula

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

Bézier Curves

- Given 4 control points {p_0, p_1, p_2, p_3} and a given time t, where $0.0 \leq t \leq 1.0$, you can interpolate a point on a curve using the formula in the previous slide, which is actually way more pleasant than it is long.
- Recall the Combination (3 choose i) is simply
  - This always evaluates to either 1 or 3
  - Please don't actually write a Combination function

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

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

Bézier Curves

- For some given time t, the terms before p_i always evaluates to a constant.
- We call the function that calculates this constant value the Bernstein polynomial
  - We use C as it's **c**onst
  - Lecture slides use B

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

Bézier Curves

- Substituting this back into the Bézier curve function gives us

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

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

Bézier Curves

- But since each p_i is a point, the previous substitution is no different than adding together vectors with scalar weights

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

Bézier Curves

- But this is a matrix * vector product in disguise!

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

Roller coaster physics: assumptions

- For the sake of simplifying things, let us assume our sphere travels around with no friction or air resistance
- The sphere therefore only has kinetic energy and potential energy
  - Kinetic energy is the energy the sphere has by being in motion
  - Potential energy is the energy the sphere has by virtue of its position
- Conservation of energy tells us any potential energy that gets lost will be converted into kinetic energy
  - $\frac{1}{2} mv^2 + ma\Delta h = 0$, where
  - m is the mass of the object
  - a is an accelerational constant
  - v is the velocity of the object
  - Δh is the change in height of the object (current height - max height)

Roller coaster physics: energy conservation

- Let us assume the mass is non-zero. Divide both sides by m to get:
  - $\frac{1}{2} v^2 + a\Delta h = 0$
- Now we can solve for v:
  - $v = \sqrt{-2a\Delta h}$
- Now we have a very simple way of calculating our velocity!
  - Δh is the height difference between the current height and **max** height ($\Delta h \leq 0$ always)
  - Please don't set *a* to the gravitational constant -9.81 (m/s^2)
    - Play around with values to see which ones work nicely
- If using the above formula, $\Delta h = 0$ when we're at the max height
  - Give it a little nudge so it can always move forward!
  - $v = \sqrt{-2a\Delta h} + c$, where c is some arbitrary constant

Roller coaster physics: moving

- Now that we've calculated our velocity, how do we actually move our sphere along the tracks?
- The velocity v that we calculated is technically in world space
  - But we don't know what direction to move in, or if the point we want to move to is part of the track or not
  - Let's make the simplifying assumption that instead of world space, v is in the "Bézier Curve space"
    - Instead of adding v to the world coordinates of the sphere, we can just add it to the current time t instead!
    - We already have the Bernstein polynomial to help interpolate a point on the curve at any given time

Roller coaster physics: moving

- Let us assume that the sphere is currently at B(t), a point on the Bézier curve evaluated at time t
- We've calculated our velocity to be v, so in the next frame, the sphere should now be at B(t + v)
- If (t + v) > 1.0, that means we've moved onto the next curve
  - Simply subtract 1 from t (i.e. t -= 1.0) and use the control points of the next curve for your position calculations
- We don't need to worry about the case where (t + v) < 0.0 since we're always moving forward