CS 4204 Animation Project

(with Bezier curves & Quaternion rotations)

By Gregory Wainer
and Danny Spatz

Libraries

- OpenGL
  - Cross-Platform API for GPU's
- GLFW3
  - Cross-Platform API for creating windows, contexts and surfaces, receiving input and events.
- GLEW
  - OpenGL Extension Wrangler Library, loads OpenGL extensions
- GLM
  - OpenGL Mathematics, OpenGL/GPU optimized math operations, handles things like quaternion to euler angles
  - Similar to C's Eigen Library
- ImGUI
  - Immediate Mode GUI for use with OpenGL

Bezier Curves

P_0, P_1, P_2 and P_3

```
glm::vec3 controlPoints[4] = {
    glm::vec3(0.0f, 0.0f, 0.1f),
    glm::vec3(0.1f,  0.1f, 0.2f),
    glm::vec3( 0.2f, 0.2f, 0.3f),
    glm::vec3( 0.3f,  0.3f, 0.4f)
};
```

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3,\ 0 \leq t \leq 1.$$

```glm::vec3 bezier(float scaler, const glm::vec3* controlPoints) {
    float scaler_coeff = 1.0f - scaler;
    float scaler_squared = scaler * scaler;
    float scaler_coeff_squared = scaler_coeff * scaler_coeff;
    float scaler_coeff_cubed = scaler_coeff_squared * scaler_coeff;
    float scaler_cubed = scaler_squared * scaler;

    glm::vec3 point = scaler_coeff_cubed * controlPoints[0]; // (1-t)^3 * P0
    point += 3 * scaler_coeff_squared * scaler * controlPoints[1];   // 3 * (1-t)^2 * t * P1
    point += 3 * scaler_coeff * scaler_squared * controlPoints[2];   // 3 * (1-t) * t^2 * P2
    point += scaler_cubed * controlPoints[3];          // t^3 * P3

    return point;
}
```

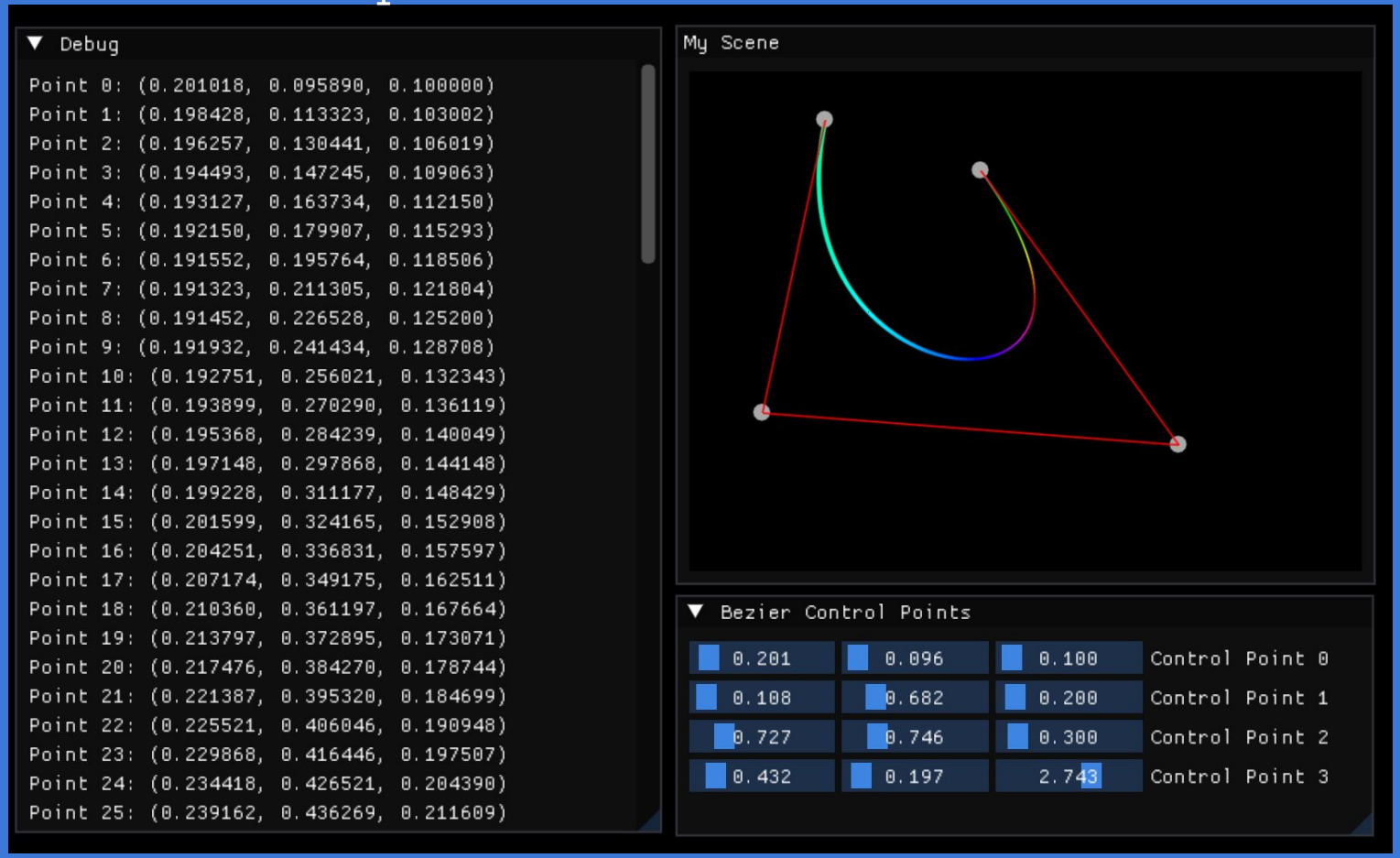
Generate curve by looping through points and calling previous function

```
std::vector<glm::vec3> generateBezierCurve(const glm::vec3* controlPoints, int numPoints) {
    std::vector<glm::vec3> curvePoints;
    for (int i = 0; i <= numPoints; ++i) {
        float t = (float)i / (float)numPoints;
        curvePoints.push_back(bezier(t, controlPoints));
    }
    return curvePoints;
}
```

Displaying the control points

```
void showBezierControlPoints() {
    ImGui::Begin("Bezier Control Points");

    for (int i = 0; i < 4; ++i) {
        ImGui::SliderFloat3(("Control Point " + std::to_string(i)).c_str(), &controlPoints[i].x, 0.0f, 4.0f);
    }

    ImGui::End();
}
```

OpenGL Cubic Bezier Curves



Quaternions

Euler to Quaternion conversion

$$
\begin{aligned}
\mathbf{q}_{\mathrm{lB}} &= \begin{bmatrix} \cos(\psi/2) \\ 0 \\ 0 \\ \sin(\psi/2) \end{bmatrix} \begin{bmatrix} \cos(\theta/2) \\ 0 \\ \sin(\theta/2) \\ 0 \end{bmatrix} \begin{bmatrix} \cos(\phi/2) \\ \sin(\phi/2) \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} \cos(\phi/2)\cos(\theta/2)\cos(\psi/2) + \sin(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \sin(\phi/2)\cos(\theta/2)\cos(\psi/2) - \cos(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2) - \sin(\phi/2)\sin(\theta/2)\cos(\psi/2) \end{bmatrix}
\end{aligned}
$$

```
// This is not in game format, it is in
Quaternion ToQuaternion(double roll, dou
{
    // Abbreviations for the various ang

    double cr = cos(roll * 0.5);
    double sr = sin(roll * 0.5);
    double cp = cos(pitch * 0.5);
    double sp = sin(pitch * 0.5);
    double cy = cos(yaw * 0.5);
    double sy = sin(yaw * 0.5);

    Quaternion q;
    q.w = cr * cp * cy + sr * sp * sy;
    q.x = sr * cp * cy - cr * sp * sy;
    q.y = cr * sp * cy + sr * cp * sy;
    q.z = cr * cp * sy - sr * sp * cy;

    return q;
}
```

Quaternion to Euler conversion

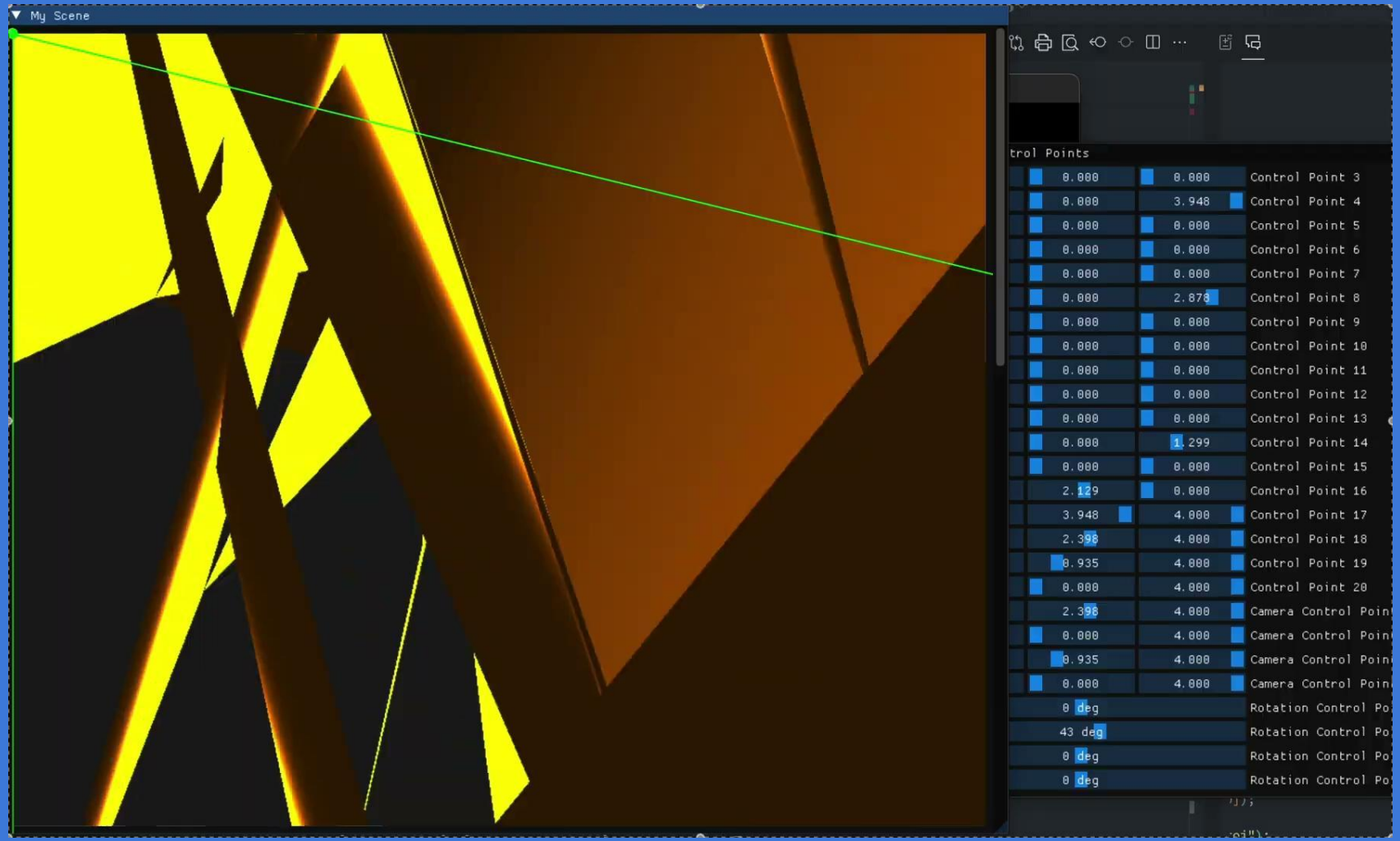
```cpp
struct Quaternion {
    double w, x, y, z;
};

struct EulerAngles {
    double roll, pitch, yaw;
};

// this implementation assumes normalized quaternion
// converts to Euler angles in 3-2-1 sequence
EulerAngles ToEulerAngles(Quaternion q) {
    EulerAngles angles;

    // roll (x-axis rotation)
    double sinr_cosp = 2 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y);
    angles.roll = std::atan2(sinr_cosp, cosr_cosp);

    // pitch (y-axis rotation)
    double sinp = std::sqrt(1 + 2 * (q.w * q.y - q.x * q.z));
    double cosp = std::sqrt(1 - 2 * (q.w * q.y - q.x * q.z));
    angles.pitch = 2 * std::atan2(sinp, cosp) - M_PI / 2;

    // yaw (z-axis rotation)
    double siny_cosp = 2 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z);
    angles.yaw = std::atan2(siny_cosp, cosy_cosp);

    return angles;
}
```

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}\left(2(q_w q_x + q_y q_z), 1 - 2(q_x^2 + q_y^2)\right) \\ -\pi/2 + 2\operatorname{atan2}\left(\sqrt{1 + 2(q_w q_y - q_x q_z)}, \sqrt{1 - 2(q_w q_y - q_x q_z)}\right) \\ \operatorname{atan2}\left(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2)\right) \end{bmatrix}$$

Quaternion Slerp

Provides a way to linear interpolate between two quaternion states based on time.

$$
\begin{aligned}
\operatorname{slerp}(q_0, q_1, t) &= q_0 (q_0^{-1} q_1)^t \\
&= q_1 (q_1^{-1} q_0)^{1-t} \\
&= (q_0 q_1^{-1})^{1-t} q_1 \\
&= (q_1 q_0^{-1})^t q_0
\end{aligned}
$$


My Scene
trol Points
0.000 0.000 Control Point 3
0.000 3.948 Control Point 4
0.000 0.000 Control Point 5
0.000 0.000 Control Point 6
0.000 0.000 Control Point 7
0.000 2.873 Control Point 8
0.000 0.000 Control Point 9
0.000 0.000 Control Point 10
0.000 0.000 Control Point 11
0.000 0.000 Control Point 12
0.000 0.000 Control Point 13
0.000 1.299 Control Point 14
0.000 0.000 Control Point 15
2.129 0.000 Control Point 16
3.948 4.000 Control Point 17
2.398 4.000 Control Point 18
0.935 4.000 Control Point 19
0.000 4.000 Control Point 20
2.398 4.000 Camera Control Poin
0.000 4.000 Camera Control Poin
0.935 4.000 Camera Control Poin
0.000 4.000 Camera Control Poin
0 deg Rotation Control Po
43 deg Rotation Control Po
0 deg Rotation Control Po
0 deg Rotation Control Po

Any Questions?