Curve Editor HW2

Student: Mengyu Li
Date: 2021/10/8

1. (30 points) **Euler Angle Splines.**

   Given Euler angle keyframe data of the form $\mathbf{p}_i = [x_i, y_i, z_i]^T$ (i = 0 to m), construct a piece-wise linear spline and a cubic Catmul-Rom spline comprised of Bezier curve segments that interpolates the $\mathbf{p}_i$ data points.

   - (10 points) **Linear Euler angle interpolation**

     Implement `AEulerLinearInterpolatorVec3::interpolateSegment()`

   - (20 points) **Cubic Euler angles interpolation**

     Implement `AEulerCubicInterpolatorVec3::interpolateSegment()`

     Implement `AEulerCubicInterpolatorVec3::computeControlPoints()`

All function in Euler Angle Splines implemented, both linear and cubic interpolation for the Euler angles.

2. (30 points) **Orientation Representations.**

   In this part of the assignment you need to implement conversions between 6 different orientation representations by filling in the details of the functions listed below.

   - (10 points) Functions to convert between Euler Angles and Rotation matrix representations:

     ```
     mat3::FromEulerAngles()
     mat3::ToEulerAngles()
     ```

   - (10 points) Functions to convert between Rotation matrix and quaternion representations:

     ```
     quat::ToRotation()
     quat::FromRotation()
     ```

   - (10 points) Functions to convert between quaternion and Axis/Angle representations:

     ```
     quat::ToAxisAngle()
     quat::FromAxisAngle()
     ```

All function in orientation representation implemented. Note for the quat::FromRotation, the algorithm is used as the one from Orientation Representation -Van Verth and Bishop.

3. (40 points) **Quaternion Splines.**

   The AnimationToolkit base code includes the class **ASplineQuat** whose interface is similar to the vec3 splines you developed in Part 1.

   - (15 points) **Linear quaternion interpolation**

     Implement `quat::Slerp()`

     Implement `ASplineQuat::getLinearValue()`

   - (25 points) **Cubic quaternion interpolation**

     Implement `quat::Scubic()`, `quat::SBisect()`, `quat::SDouble()`

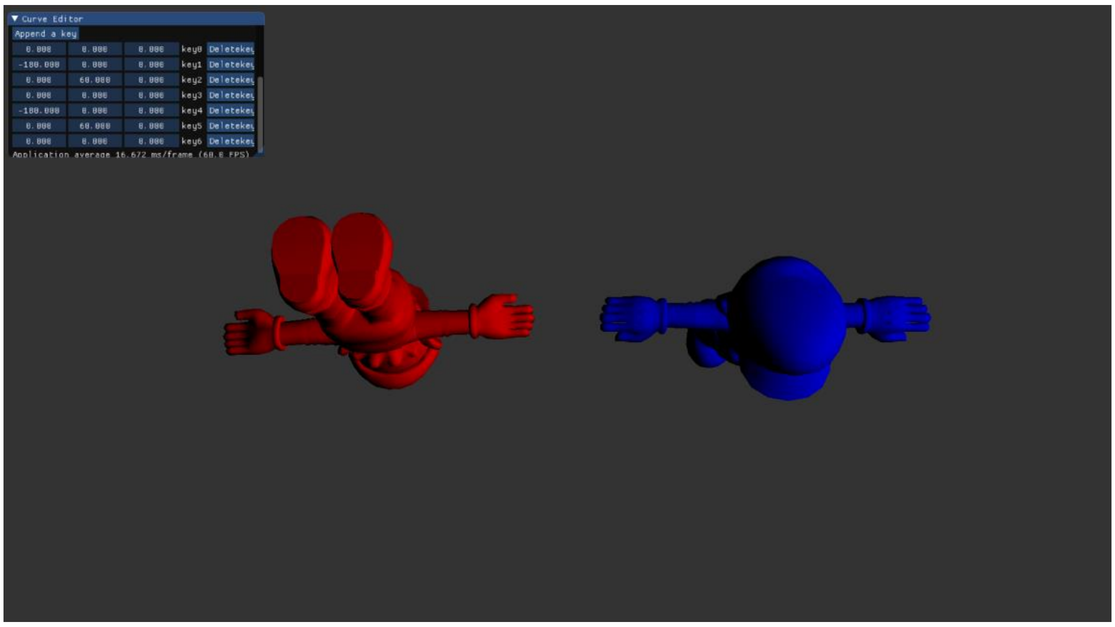
     Implement `ASplineQuat::getCubicValue()`

     Implement `ASplineQuat::computeControlPoints()`

All function implemented or quaternion Splines. Note some cases such as from start point angle (0, 0, 0) to next key anlge (-180, 0, 0) may have different path from the demo. After discussion with Professor Lane in office hour, the reasons are illustrated as following.

1. quat::FromRotation in the demo code is using algorithm from Orientation Representation -Van Verth and Bishop as well. However, it does not care for the orientation from cases such as 0 to 180 degrees. Thus, there will be sudden direction shift from key2 (-180, 0, 0) to key3 in the demo (which is not optimal in a cubic interpolation). In submission, the code takes consideration of previous orientation, so such direction bump will not occur from one key to another.

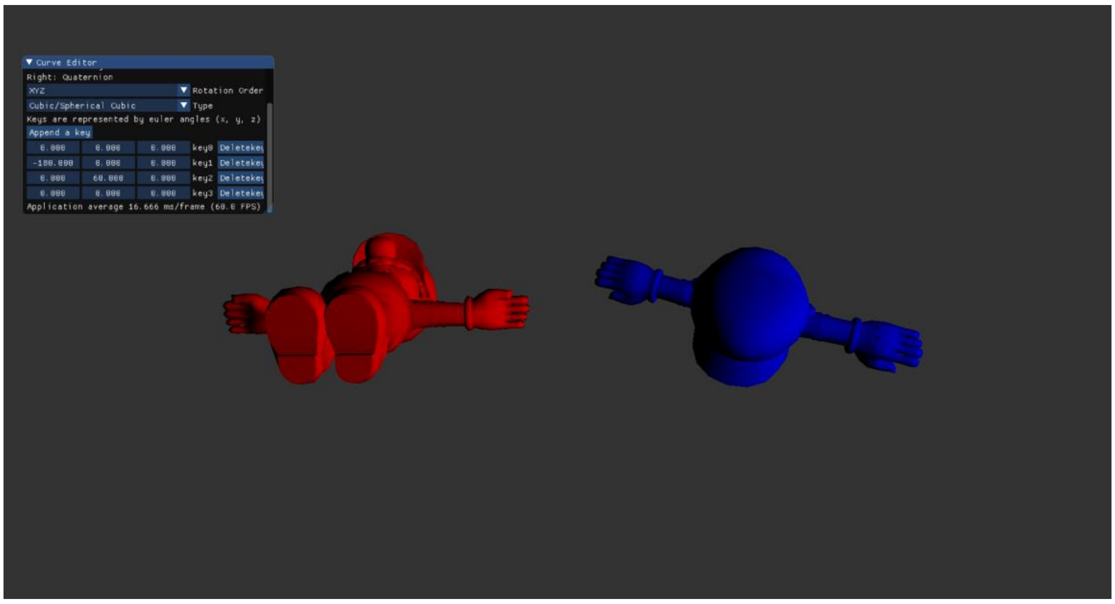
2. Also in the code submission, there will be obvious orientation tipping from cases such as $key0\ (0,0,0) \rightarrow key1(-180,0,0) \rightarrow keys(0,60,0)$, which does not occurs in the demo. The comparison is shown in the Figure 3.1a, Figure 3.1b.



0.000	0.000	0.000	key0	Deletekey
-180.000	0.000	0.000	key1	Deletekey
0.000	60.000	0.000	key2	Deletekey
0.000	0.000	0.000	key3	Deletekey
-180.000	0.000	0.000	key4	Deletekey
0.000	60.000	0.000	key5	Deletekey
0.000	0.000	0.000	key6	Deletekey

```
Application average 16.672 ms/frame (60.0 FPS)
```

Figure 3.1a: Demo code of $key0\ (0,0,0) \rightarrow key1(-180,0,0)$



Curve Editor

Right: Quaternion

XYZ — Rotation Order

Cubic/Spherical Cubic — Type

Keys are represented by euler angles (x, y, z)

Append a key

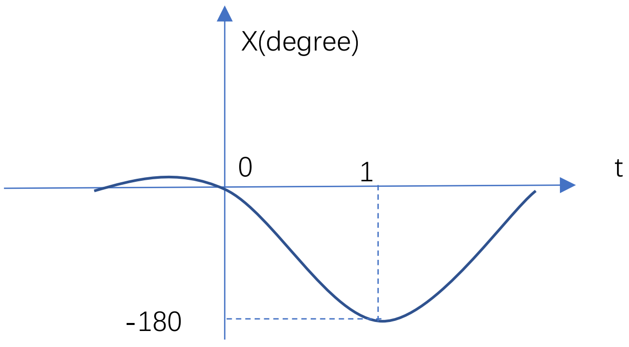
0.000	0.000	0.000	key0	Deletekey
-180.000	0.000	0.000	key1	Deletekey
0.000	60.000	0.000	key2	Deletekey
0.000	0.000	0.000	key3	Deletekey

Application average 16.666 ms/frame (60.0 FPS)

Figure 3.1b: Submitted code of $key0\ (0,0,0) \rightarrow key1(-180,0,0)$

Notice the tipping of the lateral orientation of the quaternion interpolation. The reason that occurs is because of the different ways of interpolating the edge cases at start segment and end segment. In demo code, it seems the code simply isolate the segment 0 and end segment as special cases and set the q_{-1} for start segment equals q_0 and q_2 for end segment equals q_1. Since the two cases are isolated, it will be just a simple cubic interpolation with only too keys $key0\ (0,0,0) \rightarrow key1(-180,0,0)$. Thus, there is no tipping and causes bump as well.

However, in the submitted code, the cases at start and end segment are considered as extended segments with additional neighbor keys, where $q_{-1} = SDouble\ (q_1, q_0)$, $q_2 = SDouble\ (q_0, q_1)$. Thus, it will be a full interpolation of $q_{-1} \rightarrow q_0 \rightarrow q_1 \rightarrow q_2$. In this case it will be $key_{-1}\ (0,0,0) \rightarrow key_0(0,0,0) \rightarrow key_1\ (-180,0,0) \rightarrow key_2(0,60,0)$
If we illustrate in graph, it will be as



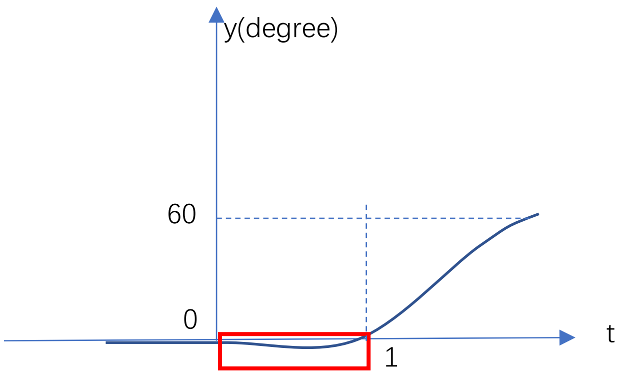
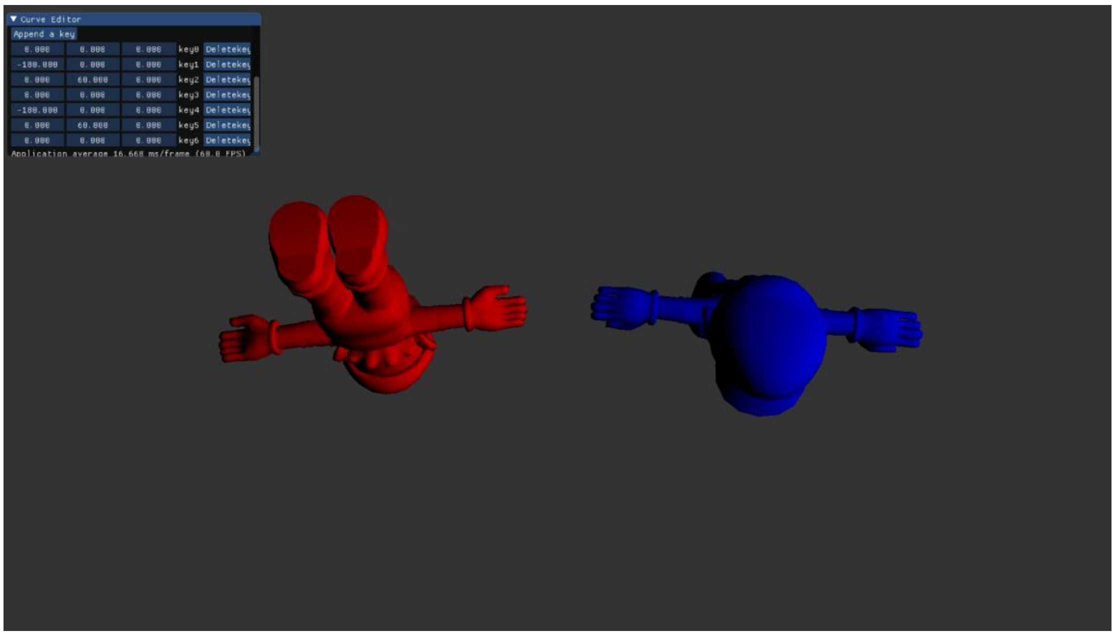


Figure 3.2: Curve demonstration of interpolation

Notice the red squared area, where is the tipping occurs.,

Check:
If the explanation is valid, it means tipping can also occur in demo code. We experiment with a segment sequence of $key_3\ (0, 0, 0) \rightarrow key_4(-180, 0, 0) \rightarrow key_5\ (0, 60, 0) \rightarrow key_6\ (0, 0, 0)$ in a middle of a continuous splines. As expected, the tipping occurs as shown in Figure 3.3.



0.000	0.000	0.000	key0	Deletekey
-180.000	0.000	0.000	key1	Deletekey
0.000	60.000	0.000	key2	Deletekey
0.000	0.000	0.000	key3	Deletekey
-180.000	0.000	0.000	key4	Deletekey
0.000	60.000	0.000	key5	Deletekey
0.000	0.000	0.000	key6	Deletekey

```
Application average 16.668 ms/frame (60.0 FPS)
```

Figure 3.3: Tipping Also occurs in demo.
Due to the orientation of camera, it's the same amount of tipping as the submitted code but just seen from different angle.

4. *(optional 30 points)* **Cubic B-Spline**. Implement a cubic spline curve with natural end point conditions using a B-Spline formulation. This can be accomplished by implementing the following functions:

- `ABSplineInterpolatorVec3::computeControlPoints()`
- `ABSplineInterpolatorVec3::interpolateSegment()`

Feel free to add helper functions to the `ABSplineInterpolatorVec3` class to make the implementation easier.

Cubic B-Spline fully implemented; it shows the same result as Hermite curve as expected.