

CS 408 Assignment 3

Wyatt Drew

Part 1 Animation algorithm analysis

```
00a  $m = \|A\|$ 
00b for  $a = 0, 1, 2, \dots, m - 1$            // for all attributes
00c    $p_a = K.head_a$ 
00d    $n_a = K.head_a$ 
00e end for
00f  $t = 0$ 

01 while  $t \leq T$                            // draw all frames over time

02   for  $a = 0, 1, 2, \dots, m - 1$            // for all attributes interpolate values

03     while  $t > n_a \rightarrow time$ 

04a        $p_a = n_a$ 
04b        $n_a = n_a \rightarrow next$ 
04c     end while

05      $u = (t - p_a \rightarrow time) / (n_a \rightarrow time - p_a \rightarrow time)$ 
06      $A_a = (1 - u) * (p_a \rightarrow value) + u * (n_a \rightarrow value)$ 
07   end for

08   draw object with values  $A_0, A_1, A_2, \dots, A_{m-1}$ 

09a    $\Delta t = getElapsedTime()$  // The time since the last frame
09b    $t = t + \Delta t$ 

10 end while
```

Real time animation can have lag. As such it is important to make sure progress is a factor of time rather than framerate. This gives the impression of smooth motion. I have adjusted several lines (marked in white above) to consider time rather than framerate. Thus, we are storing time rather than frames in the linked list.

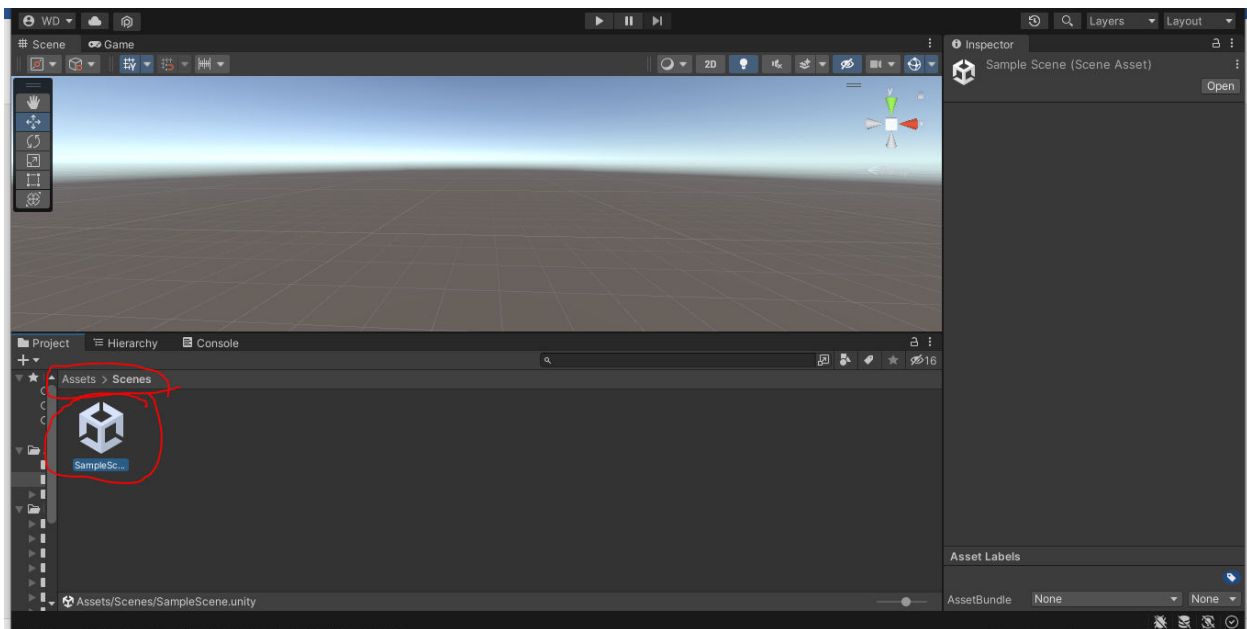
Part 2 Uniform Cubic B-Spline Animation

Compatibility (see Setup)

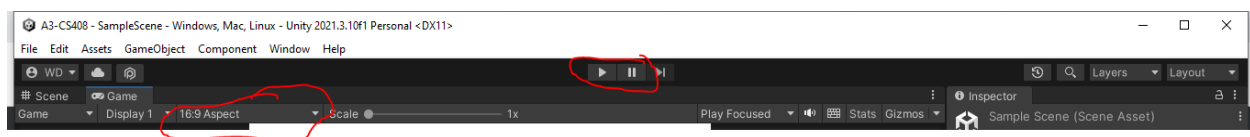
1. Unity Editor
2. Windows
3. WebGL

Unity Setup:

1. Download and install Unity.
 - a. Here you can sign up for either a student or personal account to download Unity for free.
Download: <https://store.unity.com/#plans-individual>
2. Download CS-408 A3 (my assignment) from Github.
3. Open up the project, under the project tab you will see the "SampleScene" scene. Click it to open the scene.



4. From there click the game tab, set it to 16:9 Aspect ratio, and click play



Windows Setup:

1. Navigate to the Windows build folder under A3-CS408.
2. Click on A3-CS408.exe to launch the application.

WebGL Setup:

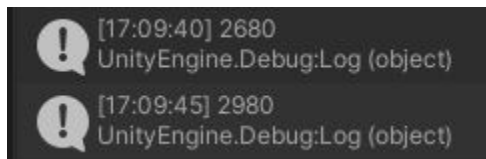
1. Visit <https://wyatt-drew.github.io/Uniform-Cubic-B-Spline/index.html>
2. Build files can be found in the WebGL build folder under A3-CS408 if you want them.

Proof of Correctness (Time)

Observe that the animation takes exactly 5 seconds (300 frames). Here (CreateSpline.cs) for linear movement I take the time that passed and divide it by 5 seconds then mod by 1 second to get animation progress between 0 and 1. In this way progress through a given movement method is always linked to time itself.

```
//Purpose: Changes progress value to represent how far along the path we should be.
void LinearProgress()
{
    progress = (float)((Time.realtimeSinceStartupAsDouble - startTime) / 5f) % 1f;
}
```

Observe that the values for sine and parabolic ease in/out total to 1 (not shown) and therefore do not change this relation. I included a console log function to verify this (only visible in the unity editor). Essentially every time the animation resets it logs the frame count. This also happens when you change the movement mode because I implemented it by just measuring delta animation progress.



```
[17:09:40] 2680
UnityEngine.Debug:Log (object)

[17:09:45] 2980
UnityEngine.Debug:Log (object)
```

Proof of Correctness (Movement Control)

getFrame() can be found in CreateSpline.cs. Observe that for any progress from 0 to 1 it is translated to the corresponding point which best matches the cumulative arc distance and then linearly interpolated between points using u . In this way progress is directly linked to distance traveled (fraction of arc distance = fraction of progress). I can put as many points as I want so I wasn't concerned about rounding error here. After all, with 300 frames and 800 points we already skip about 3 points per frame and if it was an issue I can just add as many points as I want. Therefore, interpolating exactly along the curve is a total waste of computations and has no visible effect.

Additionally, since the sin and parabolic equations are close translations from class/the textbook these are unlikely to have any errors in them when calculating progress.

```
int getFrame()
{
    float curDistance = progress * cumulativeLength[cumulativeLength.Count - 1];
    for (int i = 0; i < maxSteps * (size - 3) + 1; i++)
    {
        if (curDistance == cumulativeLength[i])
        {
            u = 0f; //exactly matches a point
            return i;
            //it can never be beyond the last point so we are safe from core dumps on the next line.
        } else if (curDistance > cumulativeLength[i] && curDistance < cumulativeLength[i + 1])
        {
            u = (curDistance - cumulativeLength[i]) / (cumulativeLength[i + 1] - cumulativeLength[i]); //step Distance traveled/total step distance
            return i;
        }
    }
    Debug.Log("error");
    return 0; //This will never happen
}
```

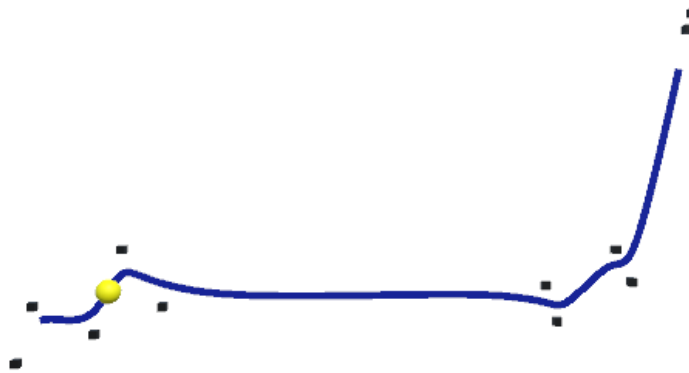
Creative Features

1. Labels

The animation is clearly labeled. As you change movement style the description at the top will change appropriately.

Uniform Cubic B-Spline

Linear Movement



Controls

b: Linear Movement
c: Sinusoidal Ease-in / Ease-out
d: Parabolic Ease-in / Ease-out

Location of Files: CreateSpline.cs: Start(), getInput()

2. Prefab spline to save computations

My code dynamically makes the spline currently by just making a spline along whatever path was calculated (thus if the control points changed it would still work) but if you toggle drawline to false at the top of CreateSpline.cs it saves some start up time by just spawning in the prefab version.

Location of Files: CreateSpline.cs: Start()

3. Unity Editor, Windows, and WebGL support

Location of Files: A3-CS408

Works Cited

1. The B-Spline algorithm was based on Alain Maubert's lecture slides.
2. The algorithms for sinusoidal and parabolic ease in/out were taken from a combination of Alain Maubert's lecture slides and Computer animation: Algorithms and techniques.

Source: Parent, R. (2012). *Computer animation: Algorithms and techniques* (Third Edition). Elsevier.