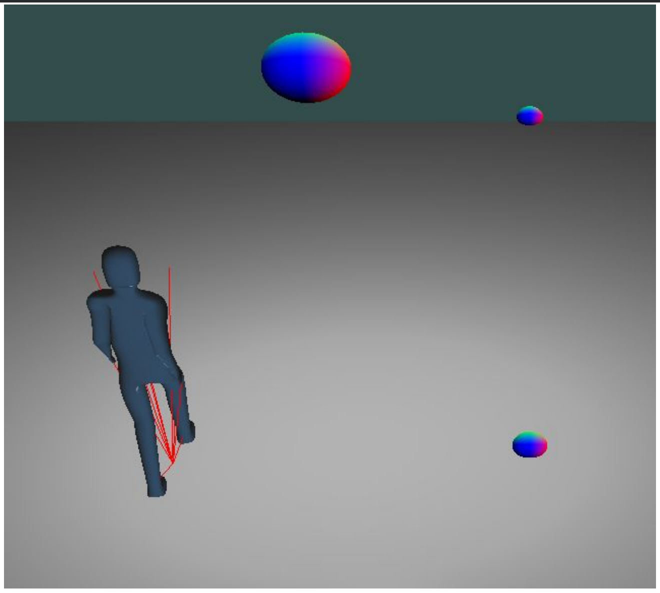
Cody Morgan
CS460
Project 2
22 Oct 2020

Curve Generation

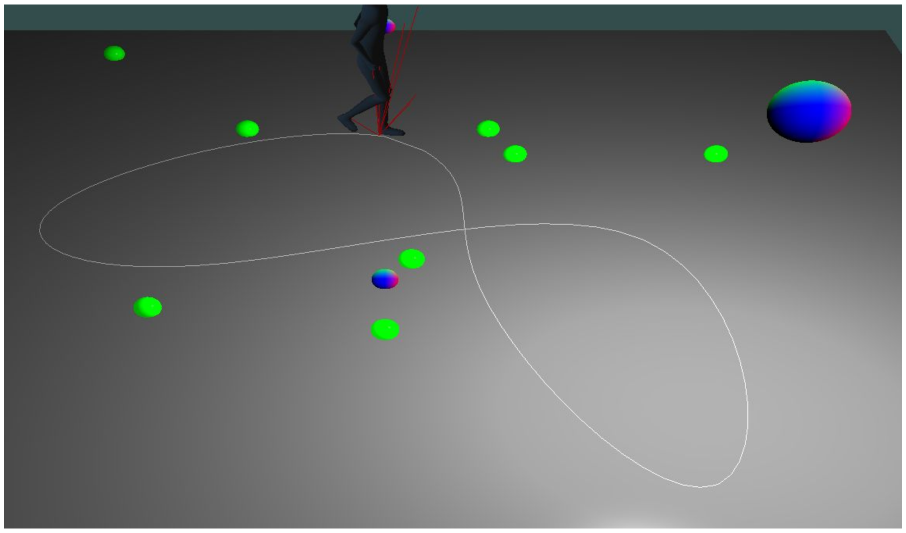
The first thing that I did for this project was to get the model moving in a circle with the correct heading and minimal foot slipping. This was easy enough to do using the formula for a circle. (Main.cpp SceneUpdate, now removed)

```
model->setPosition(vec3(-radius * glm::cos(t), 0 , -radius *
glm::sin(t)));
model->rotate(((dt * speed) * 360.0) / glm::two_pi<double>());
```



My thought was essentially to take a b-spline from my MAT 300 class, stuff it into this engine and see if it worked (my MAT 300 code is in C#). This formed the basis of my Curve class (Curve.h), though I did add several functions such as finding the tangent. I fixed my curve in space and it is not updated in the simulation (though it could be if we cared about changing the curve).

Cody Morgan
CS460
Project 2
22 Oct 2020



Now, that’s a sweet curve. I instantiated green balls to mark the control points - to my surprise the whole curve was backwards as openGL points Z to the rear of the camera and Unity points towards the camera. A simple negative solved this giving me the above curve. This curve is generated using the DeBoor algorithm (Curve.cpp 158)

```
vec3 Curve::deBoor(int k, int i, float t)
{
  if (k == 0 || i == 0)
  {
    return controlPoints_[i];
  }
  else
  {
    float c0 = (knot[i + dimension_ - (k - 1)] - t) / (knot[i + 
dimension_ - (k - 1)] - knot[i]);
    float c1 = (t - knot[i]) / (knot[i + dimension_ - (k - 1)] - 
knot[i]);
    return c0 * deBoor(k - 1, i - 1, t) + c1 * deBoor(k - 1, i, t);
  }
}
```

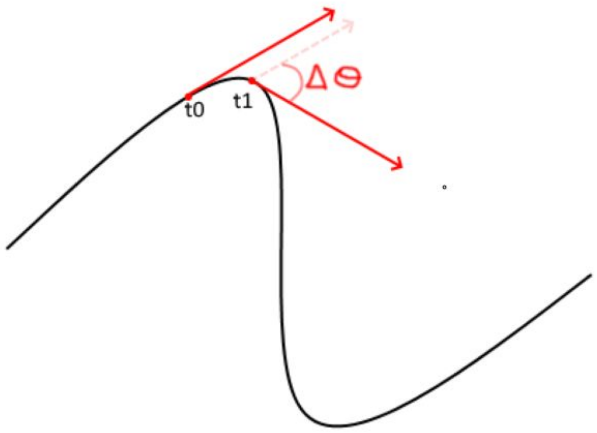
Cody Morgan
CS460
Project 2
22 Oct 2020

Model Motion

With the curve now drawing I sought to solve the same problems as I had done with the circle motion - namely he needed to follow the path and rotate accordingly. Getting him to follow the path was easy, as it turns out, just evaluate the curve. This gave me a nice "moonwalking" model. (Main.cpp 262)

```
model->setPosition(curve.evaluate(t));
```

Getting the model to rotate correctly was a bit more complex than with the circle rotation. The idea is to match the model's rotation with the tangent vector of the curve.



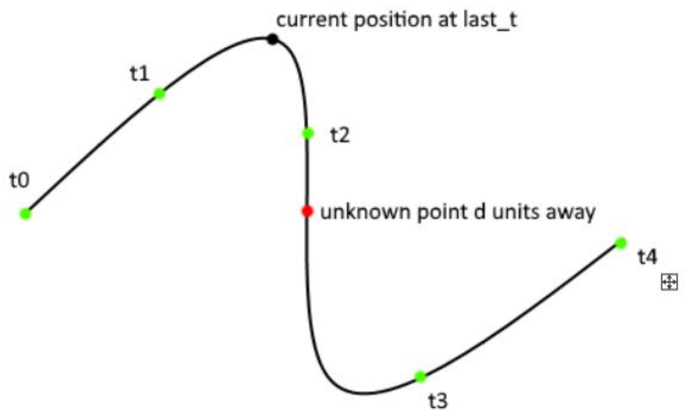
This actually worked pretty well - though I did have to adjust this to fit in glm's rotation scheme as it rotates by x degrees per call - this would be total rotation. (Curve.cpp 48)

```
  auto slope = tangent(t);
  if(slope.x == 0)
  {
    return slope.z > 0 ? 90.0f : 270.0f;
  }

  float degrees = glm::degrees(glm::atan(-slope.z , slope.x));

  return degrees;
```

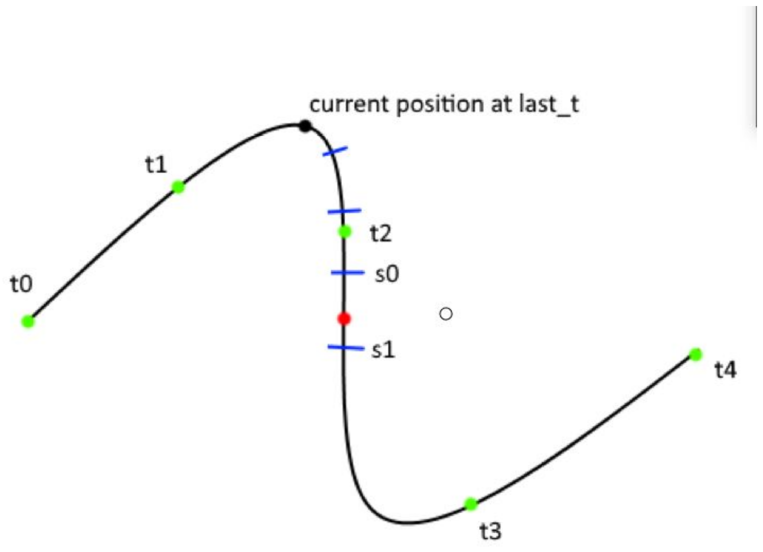
(Main.cpp 264)

```
float thisRotation = curve.getRotation(t);
rotateBy = thisRotation - lastRotation;
model->rotate(rotateBy);
lastRotation = thisRotation;
```

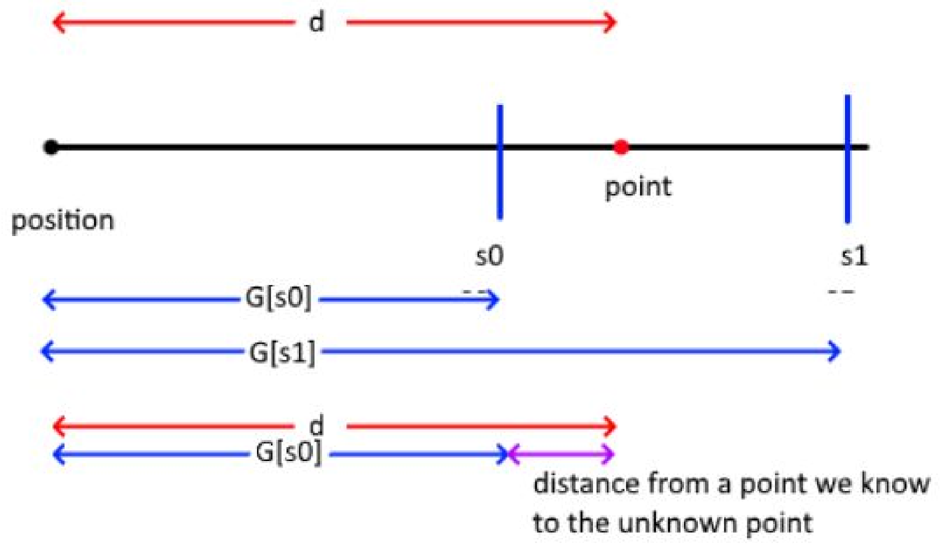
The next issue was the foot slipping. To solve this I would need constant velocity around the curve. I did this, following the notes in class, by calculating the inverse of the distance I want to go to give me the t value that gets me there. Essentially I know where I am and I know how far I want to go - but I don't know the t value to get me there.



We know that the known point is at some t > last_t and by moving in small steps we can find a t value on either side of this point



Once we know of a point on either side we can use a ratio to evaluate approximately what that t should be. I found that this approximation was pretty good at only a few iterations. I tried 10 iterations at first but it was too computationally expensive - this used about 3 evaluations and was 0.07% farther than the actual distance.



If we take a ratio of the purple line over the distance between s0 and s1 we can determine approximately the t value at the unknown point.

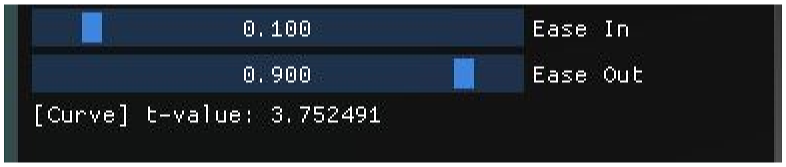
(Curve.cpp 92)

```
float Curve::adjustToNearestT(glm::vec3 pos, float d)
{
  std::map<float,float> G;
  float step = 1.0f / (2 << 11);  float t = last_t;
  float dist = 0;                 float s0;
  float s1;
  while (dist < d)
  {
    if((t + step) >= (t_max - 0.1f))
      t= t_min;
    s0 = t;
    s1 = t + step;
    t = s1;
    auto p0 = evaluate(s0);
    auto p1 = evaluate(s1);
    G[s1] = G[s0] + glm::distance(p1, p0);
    dist = glm::distance(pos, p1);
  }
  return s;
}
```

Doing so allows us to have more uniform velocity around the curves.

Ease In / Ease Out

For this feature, I first created two sliders to control when the ease would be activated. The ease in activates from t_0 to 0.1 * range of t (in my case 0.1 * 7 = $t_{0.7}$). The same for ease out (0.9 * 7 = $t_{6.3}$ to t_7).


```
0.100        Ease In
0.900        Ease Out
[Curve] t-value: 3.752491
```

Cody Morgan
CS460
Project 2
22 Oct 2020

Referencing the notes I implemented different velocities for the eases. (Main.cpp 523)

```
      if (t < t1)
      {
        mode = "EaseIn";
        velocity = t * (originalVelocity / t1);
        model->skeleton.animation.animationSpeed = velocity /
originalVelocity;
      }
      else if (t > t2)
      {
        mode = "EaseOut";
        velocity = (t3 - t) * (originalVelocity / (t3 - t2));
        model->skeleton.animation.animationSpeed = velocity /
originalVelocity;
      }
      else
      {
        velocity = originalVelocity;
        if (std::abs(rotateBy) <= 2)
        {
          model->skeleton.animation.animationSpeed = std::max(0.8f,
1.0f - std::abs(rotateBy / 4));
        }
      }
```

Video

I have uploaded my video and it can be found at the following URL https://youtu.be/zWud_kK94do . I found that there is still a little foot slipping around the tight corners - I think that the curve may possibly be too tight for him to reasonably turn without slipping - something that I may address at a later point.