Keith M. Programming

Virtual Reality Wizard and Nerdologist.

Math Magician – Path Smoothing with Chaikin

Posted on May 29, 2018June 19, 2019 by KeithM
There are a lot of algorithms out there for making something follow a path. I should know, I've posted a couple. The thing is, a lot of times I find myself following a path just fine, but trying to make that path smooth is pretty frustrating. Some methods might take too long to execute, or don't really work in realtime, or are just *waaay* too beefy for what I'm trying to do. So, I found a pretty simple algorithm for your Game-Dev-On-The-Go to help you find the way.

Alright, here's the setting:

You have an enemy, he's flying in space. He flies a basic patrol that selects a random path from a collection of waypoints. You generate your path, and it returns a List of Vector3s. Watching your enemy fly a path filled with extreme <90 degree turns looks dumb. Real dumb. You start to consider taking a job in construction, but wait; there's a simple solution: what if you could just cut out those sharp corners?

Well, that's when we call in **Chaikin**.

Chaikin It Easy

In the far-off future of 1974, at the University of Utah, George Chaikin gave a lecture, providing an algorithm for taking a set of points and creating a curve (http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm/Chaikins-Algorithm.htmlhttp://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm/Chaikins-Algorithm.html). in those days, there wasn't a lot going for simple curve calculations, and Chaikin's was one of the first to implement corner-cutting (or refinement). This algorithm was largely lost and forgotten by the mathematics community in favor of B-spline curves and so-on, but something can be said for it's ease of implementation and execution.

So, the problem that was proposed was, with a set of connected points (AKA a "Control Polygon"), how could someone efficiently smooth the path? How can someone turn a square into a circle. Chaikin's method was a simple concept – **cut off the corners**. If you have point 1 (P_1 {0, 0}) going to point 2 (P_2 {0, 1}) which, itself, goes to point 3 (P_3 {1, 1}), you have yourself a right angle. Chaikin's method would

travel 75% between P_1 and P_2 (Q_1 {0, 0.75}) and 25% between P_2 and P_3 (R_1 {0.25, 1}) and connect them (leaving P_2 out completely), you've rounded that corner! "*But wait*," your disembodied voice is complaining, "*if I ran that method on every corner of the square, this will just make an octagon!*" That is true. And, certainly, this method is only good for making more complex polygons, not true circles or splines. But what we can do is run this method on out path *multiple times* to continuously smooth out the path! Nifty.

Smoothed Over

You have the concept, but here's how to make it happen.

The algorithm, mathematically, looks as such:

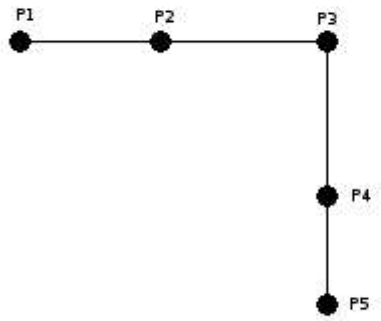
$$\mathbf{Q_i = \tfrac{3}{4}P_i + \tfrac{1}{4}P_{i+1}}$$
$$\mathbf{R_i = \tfrac{1}{4}P_i + \tfrac{3}{4}P_{i+1}}$$

See? There's a reason I used those letters above! Anyway, this algorithm is like a simplified version of linear interpolation (https://keithmaggio.wordpress.com/2011/02/15/math-magician-lerp-slerp-and-nlerp/) between 2 points, then connecting the results.

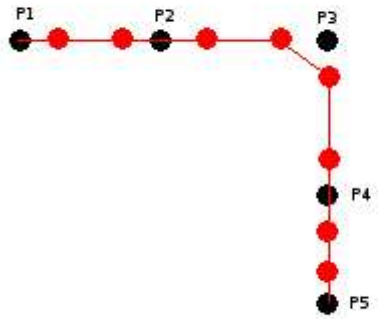
Now, that being said, the code for such a smoothing algorithm isn't that hard to figure out from the example. So, tell you what: I'm not going to write it. Nope. Not gonna. Why? Because it'd be wrong and horribly inefficient, that's why!

Chaikin's Method is simple, but there's a fundamental flaw: It will *always* double the size of your path, regardless of point positioning. If you had a path of 3 points going in a straight line, it will make a path of 6 points going in a straight line!

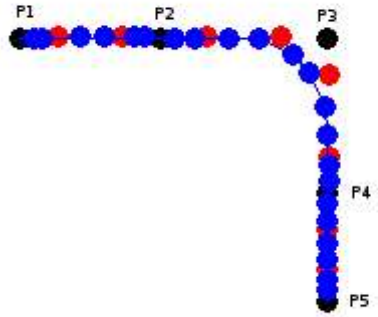
Need a visual? Here, look at this: Imagine you have a path of 5 points – 3 going to the right, and 2 going down.



OK! Good. Now, Chaikin will run over each point in the path, and create a new point between 25% and a point 75% of the way between points.



Now, that was just 1 pass. Fun-fact: **Chaikin's Method is designed to be run in multiple passes.** That means your path gets doubled. **Again.**



Yeah. That's nasty. So we're going to modify it.

I'll be implementing 2 things: **Angle-Limits** and **Outlier Rejection**. You are free to mix-and-match these in any way that you want. I figured I'd implement them both to give you at home an idea of how you can work this into your own path-refinement-method.

Angle-Limits

OK, Chaikin gets a bit carried away. It's a math algorithm that's only doing 2 steps. We need to step it up. We need Chaikin to only refine sharp corners. If a path is a straight line, it needs to stay a straight line. If a path is doing some sharp lefts, we need to smooth. So, here's what we're doing:

In our method, looping over the path, we're looking at our current point (C), our previous point (P), and our next point (N). What we want to know is the difference, in degrees, between traveling from P-to-C, and traveling C-to-N. We subtract P from C to get the current heading (cH), and subtract C from N to get our next heading (nH). It then comes down to a simple Angle Between Vectors check to see if we're

within a particular range (Say, about 30 degrees?), and if we are, then only put into our new path the C, and we don't run Chaikin's. This will keep our straight lines straight, our small curves small, and our big turns smooth.

Outlier Rejection

There's a reason I'm leaning into Chaikin's Method; I recently had to use it on a large collection of path data that I needed to "play back" in an application I wrote. Problem was that some of the data was a bit… scrambled. Maybe there was a glitch, or the recording application got confused, or maybe it was aliens — whatever it was, it messed up the path, and had some points suddenly making a sharp left, then turn quickly back, and some go completely all over the place.

I decided that the best way to fix this was to not try to manually fix the CSV myself, but to make my application smart enough to figure out what to do in case of bad data.

Using the above algorithm for Angle Limits, I checked to see if the angle was greater than an extreme range (Greater than 90 degrees) and then threw that point right the hell out. Smoothing that point would've only made some sort of weird loop-back, or some crazy break-neck turn. That data has no *point* in being there (get it?).

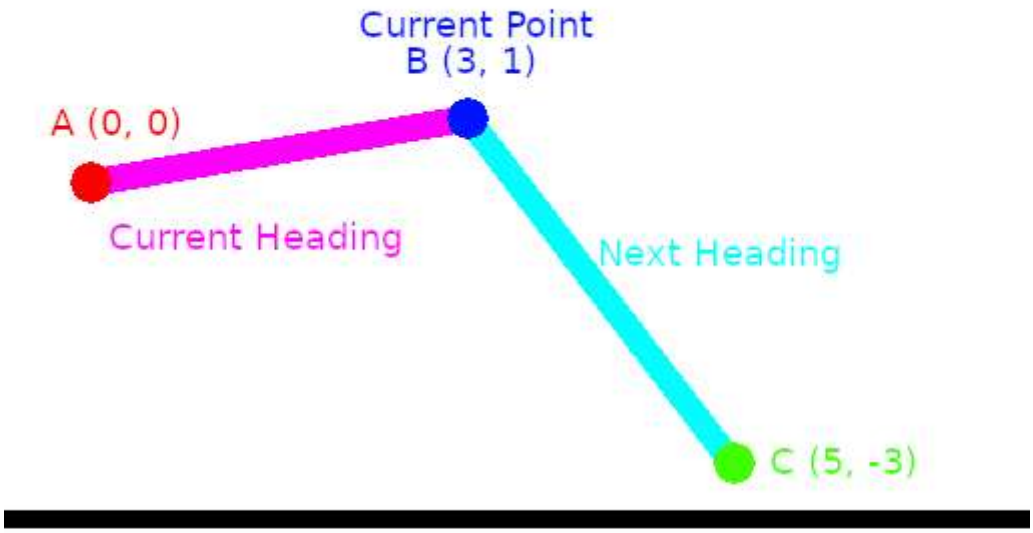
With that, my refinement algorithm is complete, and is listed here:

```
VectorList RefinePath(VectorList path)
{
  VectorList ret = new VectorList();
  VectorQueue procPath = new VectorQueue(path);
  ret.Add(path[0]);

  Vector3 curPoint, prevPoint, nextPoint,
          currentHeading, nextHeading,
          pointQ, pointR;
  float angle;
  prevPoint = procPath.Dequeue();
  while(procPath.Count > 1)
  {
    curPoint = procPath.Dequeue();
    nextPoint = procPath.First();
    currentHeading = (curPoint - prevPoint).normalized;
    nextHeading = (nextPoint - curPoint).normalized;
    angle = Vector3.Angle(currentHeading, nextHeading);
    if (angle >= 30)
    {
      if (angle >= 90)
      {
        procPath.Dequeue();
        prevPoint = curPoint;
        continue;
      }
      pointQ = (0.75f * curPoint) + (0.25f * nextPoint);
      pointR= (0.25f * curPoint) + (0.75f * nextPoint);
      ret.Add(pointQ);
      ret.Add(pointR);
      prevPoint = pointR;
    }
    else
    {
      // Paranoid check.
      if(ret.Contains(curPoint) == false)
          ret.Add(curPoint);
      prevPoint = curPoint;
    }

  }

  // Make sure we get home.
  if(ret.Contains(path[path.Count - 1]) == false)
    ret.Add(path[path.Count - 1]);
```

```
    return ret;
}
```

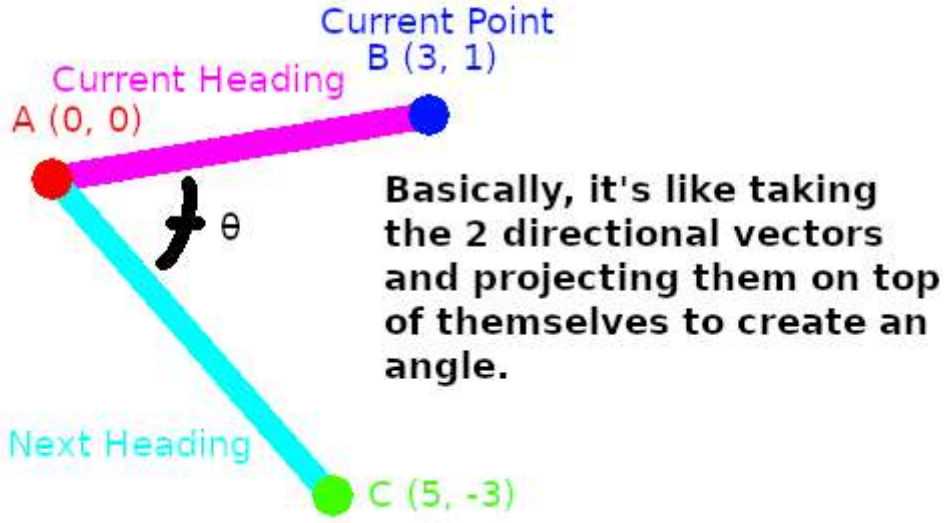
Lets break this down:

- Lets look at the first line: `VectorList RefinePath(VectorList path)`. Let me start by saying `VectorList` and `VectorQueue` are just a List/Queue of `Vector3`s. This was originally written for Unity in C#, but, since WordPress can't handle triangle brackets in HTML, I had to make this pseduocode. The idea here is that you pass in your path, and it outputs a new path with the corners cut.
- Next, we declare our output path, `ret`, and a `VectorQueue` called `procPath`. This is what we'll be "processing" when we loop over the path elements, dequeue-ing path points along the way. We also add the first node to our output, because Chaikin's has a nack for shortening the path.
- Now we declare all the variables for our loop (we *could* define them inside the while loop, but, if we're going to do that, why not just start littering and urinating in public?!). We also set our `prevPoint` to the first point in our path by pulling it from our `procQueue`. There's no reason to check the angle between the first point and the second, right?
- And thus, be begin our loop. We're going to loop through our `procPath` queue until there is 1 item left in it (we can't go all the way to the end because our algorithm needs to check the next point from our current point, and there's nothing left after the end). We then initialize our variables: Dequeue to get our `curPoint`, peek at the first item in the `procQueue` to get the `nextPoint`, then subtract the `prevPoint` from the `curPoint` to get our `currentHeading`, and then subtract the `curPoint` from the `nextPoint` to get the `nextHeading`.
- The next step is the important part of our Angle Limits and Outlier Rejection. We need to check the angle between our `currentHeading` and our `nextHeading`. Here's a diagram to show basically what we've done up to this point:



Angle Between Vectors:

$$\cos\theta = (u \cdot v) / (||u|| \cdot ||v||)$$



- Now, this next if check is our **Angle Limit** check. If the angle between `currentHeading` and `nextHeading` is greater than 30 degrees, then we need to curve the angle. Inside this block is where we'll perform either our Outlier Rejection or Chaikin's method.
- This next part is our actual **Outlier Rejection**. We found that our angle is over 30 degrees, now we're checking to see if we're over 90 degrees. In the simulation I was making at the time, there would have been no way in the data I was reading that we would be making a 90 degree turn, ever. What we do here is throw out the next point, set our previous point to our current point, and continue to the next point. Done.
- After all that, it's time for **Chaikin's Method**. As mentioned before, we create our Q and R points, creating our curve. We add them to our return list, and set our previous point to R, and move on.

- In this else block (continuing from the Angle Limit check), we've determined that the angle between `currentHeading` and `nextHeading` is less than 30 degrees, so we don't need to run Chaikin's method, so, simply, check to make sure this point isn't in the return path (paranoid check), and then just add the point to the return path.
- Now that our loop is done, we finish up by adding the last point into the return path. We do this, because Chaikin's tends to shorten the path if the Q point is generated from the last point in the path. So, just to be safe, we add it back to complete the path.

Ladies and gents (and the Technicolor rainbow in between), that's it. That's a super simple way to refine your path. Just call this function a few times and you'll get yourself a fine curve without too many added "fluff" points. As mentioned before, this algorithm gave way to B-Splines and such, which are more efficient in larger systems (or, depending on your project, less efficient). This is here for anyone looking to do quick-and-dirty path refinement. So, good luck, use it well, and if you have any improvements or modifications, please post it below. I'd love to see 'em!

Posted in Algorithms, Code, Math, Programming, UncategorizedTagged chaikin, Linear Algebra, Math, path, path smoothing, smooth, spline, Vector Math

---