Joon Ki Hong & Krista Patel

Project 4

CS 3451

November 6, 2012

Velocity Field

We wanted the particles to behave so that particles closer to the curve would move faster and closer to the trajectory of the curve. To do this, we based the particle's velocity from the distance between the particle and the closest point on the curve. We found the distance by calculating the absolute value of the dot product between the particle and the closest point on the curve from the particle. After we calculated the distance, we made the function of the distance correspond to the cos decay. To do this, we mapped the points [0, pi/2]. Then, we used this value as the theta value in the cos^2(theta) equation. This gave a value between [0,1]. To increase the velocity of the particles, we multiplied this scalar value by 2\*velocity, rather than just velocity. Choosing the closest point on the curve to define the particles’ velocity however limited the way we could define our curve. If the curve crossed paths with itself or almost crossed itself then the motions of the particles may not necessarily follow the entire path of the curve since based on its distance, it may choose the other portion of the curve which almost or does cross itself. This may have adverse behaviors in certain curve formations. For example, if you the curve was a tight loop and the particle was generated in the middle of the loop, the algorithm would try to choose the closest point on the curve. However, it may become “stuck” in that it’s choosing points all over the curve in a somewhat cyclic behavior.

Particle Generator

To create particles on the surface of the particle generator, we generated a random angle value between 0 and 360. Based on this angle we created x, y, and z points such that:

a = x + (radius+r)\*cos(angle)

b = y + (radius+r)\*sin(angle)

c = z + (radius+r)\*cos(angle),

where x, y, and z are the coordinates of the center of the particle generator, radius is the radius of the particle generator, and r is the radius of a particle. We used radius+r in our calculation to ensure that the generated particles were created on the surface of the particle, rather than intersecting with the surface of the particle generator.

Collision Detection and Reflection

Our collision detection algorithm differed from the one mention by our professor. Instead of solving for the roots of a quadratic equation to determine the smallest time, (which we implemented initially), we used a simple but slightly less accurate for loop from [0,T] in very small increments of to the 10^-4th order. For every pair of particles, we looped through this range and checked to see if the distance between the two centers of the particles was less than the radius of the first particle plus the radius of the second particle. This is slightly less accurate in that we don’t necessarily find the exact point in time they collide. We merely increment our time value and see if the new positions of the two particles result in a collision. We kept a array outside of this loop which keeps track of the smallest time s and indices of the two particles. If the loop finds a smaller value, we replace the one currently existing in our array. Then once we have looped through all pairs, we return the array with the smallest time s and indices. If the array is null, then there was no collision within the inter-frame [0,T]. If it’s not then we have a collision. All of this is abstracted into a method called computeS. If we find a collision, we calculate the reflection vectors of the two particles that collided and then advance all particles by t. We then subtract t from T and start the loop again. The while loop will break if we don’t find a collision within the current inter-frame.

Our initial implementation of computing collisions involved solving for the roots of a quadratic equation. We find the discriminant of this equation and determine from there whether or not we have a collision. We then filtered out any results that were negative, non-numbers, or larger than the other positive root if one existed. However, we had problems with this implementation since our roots almost always came out to 0. We decided to change up our algorithm to a more simple for loop to see a general estimation of the roots. However, this implementation gave us the same results. We then spent a lot of time debugging and concluded late into the development stage that we had issues with shallow copies of objects and pointers which both caused the false roots and reflection vectors. We believe our current implementation calculates collisions and reflections correctly to a certain degree. However, implementing inter-frames somehow incurs an eventual infinite while loop. Our program runs fine for the first minute and then goes into a while loop. We believe this may be due to collisions that give smallest s values of 0. Our outer decrementing of T=T-t would then result in the same T for the next loop. We tried to remedy this by advancing our particles by t+0.01 or t+0.1 even if we have an s value of 0, but we failed. This is our current state of the project and as a result we have not implemented gravity.