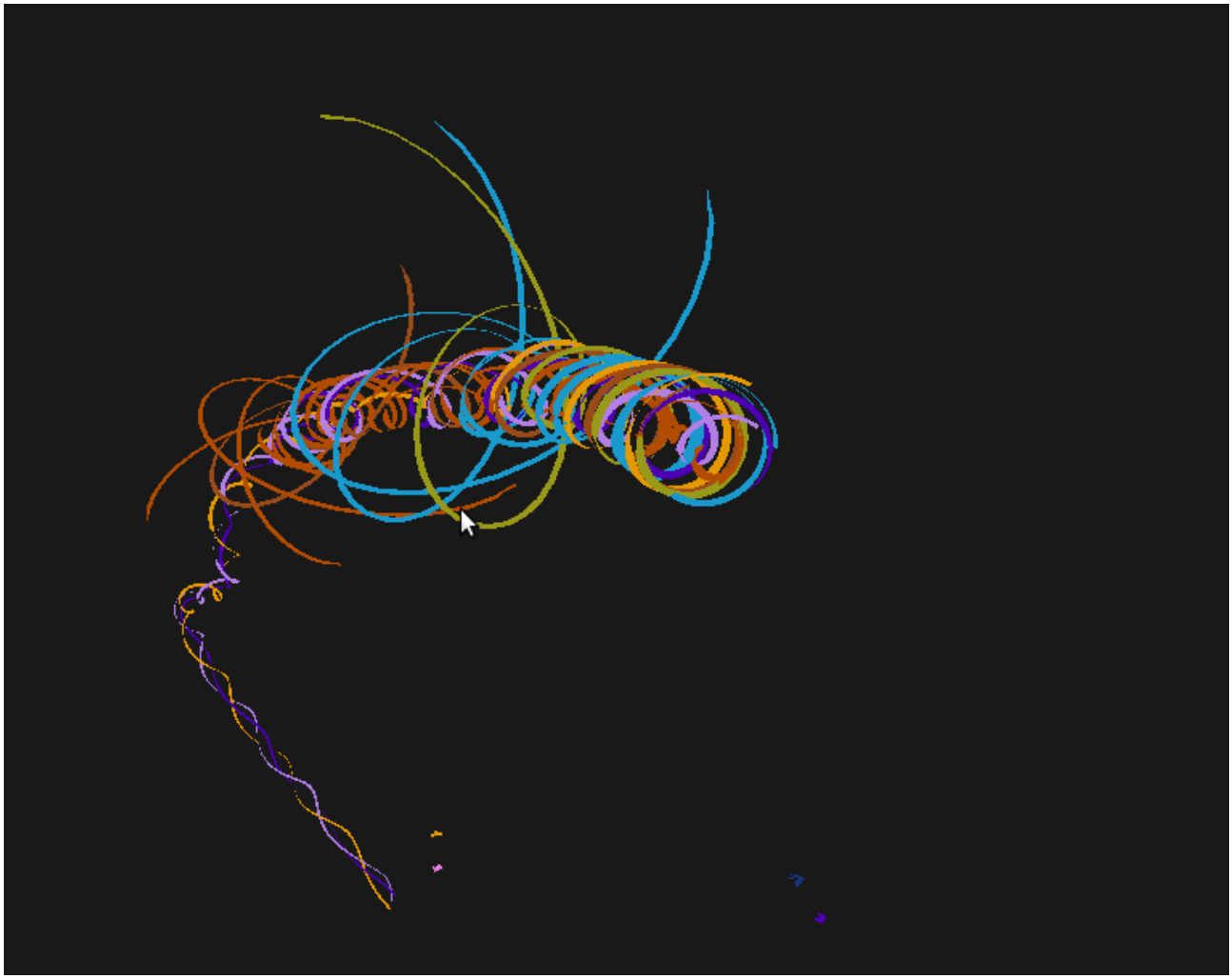
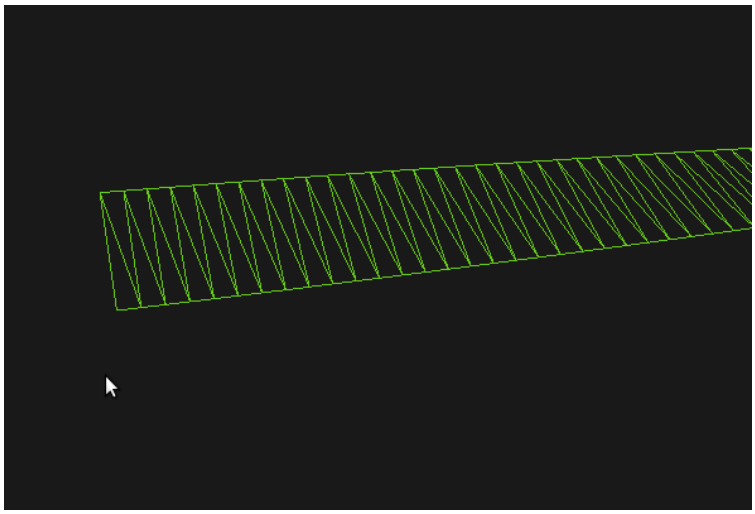
Ribbons



User Manual

The application is quite self-explanatory when you run it but here is are the main functions you can do:

- Hit Space to add ribbon to current bundle
- Use arrow keys to move the ribbons relative to the camera
- Hit Enter to split the current bundle of ribbons evenly into 2.
- Use Tab to cycle between all the available bundles
- Press 1 for Phong Shading
- Press 2 for Constant (Surface) Shading
- Press W for wireframe
- Press N to display vertex normals
- Press R to reset
- Left mouse will tumble
- Right mouse will dolly in and out
- Middle mouse will track but only if it isn't following any ribbons



Algorithms and Techniques

Ribbons: General Overview

I have created the basis for a system of ribbons which can flock together and produce interesting interactive animation. The end result can be seen in the video RibbonsDemo.ogv.

The basic ribbon is defined entirely by a position at the tip of the ribbon and a twist around it's center. From these two values I can work out the first two vertices which I refer to as v0 and v1. The trail is made purely out of the history of v0 and v1 across time.

I then have to specify the order in which OpenGL will draw the triangles and calculate the normals on a per vertex basis.

Calculating the Position

The position of the tip of the ribbon is predominantly driven by forces. These forces come from a few different areas.

First there is the force to the center of the parent Bundle. This acts like a spring (F=-kX) and has a damping to determine how much oscillation will occur.

The second is a force which repels away from all the other ribbons in the bundle. Together with the force to the center of the Bundle, this begins to produce very simple flocking behaviour. For the most part this on its own is good enough to produce a cool looking effect but there were a lot of intersections with the trails of other ribbons so I added an extra force which repels away from the trails of the other ribbons. Since this avoidance was only approximate anyway, I made it repel every 3rd position in order to save time.

When all the forces for the ribbon are calculated, it moves on to apply a rotational velocity based on the twist of the parent Bundle. In it's current state, however, my program doesn't actually calculate any vortex forces since I was running low on time. Instead each of the ribbon's positions is rotated by a quaternion uniformly which is what form the final twist.

Calculating the vertices
Ribbon::update();

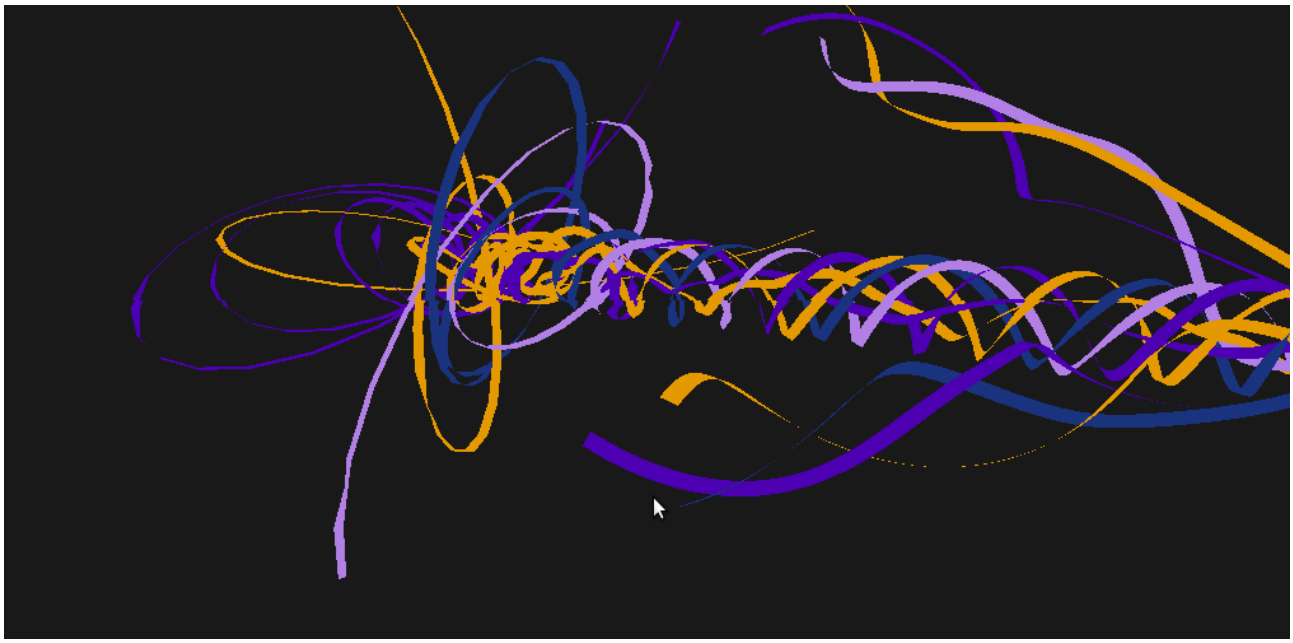
My two main vertices, v0 and v1 are calculated through the position and the twist like I mentioned earlier. To calculate the axis in which to twist the two vertices, I created orthonormal axes based on the velocity vector. This gave me the axis along which I wish to displace the vertices.

```
V = p0 - p1;
U = V.cross( ngl::Vector(0,1,0) );
ngl::Vector N = V.cross(U);
v_v0 += (N * m_width/2);
v_v1 += (-N * m_width/2);
```

Now I have this I use a quaternion to rotate the vertices by the twist value.

Building the Vertex Arrays
Ribbon::buildVAO();

To draw the entire ribbon, I needed to pass down the list of all the vertices in the vertex history array, on every frame. Since my ribbons grow from 0 up to a max size, this means I have dynamic topology and cannot simply upload all my vertices to the GPU at the beginning. After reading online, I decided that using a Vertex Buffer Object would be the best option for transferring my data. Unfortunately I was unable to make the VBO work for more than about 600 vertices which was much too limiting so ended up resorting to a slower but safer method.
My current method is to create a Vertex Array Object.
I then push back all my vertices, specifying the winding order of the vertices so that my normals all point in the same direction.
Next I calculate the normals. I am actually smoothing the normals between the edges of the triangles in order to produce a much smoother ribbon.
Lastly I upload the vertex arrays to the GPU and draw. After the draw method, I delete the VAO so that there are no memory leaks.
I realise this is quite inefficient and there is a significant performance hit but this is the only stable way I have found so far.



Branching

The branching is a cool feature I added because the new branches which are created have more user control of their flight path compared to the Solo method.
The RibbonManager::splitBundle() function has the ability to spit one bundle into an arbitrary number of output bundles although, due to time constraints I have only made a button for splitting in half.
I have also created a class called BranchBundle which is designed for a different form of branching. For example if I were to create/load up a set of l-system rules in the future I would use the BranchBundle as a simple branch segment of the tree. Again, I did not have the time to fulfil the l-system part of this code but the simple branching method also produces an equally exciting effect.

Camera Rotations

One of the main things I worked on was the camera. I extended the ngl::Camera to provide quaternion rotations for tumbling around a point, in a similar way to Maya or Houdini. The algorithm for this goes as follow:

- Get the screen x and y coords from the previous frame
- Remap x and y to a range of -1 to 1 (origin at the center of the screen)
- Map the x and y to a sphere in camera space in order to get a z value
- Repeat step 1 to 3 with coords from the current frame
  - This produces 2 vectors pointing towards the center of the camera sphere.
- I then multiply these two vectors by the ViewMatrix which gives me both vectors in world space
- The cross product then gives me an axis to rotate around.
- The dot product can be rearranged to get the angle between the two vectors.
- Finally I can transform my camera eye by the rotation quaternion given by the axis-angle pair.

This works fantastically well and I have even had requests from fellow students to use the code in their programs.
The Camera also has a simple Camera::follow(Bundle *) method which was very useful because it allowed me to re-target certain bundles with the camera and to follow them wherever they moved.