Programming Assignment #2
Rollercoaster

CSE 458 Computer Graphics, Fall 2010

1 Introduction

Your second assignment will require you to make a controlled animation and to learn some more advanced OpenGL programming. You will create a rollercoaster track along which the camera will move. You will include an environment for your rollercoaster; this may be a simple skybox, or may also include a terrain (using your heightmap code?) or other scenery. You may use the GLUT toolkit for interaction with the mouse and keyboard to give the user interactive control over rotation of the camera or other control over the motion / view.

You must submit working code, example output (including a video), and a readme file explaining whether or not you met each requirement and detailing any extra credit you added. The readme file is important! Your grade may suffer if you do not include one!

2 Environment

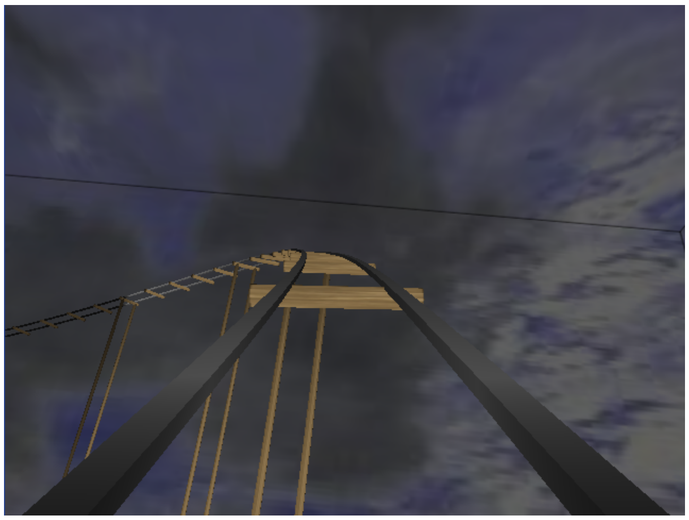
It is required to have a skybox; it is fine to reuse the skybox from your heightmap project. You may get new textures to put on the faces. You may also include a terrain using the code from your heightmap project. Feel free to add anything else you want to for scenery; be creative, it may earn you some extra credit!

3 Rollercoaster Track

3.1 Track Appearance

The track for your roller coaster must be detailed. It must have 3D structure, not be made merely of lines. It must use OpenGL material properties (glMaterialfv). The light or lights in your scene must also have diffuse, ambient, and specular properties set (glLightfv). Also, as your track gets large it will become too much to recreate every time the display function is called. This will result in frame rate problems. To avoid this, you must use a an OpenGL display list for your track. This utilizes your graphics hardware to save the important information about your track the first time it is drawn, then render it very quickly by calling the display list in the display function.

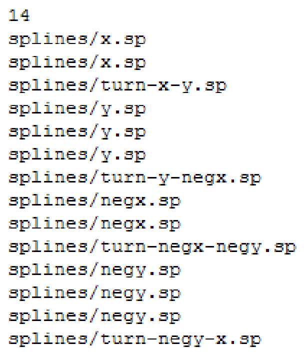
Figure 1: A Screenshot of a Completed Rollercoaster



3.2 File Types

You will not be writing one specific rollercoaster track into your code. Instead, you will be reading a track file which dictates control points, a set of locations that your track must pass through in the order they are read. The track file will instruct your program to read one or more spline files (extension .sp) which contain displacements from the last control point to the next one. This allows you to reuse common pieces (a turn, a hill, a loop...) by including a spline file multiple times in your track file. An example spline file is shown in Fig. 3.

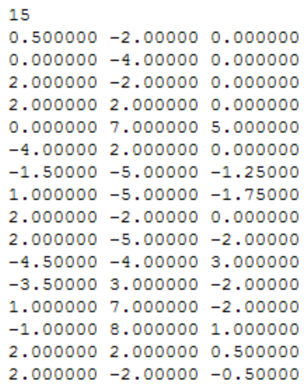
Figure 2: A Sample Track File



```
14
splines/x.sp
splines/x.sp
splines/turn-x-y.sp
splines/y.sp
splines/y.sp
splines/y.sp
splines/turn-y-negx.sp
splines/negx.sp
splines/negx.sp
splines/turn-negx-negy.sp
splines/negy.sp
splines/negy.sp
splines/negy.sp
splines/turn-negy-x.sp
```

The first number indicates the length of the file (how many entries there will be). Then each line is a spline file to be read. A track file may include many spline files or just one. It may also reuse pieces, as seen here.

Figure 3: A Sample Spline File



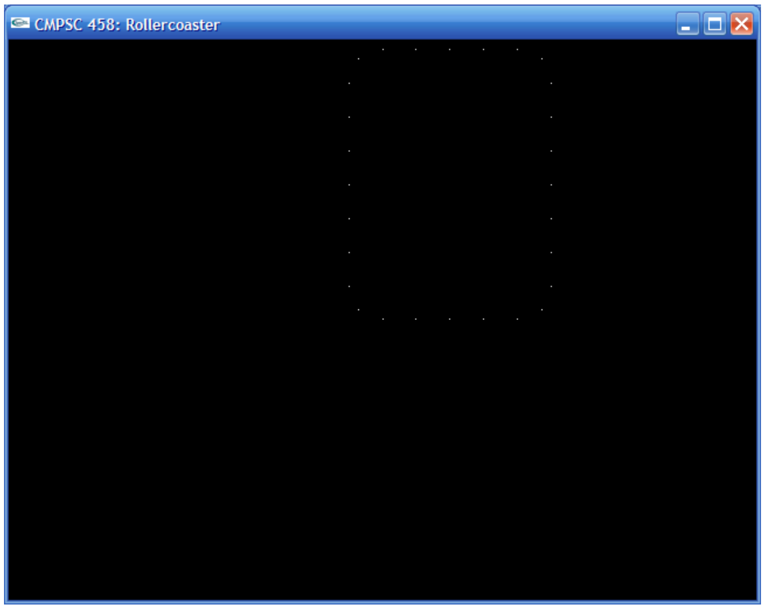
```
15
0.500000 -2.00000 0.000000
0.000000 -4.00000 0.000000
2.000000 -2.00000 0.000000
2.000000 2.000000 0.000000
0.000000 7.000000 5.000000
-4.00000 2.000000 0.000000
-1.50000 -5.00000 -1.25000
1.000000 -5.00000 -1.75000
2.000000 -2.00000 0.000000
2.000000 -5.00000 -2.00000
-4.50000 -4.00000 3.000000
-3.50000 3.000000 -2.00000
1.000000 7.000000 -2.00000
-1.00000 8.000000 1.000000
2.000000 2.000000 0.500000
2.000000 -2.00000 -0.50000
```

The first number indicates the length of the file (how many entries there will be). Then each line is a 3 dimensional offset to the next control point. So (1, 0, 0) would indicate that the next control point should be one unit in the X direction from the last control point, but not that it should actually be at the point (1, 0, 0) in the global coordinate system.

3.3 Interpolation

The output of your starter code, seen in Fig. 4, only places a dot at each control point. In order to draw a detailed track, you must have a way of figuring out intermediate points between the control points. This is called interpolation.

Figure 4: The Output of Your Starter Code

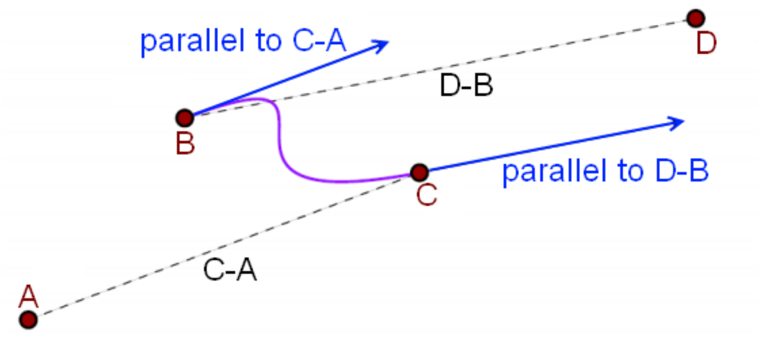


To interpolate, you will use the Catmull-Rom splines that you learned about in class. Let's review: To interpolate between two control points, we need to know their locations as well as the locations of the control points immediately before and after the points we wish to interpolate.

In Fig. 5, if we wish to interpolate between B and C, we need to know A, B, C and D. We will perform the interpolation by defining a parametric function $f(u)$. As u goes from 0 to 1, $f(u)$ will travel along the magenta line from B to C.

$$f(u) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{bmatrix} \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \\ D_x & D_y & D_z \end{bmatrix}$$

Figure 5: Interpolating spline from point B to point C



In your code, you will implement this matrix multiplication (you can work it out on paper to get a long function in terms of A, B, C, D, τ, and u). You probably want to define a function that takes four points and a u value and returns the 3D location at that u value interpolating between the 2nd and 3rd point. It is okay for τ to always equal 0.5.

When you draw your track, to get smoothly interpolated locations for vertices, you can keep a queue of 4 control points and iterate u by a small increment from 0 to 1. Each time u reaches 1, pop the oldest control point from your queue and push the next one to the end, then set u back to 0 and iterate again.

4 Camera Motion

4.1 Physics

Your camera must move along the track with somewhat realistic physics. This means that it must follow gravitational acceleration. You need to either start your rollercoaster on a hill or give it some initial velocity so that it can get over hills. The basic physics equations you need to know are that the sum of $KineticEnergy$ and $PotentialEnergy$ must never change. $KineticEnergy = 0.5 * mass * velocity^2$. $PotentialEnergy = mass * 9.8 * height$. Give your coaster some non-negative mass, it does not matter what. At the beginning of your program, calculate how much energy the coaster has from its starting height and initial velocity. Every time you need to move the coaster, you can calculate its current velocity by examining its current height and making sure that the sum of $KineticEnergy$ and $PotentialEnergy$ is the same as at the start.

4.2 Moving the Camera

You can move the coaster in either the doIdle function or create a glut Timer function. A timer is better because you can ensure the same running speed on any system. If you use the doIdle function, the running speed is dependent on processor speed. The camera position is determined essentially the same way that you drew the rollercoaster; iterate through the control points keeping a queue of 4 points and, within that loop, iterate u from 0 to 1 by some small increment to get smooth interpolation.

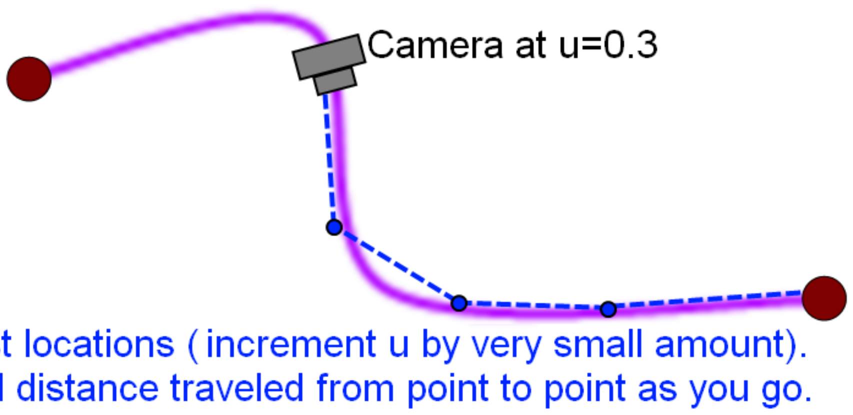
Each time you move the camera, you know its current position (both in global coordinates and as a u parameter value along a segment of your track) and its velocity (dependent on initial energy and current height). If you use a timer function, you set the refresh rate, so you know how much time will elapse before the next frame is rendered. The distance you want to travel is equal to your current velocity times the time to the next rendering.

When you know the distance you want to travel, there is still a problem; you need to know what value of parameter u is that distance away from your current position. In order to find this, we need to examine the distance along the curve. Distance between points (a, b, c) and (x, y, z) is equal to $\sqrt{(a-x)^2 + (b-y)^2 + (c-z)^2}$. You can use this formula for distance to determine the correct u value: beginning at your current u position, increment u by a very small amount and test the distance (in global coordinates) between your current location and the one at the new u value. If the distance is less than your desired travel distance, move the camera to the location at the new u value, decrease your desired distance to travel by the distance you just moved the camera, then repeat the process (Fig. 6). Repeat this until your very small increase in u would result in greater distance travel than you desire, at this point you have moved enough and will render the new image.

5 Extra Credit

Possible ideas are extra track features (like support columns), secondary animations (other moving things in the environment), banking turns, interactive

Figure 6: Moving Your Camera


Camera at u=0.3
Test locations (increment u by very small amount).
Add distance traveled from point to point as you go.

features, or anything else you can think of. As usual, be creative, describe everything you do in the readme file, and make things look nice to get more points.