Theme Park Documentation

Ferris wheel: By holding down the “R” key or clicking on the ride, the wheels, support beams and the carriages rotate around the centre of the wheels and the carriages also rotate around their own individual axis. As the wheels rotate they increment a rotation variable so they spin anti-clockwise. Whereas the carriages decrement a rotation variable so they spin clockwise and by positioning each carriage hanging from their support beams gives the illusion that the carriages are affected by gravity as the wheel spins.

Merry-Go-Round: Apart from the horses, the merry go round was made using “GL_TRIANGLE_FAN” and “GL_QUAD_STRIP” within for loops that go from 0 to 360 and increment by 18. This along with the line “glVertex3f(Cos(j),+1,Sin(j));” draws each quad strip or triangle fan in a slightly different position each time to create a circle, a cone or a cylinder, with j being the variable that is incremented. As the quad strips and triangle fans are defined the normals are also set by the line “glNormal3f(Cos(j),0.5,Sin(j));”. This gives the cylinders and cones more dynamic lighting because light affects each quad or triangle individually, instead of just the one finished shape. Holding down the “R” key or clicking on the ride, rotates the Merry-Go-Round’s middle section, as well as raises the “horses” up and lowers them down continuously as it spins, just as it does with the real thing.

Teacups: The teacups were created using scaled cylinders along with a glutTeapot() in the middle. Holding down the “R” key or clicking on the ride, will rotate each teacup and the teapot around their individual axis, with the teapot rotating in the opposite direction as the tea cups and the teacups rotating around the teapot.

Cube and Prism: To create the cube and the prism that I have used for many of the objects in my scene, I used vertex arrays to hold the 8 or 6 vertices, along with the colours and setting the normals for each face.

Mountains and rocks: To create the mountains and the rocks I used subdivision and recursion. By drawing a triangle base and calculating the midpoints of the three sides along with the centre points by getting the mean x, y and z values of each set of three points, I can use the values to draw triangles in the correct positions to draw multiple tetrahedrons. Adding a random value to the y variable for each “centre” point, gives each tetrahedron a different shape and size from the other. The original subdivision object has different heights for each tetrahedron but the copies are the same as the original.

Swimming pool: For the water I used subdivision to create a random (wave) effect, but instead of starting with a triangle I used a quad. Using quads made sense because most pools are square. For the steps leading up to the pool I scaled the Cube() object to the shape of a step, I then used a for loop to draw 4 steps in the correct positions, by translating each one to the correct position before it is displayed.

Benches: For the tables and benches I used the “Cube()” and transformed them appropriately to create each part of the objects.

Fencing: For each fence post I scaled the “Cube()” object to the correct shape and placed a scaled “Prism()” on top to create a fence post. Instead of translating and displaying each fence post manually, I used for loops to do it automatically, which makes it much more efficient.

Trees: When I display the trees I kept the leaves (cone) and the tree trunk (cylinder) separate from each other so I could colour one green and the other brown, creating a more convincing tree. This also means that the leaves can be coloured white when the user selects the “winter wonderland” mode. To display all of the trees I used a double for loop, one to draw one row and the other to copy and translate that row multiple times in the x direction.

Sun and Moon: By pressing the “T” key or clicking on either the sun or the moon, the variable “rotateLight” increments by 1 and the “Sun”, “Moon” and the main light (which is a directional light) translate around the scene using the line “glTranslatef(35*Cos(rotateLight), 35*Sin(rotateLight),30);” simulating the passage of the day. 35 is used for setting how wide the circle of rotation is, essentially its radius, 30 sets the sun and the moon at a 30 degree angle to the scene, to more accurately simulate the position of the real life sun and moon. When “rotateLight” is equal to 180, i.e. the day has ended, the main light “light0” is disabled and the three spot lights are enabled. Also while “rotateLight” is less than 30 degrees or more than 150, the light0 colour is set to a yellow colour to simulate sun rise and sun set. The light is set to white while “rotateLight” is between 30 and 150 degrees to simulate daytime. I also set them to emit yellow and grey using (GL_EMISSION) so they wouldn’t be affected by lighting, since they are meant to be generating the light.

Stars: I used a glutSphere() for the stars and just as I did for the trees, I used a double for loop to display them. I also set each star to emit a white colour through “GL_EMISSION”, which meant they weren’t affected by the lights, but stayed white, making them look more like stars. The stars will only be displayed when “rotateLight” is more than 180, since that is when the “night” begins and once “rotateLight” is equal to 360 it will reset back to 0 and the stars won’t be displayed.

Colours: Instead of specifying which elements of each material properties array that I wanted to change and to what value every time I displayed an object, I created functions for each colour to handle this. This means that I only have to call the correct function for the colour that I want the next object to be displayed as e.g. “colourWhite(mat_diffuse, mat_specular, mat_ambient);”.

Changing colours: The user can change the colours of the three rides (the Ferris wheel, merry-go-round and teacups) by selecting the appropriate “view” option in the menu and then use the 1, 2 and 3 keys to change the colour of the object. For example select the “Ferris Wheel View” option in the menu and then press the 2 key to change the colour of it to green. The user can also select the ride by clicking on each ride and using the 1, 2 and 3 keys to change its colour.

Cross product and normals: I created a function called “CalculateNormals” that inputs the x, y and z positions of two vertices and calculates the normal. Unfortunately it doesn’t produce the desired results so I resorted to defining the normals manually, which wasn’t any trouble since the Cube() and Cylinder() objects

were re-used many times in the scene. The reason that the cross product calculations don't work is because of how the objects were defined. By using the same value each time, they cancel each other out, leaving a normal of (0,0,0). It was easy to define the normals for the cylinder and cone, because I integrated it into the for loops with the line "glNormal3f(Cos(i),0.5,Sin(i));", so as the quads and triangles are being defined, the normals are as well.

Textures
For textures I used the image loader created by Jacobs. B (2007-2013), both the imageloader.cpp and the imageloader.h, along with the code from the texture-sample main.cpp.

Grass/snow: For the ground I used a grass texture created by Rodrigosetti (2009) which makes the ground look much more realistic than using a solid green colour. The user can also swap the grass for a snow texture, which was created by Sicuatrirrotor, by selecting the "winter wonderland" option in the menu.

Winter Wonderland: By selecting the "Winter Wonderland" option in the menu the ground texture is changed from grass to the snow texture created by Sicuatrirrotor and some of the objects are coloured white to simulate snow fall. The user can choose to set everything back to how it originally was by selecting the "Summer" option in the menu.

Fog and mist: By selecting the "Foggy Day" or "Misty Day" options in the menu, the user can choose to display either a fog or a mist over the scene, they can also switch off the fog or mist by selecting the "Clear Day" option. The fog and the mist are basically the same except for the density being 0.1 for mist and 0.3 for fog. I used the code for the fog that C. Swiftless Tutorials (2010) created to display the fog and mist.

Camera: The camera can either be controlled fully by using the keyboard or by moving with the W, A, S, D keys and looking by using the mouse. The camera is defaulted to the keyboard, but the user can swap between the keyboard and the mouse by using the two options in the menu "Mouse Camera" and "Keyboard Camera". I used the camera class that was created by A. Swiftless Tutorials (2010) which uses trigonometry and circle theorem to calculate the relative position of the "camera" and translate the scene accordingly. To enable the use of the mouse to control the camera I used the mouse movement function created by B. Swiftless Tutorials (2010).

HUD: By drawing text onto the viewport you can create a HUD, where you can display a variety of information. The user can also hide the HUD by selecting the "Hide HUD" option in the menu. I used the HUD code that was provided to us and added to the code what I wanted to display.

Picking: The user can use the mouse to click on the three rides and either the sun or the moon, to allow them to rotate on their own The user can also change the colour of the last selected ride by pressing the 1, 2 and 3 keys; 1 = red, 2 = green and 3 = blue. By selecting the "stop animation" option in the menu, the rides, the sun and the moon stop rotating.

Jacobs. B. (2007-2013) OpenGL Tutorial [online]. Video Tutorials rock Available at: <http://www.videotutorialsrock.com/opengl_tutorial/textures/text.php> [Accessed on 5th December 2012].

Rodrigosetti. 2009 *project-ia725* [online]. code.google Available at: <http://code.google.com/p/projeto-ia725/source/browse/trunk/data/grass.bmp?r=5> [Accessed on 17th November 2012].

Sicuatrirrotor. *Sicuatrirrotor - Revision 558* [online]. Sicuatrirrotor.googlecode Available at: <http://sicuatrirrotor.googlecode.com/svn/trunk/ProyectoDescargable/Proyecto/Data/Texturas/> [Accessed on 5th December 2012].

A. Swiftless Tutorials. 2010 *22. OpenGL Camera* [online]. Swiftless Available at: <http://www.swiftless.com/tutorials/opengl/camera.html> [Accessed on 22nd November 2012].

B. Swiftless Tutorials. 2010 *23. OpenGL Camera Part 2* [online]. Swiftless Available at: <http://www.swiftless.com/tutorials/opengl/camera2.html> [Accessed on 29th December 2012].

C. Swiftless Tutorials. 2010 *14. OpenGL Fog* [online]. Swiftless Available at: <http://www.swiftless.com/tutorials/opengl/fog.html > [Accessed on 10th January 2013].