3D Modelling and Animation of a Rollercoaster using OpenGL and GLSL

# Introduction

In this assignment I will be using C++ with the OpenGL and GLSL libraries, to develop a 3D simulation of a rollercoaster. It will involve using OpenGL and GLSL to create models, animations, special effects and simulating them using Ordinary Differential Equations (ODE).

The simulation will consist of 3 trains/cars, each of which will have four fully animated wheels. The motion of the cars round the track will be governed by an ODE, they will go round the track in a clock wise manner (this is may change depending on the orientation and view of the camera). Along with the track and cars the simulation will also display a wider 3D environment which will include but not be limited to the following; a floor, a skybox, lighting, shaders and other effects in order to make it look as realistic as possible.

# 3D Scene Modelling

In order to demonstrate the simulation in the manner required, the scene will be 3 dimensional. In order to produce the desired effect, a 3D projection matrix will have to be used in order to make the scene 3-dimensional. A 3D projection matrix is a method that allows us to map 3-dimensional points onto a 2-dimensional plane. A computer screen is a type of 2D plane, so we need to use 3-dimensional projection in order to display the objects of the rollercoaster in the simulation. The simulation will be viewed at through a camera, which the user will be able to move. The space the camera is able to view at any given moment is known as its View Frustum this is created by 2 planes; the Front and the Back clipping planes. Objects within the 2 planes (View Frustum) are drawn onto the scene, whilst those that lie outside the cameras View Frustum are either clipped (cut off) or discarded all together. Objects too close the camera or those beyond the 2nd plane are discarded because if they weren’t then they’d appear too blurry.

The following code is used to create the cameras’ View Frustum.

Glfloat aspect = (GLfloat) width / (GLfloat) height; //calculate the aspect ratio of the window

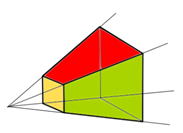
by dividing the width by the height.

glMatrixMode(GL\_PROJECTION); // Allows the Projection matrix to be used.

glLoadIdentity(); // Resets it.

gluPerspective(10.0f, aspect, 1.0f, 100.0f)// The perspective projection (field of view y in

degrees, aspect ratio, near view, far view).



(Picture from http://wildlifeandrew.blogspot.co.uk/2012/09/modified-frustum-culling-for-ray-tracing.html)

The coloured frustum in the picture above, shows the view frustum of a camera with both the Front and the Back planes.

# 3D rollercoaster trains and track modelling

The primary goal of this project is to simulate the physics and the dynamics of a rollercoaster, to do this there will need to be a track and a number of cars that travel round the track. The outline of the track will cover a circle within the X and Y Plane, with its height mapped using a sinusoidal function (a smooth mathematical curve) of the angle of revolution around the track itself.

The scene will also contain 3 fully animated rollercoaster trains (or cars), each car will have a main body with four wheels attached to it. The cars themselves will be cuboid in shape, whilst the wheels will be a torus shape (ring shaped). The track itself will consist of an inner and outer boundary, these will determine the width of the track. Then the track itself which will be a quad shall be placed in between these boundaries.

GLfloat track\_innerradius=5.0f;// Inner radius of the track

GLfloat track\_outerradius=8.0f;// Outer radius of the track

// Build inner track

glPushMatrix();//Translate to the next position in the track glTranslatef(track\_innerradius\*cos(x),track\_height\*sin(nh\*x),track\_innerradius\*sin(x));

glCallList(1);

glPopMatrix();

// Buid outer track

glPushMatrix();

//Translate to the next position in the track

glTranslatef(track\_outerradius\*cos(x),track\_height\*sin(nh\*x), track\_outerradius\*sin(x));

glCallList(1);

glPopMatrix();

Above is the code within the simulation, which will determine the inner and outer parts of the track. The actual track will be a quad that fits between these two. It calculates the equations using the variable track\_innerradius which is 5 for the inner track and the variable track\_outerradius which is 8 for the outer track.

# Rollercoaster simulation and animation

Behind the models of the rollercoaster certain laws of physics will have to be implemented, in order for the simulation to work in the manner required. Most notably is using the laws of classical dynamics to simulate the motion of the rollercoaster’s’ cars, this will be done by the calculation of mathematical equation also known as an Ordinary Differential Equation. There will be no effects of friction in the simulation, the cars will not appear to lose any energy over time (no friction between cars and track) or when two cars collide with each other. The cars will also be under the effects of gravity, they will stick to the track and not float away.

The following notation will be used to simulate the motion of the cars.

R is the radius of the rollercoaster track (in metres).

H is the height of the rollercoaster track (in metres).

n is the complete sinusoidal waves the track has (the number of hills).

θ(t) is the angle of revolution of the car going round the track at time t (in radians).

Is the angular velocity of revolution of the car at time t (in radians per second).

Is the angular acceleration of revolution of the car at time t (in radians per).

Is the x-coordinate of the car at time t (in metres).

Is the y-coordinate of the car at time t (in metres).

Is the height of the car at time t (in metres).

The following shows the values that will go into the above equation.

R = 100m

H = 20m

n = 4

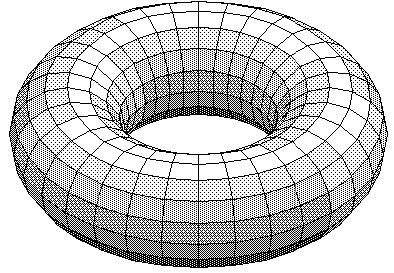
g = 9.807 ms

The simulation will also contain more than just a track and cars; it will also have a sky box (this will be a cube that the scene sits inside) and a floor (a flat plane that is separate to the skybox) these will show the boundaries of the viewable parts of the simulation, lighting which will be used to illuminate the scene, shaders which will be used to add special effects to the scene and some particle effects again to add extra effects into the scene.

Different particle systems can be implemented in a variety of ways; from fire, smoke and water etc. Therefore certain physics will also play an important role when it comes to a particle system, depending on the desired effect. Smoke is able to flow freely flowing the push of the wind, where as fire generally travels in an upward motion.

# Wheels modelling and animation

To model the wheels of the cars in the simulation, I will be using tori (singular torus) which are a kind of ring or donut shaped object. The wheels will be animated so that they turn round, in the direction that the car is travelling. Each of the cars will have four wheels attached to it, therefore there shall be a total of 12 wheels in the simulation.



(Picture from <http://www.math.cornell.edu/~mec/2008-2009/HoHonLeung/page6_knots.htm>)

The picture above show the torus shape, this will be used to represent the car wheels in the simulation.

# Special effects using shaders

A shader is a type of computer program, which is executed on a devices graphics processing unit (GPU). It is used to produce certain levels of both light and darkness of images whilst also being used to produce special effects and do post processing. In OpenGL a shader is programmed using the OpenGL Shader Language (GLSL), multiple shaders can be programmed and implemented in parallel of each other in an OpenGL application. Due to the fact that the GLSL shader language is different to the OpenGL language, the shader is stored in a separate file to included and initialised in the OpenGL application. File extensions of .vert and .frag are used to denote a GLSL shader, .vert is a vertex shader and .frag is a fragment (pixel) shader.

Light is something that can be programmed into shader, without light we wouldn’t be able to perceive an object as being 3-dimentional. Light can be implemented into the simulation in many ways, as there are different types of light and light sources; Ambient, diffuse, specular and shininess lighting. Ambient light is a general brightness level of light that is incorporated into the scene, a good example of Ambient light would be that, which comes from the Sun and fills a room even thought the Sun isn’t itself present in the room.

# Camera definition

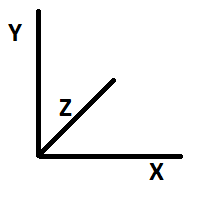
The camera is a very important part of the simulation, it will allow the user to traverse through the scene. Without one you would only be able to view the simulation from a predefined position and angle.

The main features of a camera are as follows;

* The position of the camera.
* The camera’s view direction.
* The camera’s orientation
* The camera’s field of view e.g. wide angle
* The depth of the camera’s field of view taking into account both the near and far distances.

The user will be able to move and manipulate the camera within the simulation, allowing them to observe the different parts of the simulation. They will be able to do this by pressing certain keys on the keyboard.

* ‘a’ and ‘A’ will allow the camera to move forwards.
* ‘z’ and ‘Z’ will allow the camera to move backwards.
* ‘.’ (Full stop) will rotate the scene clockwise around the origin.
* ‘,’ (Comma) will rotate the scene anti-clockwise around the origin.
* ‘r’ and ‘R’ will reset the camera to its original parameters.
* ‘x’ and ‘X’ will rotate the scene clockwise around the x axis .
* ‘s’ and ‘S’ will rotate the scene anti-clockwise around the x axis .
* ‘y’ and ‘Y’ will rotate the scene clockwise around the z axis.
* ‘h’ and ‘H’ will rotate the scene anti-clockwise around the z axis.
* The Escape key ,’q’ and ‘Q’ will exit the application.



# Collision detection and response

During the simulation if two of the cars collide with each other, then they should respond accordingly. In order to do this effectively we need the simulation to bound the cars up within a box, then to keep checking if these bounding boxes intersect with one another. If and when the cars do collide with each other, the simulation needs to display this to the scene. Seen as there is to be no friction between each of the cars, neither of the cars will lose any energy when they do collide with each other. This will be done by getting the cars that have collided to swap velocity with each other.

The following code is a function that handles the collision between two cars, whihcswaps their velocities round.

void handle\_collision(Car\_Physics &car1, Car\_Physics &car2)

{

if (!check\_collision(car1, car2)) return;

//Swapping Velocity of car1 and car2

float av = car1.angular\_velocity;

car1.angular\_velocity = car2.angular\_velocity;

car2.angular\_velocity = av;

//Adjusting angle of car1 and car

if (car2.angular\_velocity > car1.angular\_velocity)

{

car2.angle += car1.angular\_velocity \* timeStep;

car1.angle -= car2.angular\_velocity \* timeStep;

}

Else

{

car1.angle += car2.angular\_velocity \* timeStep;

car2.angle -= car1.angular\_velocity \* timeStep;

}

car1.SyncPos(); //calculate new position with angle

car2.SyncPos(); //calculate new position with angle

}

# Conclusion

In order to portray a rollercoaster simulation in C++ with OpenGL, it needs to have the track and some cars, but it must also be underpinned with certain laws of physics derived from the calculation of mathematical equations such as Ordinary Differential Equations (ODE). Without either of them I wouldn’t be able to properly simulate a rollercoaster. The cars and the track must have a relationship between each other, which allows the cars to follow the same curve that the track does whilst also detecting any collisions that may occur . To demonstrate a further understanding of the OpenGL and GLSL languages I must also demonstrate other techniques such as shaders, lighting and particle systems, so as to convey what I have learned in this module.