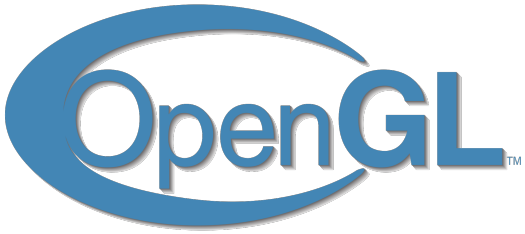

Universite Libre de Bruxelles

INFO-H-502

Virtual Reality

Report: Rollercoaster Animation in modern OpenGL

Elisabeth Gruwé
Oskar Radermecker

Professor : Gauthier Lafruit

2017-2018

Contents

1. Description of the project

The aim of the project was to develop a roller-coaster in modern OpenGL. It was decided to build the whole environment in a "creepy", Halloween-like atmosphere. A number of objects and features were implemented to achieve this goal and are enumerated hereunder. The most important ones will be briefly described in this report.

1.1 Objects

1. A Cubemap with a night sky and a moon. The light is placed so that it looks like it is emitted from the moon. The amount of light is also kept to a minimum to give a creepy atmosphere.

2. A roller-coaster on which a cart is moving. The roller coaster itself has a metallic, light grey texture while the cart is painted in a shiny red colour reflecting its surroundings. This colour is the only bright colour in the scene to give a creepy impression. The cart also has a little camera attached to its bonnet whose view can be visualised anytime by pressing "C". More details can be found in chapter 2.

3. Two torches that announce the start of the roller-coaster. The flames are formed by particles in movement. More details can found in section 3.1.

4. A torn flag welcoming new visitors can be seen at the top of the roller-coaster. On it is written "Welcome to the ghost roller coaster". Brave is the one who dares sit in that cart. More details can found in section 3.3.

5. A dead tree covered with snow, placed in one corner of the support surface. It symbolises life and death and nourishes itself from the souls of the people having travelled in the ghost cart.

6. A camera placed after the first loop takes pictures of the horrified look of the cart's passengers every time it passes in front it.

7. Two floating ghosts guarding the coaster and rotating around the platform.

8. A soundtrack with a creepy fun fair music is run in the background to completely immerse the viewer in the atmosphere.

9. And last but not least, an astonishing loading image to help the viewer wait through the loading time.

1.2 Light and shadows

Without light, a scene cannot look realistic! Several light components have therefore been implemented. They are briefly described hereunder.

- **Ambient**, **diffuse** and **specular light** are applied on all objects (in the fragment shader), except on the flag and fire (code based on [1]).

- **Normal mapping** is implemented in the *DefaultShader*, that is used by the board and the lamps. There, normals are defined by a additional texture (code based on [2]).
- For the **shadows**, the first step is to generate a depthmap that stores the depth of all the objects in the scene in the light's perspective. To achieve this, a framebuffer rendering depth values to a 2D texture is used. This texture is then processed in the *DefautShader* in order to draw shadow on the board (code based on [3]).
- The flash light that is triggered periodically when the cart approaches the camera is simply a light source whose power is turned on for a tiny amount of time.

1.3 Some pictures

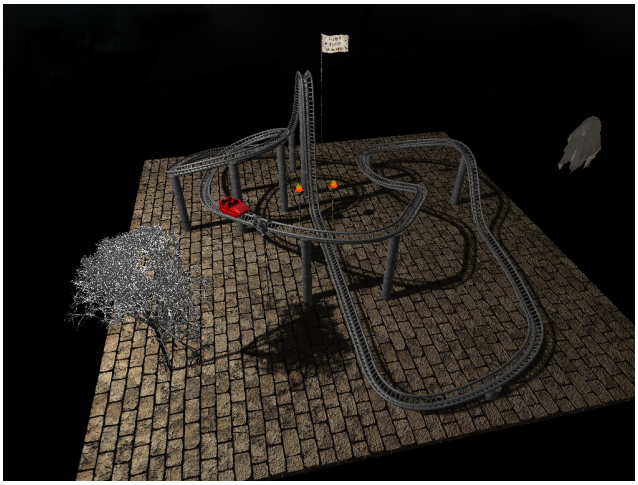

Figure 1.1: View from above of the roller-coaster and its environment. Screenshot was taken from Camera 1.

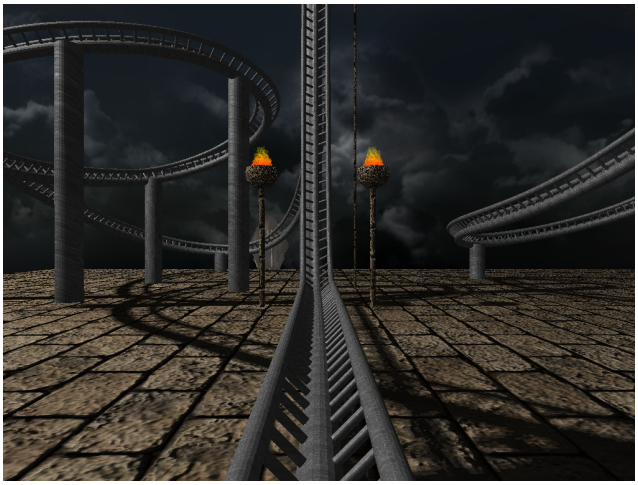

Figure 1.2: View from inside of the cart, at the start of the roller-coaster. Screenshot was taken from Camera 2.

2. Roller-coaster

2.1 Track

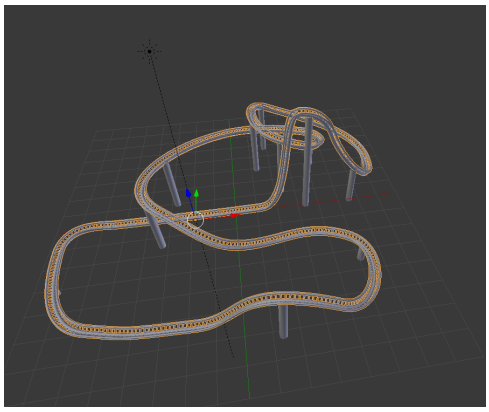

Figure 2.1: Track Creation in Blender

Once the environment is set up, the very first step in order to implement a roller-coaster in OpenGL is to create the track. Two methods were investigated for this purpose but only the one implemented in the final animation is described hereunder. This method is not entirely based in OpenGL. The Bezier curve was first created in Blender using a dedicated tool, and was then transformed to a track by using an array modifier.

This resulted in a nice and easily customisable track as can be seen on Figure 2.1. The track is then exported as an .obj file, while the curve's control points have to be extracted with a python script. It is then possible to import the control points in OpenGL, reinterpolate the curve and calculate the Fresnet vectors. This interpolated curve will then perfectly correspond to the track's object file. This method has the advantage of easily producing good looking tracks and over which more control is possible. One drawback however, is that it requires the importation of a large .obj file which can decrease the frame rate.
All the functions related to the track are located in `track.cpp`. The track object is loaded into the program as a *Model* while the curve's control points are loaded with the *loadCurve* function.

2.1.1 Bezier Curves

Although both Bezier Curves and Catmull-Rom splines were implemented, only the former was kept in the final animation as it is the only one available in Blender. Some of the functions used for this purpose were inspired by [4].

Cubic Bezier curves, the ones used in this project, are Bezier curves defined by four control points. For consistency purposes, these will be called P_0, P_1, P_2 and P_3 in this report. Figure 2.2 shows a Cubic Bezier curve interpolated from four control points. It can be seen that the curve always goes through and from P_0 towards P_1 and through P_3 arriving from P_2. This yields to four mathematical conditions on the positions and tangents of points P_0 and P_3. The explicit form of the curve is given by the following expression:

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 \quad \text{where } 0 \leq t \leq 1$$

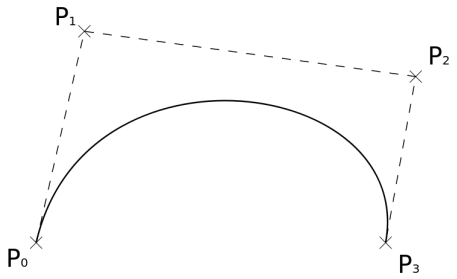

Figure 2.2: Interpolation between four points to obtain the Bezier Curve

This is directly implemented in the code (function *CalculatePosition*) to calculate the position of the interpolated point: t is increased from 0 to 1 in an iterative process. The increment is set to 0.001. When t reaches 1, it is set back to 0 and the control points are moved to the next points to interpolate the next part of the curve (function *CreatePointList*).

2.1.2 Fresnet Vectors calculation in OpenGL

The following operations are done in *StoreCurrentPoint*.

Tangents. Calculating the tangents was pretty straightforward: it is the derivative of the Bezier Curve's position. As such, the tangent at each point was calculated (and therefore for a given value of t) with the following equation:

$$B'(t) = 3(P_2 - P_1)(1-t)^2 + 6(P_3 - P_2)(1-t)t + 3(P_4 - P_3)t^2 \text{ where } 0 \leq t \leq 1$$

Normals. To calculate the normal at each point, the cross product between the previously found tangent and the *UP* vector (0,1,0) was performed.

Binormals. Similarly to the normals, the cross product between the previously found normal and tangent at each point was determined in order to find the binormal.
Note: this method fails to estimate the normal (and binormal) if the tangent is purely vertical. As this is not the case in our project, the algorithm was kept. Other methods taking into account the previous point's binormal to estimate the tangent were also implemented but resulted in dissatisfying results.

2.2 Cart and Camera Movement

The movement of the cart and cameras is initiated by the incrementation of a variable: *CpLIndex*. Since the incrementation of this variable is done every time the while loop in the main function is accessed, its speed also depends on the time between two incrementations to avoid the dependence upon the computer's performances (function *Do_Movement*).
The cart follows the spline and is consecutively rotated in two planes: y and z.
Two cameras are available and the user can switch between the two of them anytime simply by pressing "C". They are completely independent from each other.

Camera 1. The first camera is a free camera. The user is free to move around by going anywhere and looking in any direction. This camera is useful to get a more global view of the scene.

Camera 2. The second camera is restricted to the cart. It is supposed to represent to view from the little camera placed in the front of the cart. As such, it follows the spline at the same speed as the cart, but it is impossible for the user to change its view, nor its position.

3. Additional features

3.1 Fire as particle system

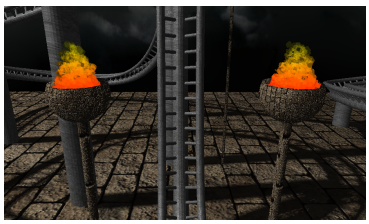

Figure 3.1: Fire

Two sources of fire can be seen on both sides of the start of the roller-coaster (figure 3.1). They were made with 6000 particles, that are small cubic-like objects. Each of these particles starts with a semi-random initial position inside a circle, and has semi-random scale and rotation (*Fire.cpp:CreateParticleModels*). They then have different initial model matrices, that are saved in buffers and sent to the *ParticleShader* via an array of generic vertex attribute data (*main.cpp:CreateParticleBuffers*). Because 12.000 particles must be drawn at the same time, it is not possible to draw them as usual and they have to be treated as *instances*. The movement of the particles is also semi-random and defined in the vertex shader. Colour and transparency change as a function of the particle's height, which is defined in the fragment shader. Note that these torches are not light sources. The code is based on ideas found in [5].

3.2 Tree with fractals

A winter tree is also implemented (figure 3.2). In an iterative way, 6 new branches are added at the end of each branch. The orientation of the branches is generated semi-randomly, in 3 dimensions, while the length progressively decreases with a factor of 0.6 until a lower bound is reached (*tree.cpp:generateFractalTree* - Code based on ideas found in Figure 3.2). The position, length and orientation of each of the branches are used to generate model matrices and draw the branches with cylinders (*tree.cpp::CreateTreePartsModels*). Since the tree is made with about 10.000 branches, they are handled as *instances*. Colour changes as a function of height, defined in the *TreeShader*. Since the tree is drawn with instances, an additional vertex shader (*ShadowParticlesShader*) is used to handle the shadow of the tree.

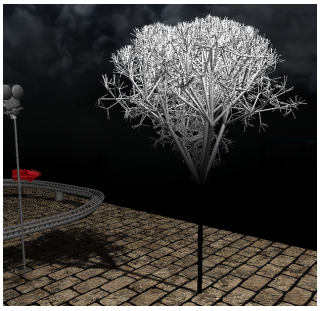

Figure 3.2: Tree

3.3 Tissue Implementation

Figure 3.3: Flag

A flag can also be seen at the top of the roller-coaster. Its 40 $\times$40 vertices, UV coordinates, VAO and VBO are created with functions in *Flag.cpp*. The vertices move respectively to each other: it is *not* a rigid body. The movement of each vertex is continuously updated in 3D with a compute shader (*ComputeShader* - Code based on ideas found in [6]): a mass is given to the vertices, and springs are considered between one vertex and the 8 vertices around it (vertically, horizontally, and in diagonal). A force that depends on time is applied on the flag. The flag is then drawn as usual, with *FlagShader*.

3.4 Soundtrack

Thanks to the non-commercial sound library *irrKlang*, a soundtrack with a creepy fun fair music is played by a speaker placed in the centre of the platform. The sound intensity and its origin depend on the position of the camera with respect to the speaker.

Bibliography

[1] Learn OpenGL. *Basic lighting*. Last consultation: 18th of December 2017. URL: https://learnopengl.com/#!Lighting/Basic-Lighting.

[2] Learn OpenGL. *Normal Mapping*. Last consultation: 26th of December 2017. URL: https://learnopengl.com/#!Advanced-Lighting/Normal-Mapping.

[3] Learn OpenGL. *Shadow Mapping*. Last consultation: 23th of December 2017. URL: https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping.

[4] Jialin Yang. *RollerCoaster*. Last consultation: 23rd of December 2017. URL: https://github.com/charlieusc/RollerCoaster.

[5] Learn OpenGL. *Instancing*. Last consultation: 27th of December 2017. URL: https://learnopengl.com/#!Advanced-OpenGL/Instancing.

[6] Muhammad Mobeen Movania et al. *OpenGL – Build high performance graphics: Using the compute shader for cloth simulation*. Pakt Publishing, 2017.

[7] GitHub repository. *RandomGeneratedFractalTrees*. Last consultation: 28th of December 2017. URL: https://github.com/teaprog/RandomGeneratedFractalTrees/blob/master/main.cpp.