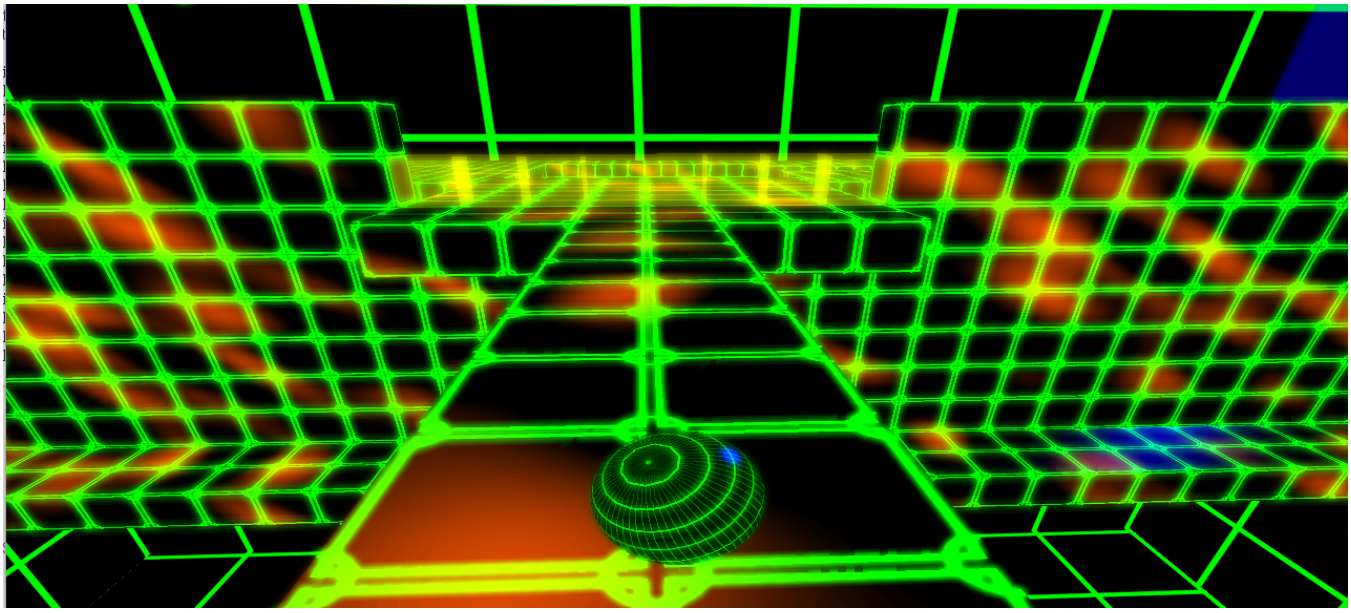
Developing a Marble Race Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Robert Lau* Benedikt Scholtes† Jan Häusler‡



Abstract

As part of the practical course 'Developing a Marble Race Game' we developed a marble race game where you have to reach the goal in a futuristic, maze-like environment - in the fastest time possible. One of our main reference points was the movie 'Tron', which is characterized by a futuristic style - involving dark environments lit up by bright lights.

1 Implementation details

In this section and its subsections the details of our implementation are presented.

1.1 General Structure

We made heavy use of classes in our code, mainly to achieve a logical separation for different parts of our code. After the creation of the window and loading of the main settings, the *Menu* class takes over and draws the menu. Starting and stopping a game is simply achieved by creating and destructing a new *Game* object. This eliminates problems with the Physicsengine, every new game is loaded from scratch, and no old physics objects are left. The *Game* class takes care of creating and calling everything needed to play the game, from Input Callbacks and Levelloading to the main draw loop. After a finished game, program returns to the menu, destroys the *Game* class and, if neccessary, starts a new game.

*robert.lau@rwth-aachen.de
†benedikt.scholtes@rwth-aachen.de
‡jan.haeusler@rwth-aachen.de

1.2 World & Physics

In our implementation the environment is wrapped into one 'World' class which manages object storing and - if necessary - the simulation of the physics of those.
For objects we differ between a pure world object, which does not get physics simulated and will only get rendered, and as a subclass physical objects, which - as soon as added to the world - gets registered with the physics engine[bul October, 2013]. All the various data necessary for each and every world object - position, rotation, geometry and so on - is stored in new instances object class and updated after every tick using a Bullet built-in callback.

1.3 Camera movement

The camera is controlled by a constraint that gets updated every tick. We use the position based method described by [Müller et al. 2007].

1.4 Terrain

The terrain in our implementation consists of single cubes. It is generated from an amount of image files, one for each layer, where the colors are depicting the type of entity at that position - start, terrain cube, finish, booster, and so on. For each white Pixel, the Terraingenerator generates the vertices and edges for a single Cube, and adds it to the level in the right location. The advantage of this lies in the modularity of this system. With minimal work you can add a new type of block, like a teleporter for example. You have to define a new colour, which represents this new block, add a new case to the levelgenerator, and create a new callback function. This gets then automatically called each time bullet registers a collision with the Ball.

1.5 Configuration

We use a static json Library (rapidjson[rap January, 2014]) for our general and level specific configuration files. We can configure a multitude of settings here, which are all loaded once at the program start and are accessible by all classes via Getter Functions. Besides Locations for all Textures, Shaders and so on, we save the assets we want to use for each object. This helped changing and experimenting with different Shaders without having to recompile the game every time you want to change a Shader or switch to another level. Furthermore, we save Keybindings, Physics attributes like gravity and maximum speed of the marble and graphics settings, for example the amount of MSAA or Anisotropy Filtering.

1.6 Effects

The very first an most basic effect we have implemented is the phong shader to start with some illumination. There we have replaced the ambient term by the texture loading because we wanted a high overall illumination. For the same reason we have deleted the diffuse term as well as the attenuation. What's left is the specular term as presented in the "Basic Techniques in Computer Graphics" lecture to create a spotlight.
The next effect we used was the simplex noise (cf. Figure 2)[Gustavson 2005]. The code itself was taken from [sim March, 2011], but the adjustment to make it well defined everywhere was still challenging. Finally we have figured out to take a 3 dimensional simplex filter with the x and z coordinates subtracted by the y coordinate as first two inputs and the time as third input.
Our last effect was the real time glow (cf. Figure 1)[gpu September, 2007]. It was done by deferred shading with two output textures. The first one was the render result the second was the glow texture with the parts to apply the glow to. The second image was then rendered twice through a Gaussian noise filter and finally added to the first render output as final result.

2 Problems and their solutions

During our work on the project we encountered various problems we had to face and solve. Especially in the early stages we had to get familiar the whole concepts of OpenGL and Bullet and therefore our technical knowledge was limited.
For instance the problem of applying the model, view and transform matrices in the correct order was causing us headaches as the ball was rotating around the world center instead of its own center at first.
Furthermore the correct integration and correlation of Bullet and the rendered world demanded alot of work at first.
Another problem was the aliasing of our textures, caused by the very high angle the camera has to the surface of around 160°. It was solved by applying the anisotropic texture filtering to get a clear surface even in the distance.

2.1 Persistent problems

The final release still 'features' some bugs and glitches.
For instance we randomly encounter the ball falling through the terrain, probably caused by some kind of race condition, as this usually is fixed by restarting the game.
Also the performance is somewhat bad in more complex levels, as we generate a new physics simulated box for each terrain block instead of combining those to primitives of bigger size.

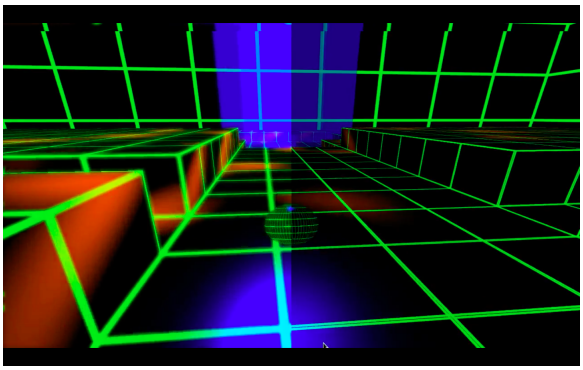


Figure 1: *glow vs. no glow*

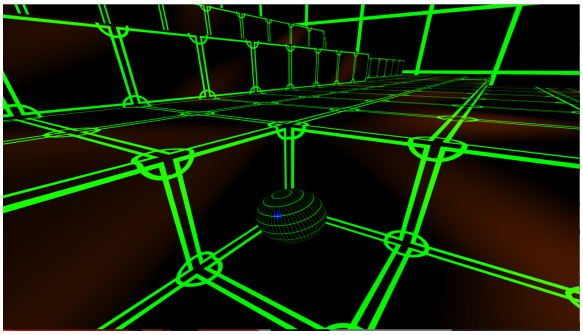


Figure 2: *Simplex3D Effect*

References

October, 2013. Bullet physics library. http://www.bulletphysics.org.

September, 2007. Real time glow. http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html.

GUSTAVSON, S. 2005. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*.

MÜLLER, M., HEIDELBERGER, B., HENNIX, M., AND RATCLIFF, J. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation 18*, 2, 109–118.

January, 2014. rapidjson. https://github.com/miloyip/rapidjson.

March, 2011. Shader library. http://www.geeks3d.com/20110317/shader-library-simplex-noise-glsl-opengl.