# Developing a Marble Race Game

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

Andreas Bring* Tarek Chebbi† Martin Josmann‡ Julian Schulz§

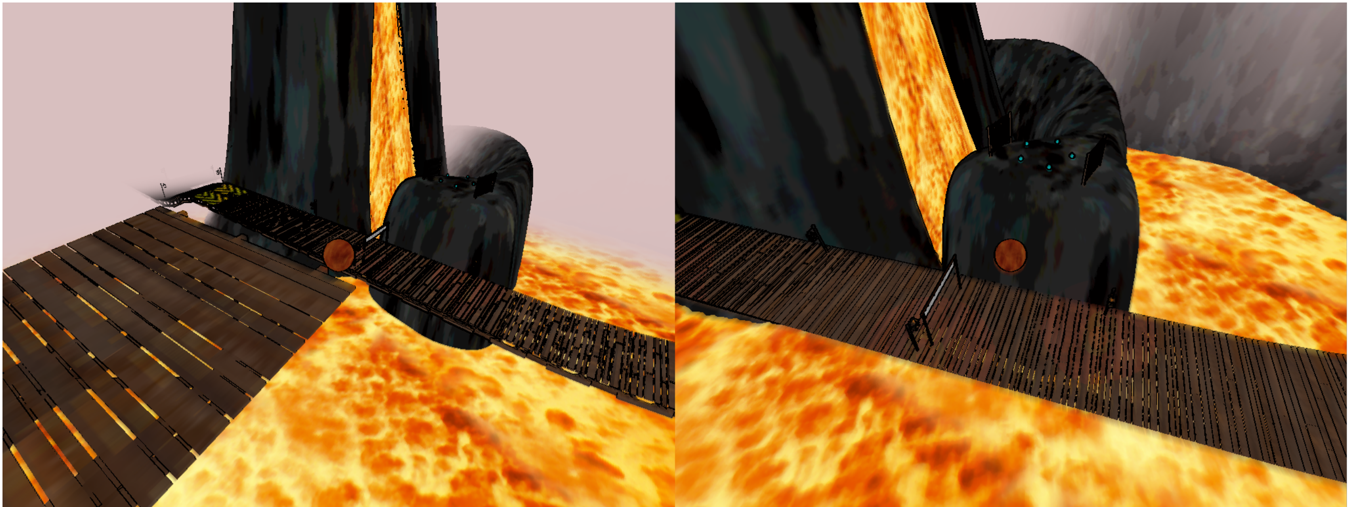

**Figure 1:** *Jumping from a high edge and motion blur*

## Abstract

In the practical course *Developing a Marble Race Game* the task was to create a racing game, using a marble as racing object and concentrate on the graphic output and the game logic (i.e. the code structure, defining how the game runs). While developing our game *Globus Ex* we employed various important parts of a game engine and shader based graphic effects. In this report, we will cover some of the implemented techniques superficially and state problems, which occurred while developing the game.

**Keywords:** game programming, marble race game, race game, deferred shading, scenegraph

## 1 General Information

*Globus Ex* is an adventure racing game, with a cel shaded look. The player controls a marble in an open world (in the limits of the level-design) and conquers different levels, some of which require puzzling, whereas others require fast thinking and speed in the style of a typical racing game. The main goal is to reach the end of the levels or complete them in the given time, ultimately finishing the game.

## 2 Graphics and Rendering

### 2.1 Deferred shading

The main look of the game is very cartoonish, drawing inspiration from games like *The Legend of Zelda: The Wind Waker*. For a real-time rendering of cartoon-like lights and objects, we changed the rendering method from the standard forward rendering to the for our needs more suited deferred rendering. The main advantages for us are, that this makes it possible to use more than just a few dynamic lights and it supports global information about a fragment in the deferred shader (used in the second render-pass), which is important for the cel shading outlines and the motion blur effect. However the deferred rendering disables us from using transparent objects in the game world, which is, considering the cartoon-like style of the game, no real issue in our case. For the two-dimensional overlay (i.e. the user interface) we disabled deferred rendering, trading unnecessary lighting and global fragment information for easy transparency.

Deferred shading works by rendering the data, which is normally combined and directly sent to the output by the graphics card, to different textures and then combining the information of these in a second render-pass and send them as output. In our case we used a total of five textures, which can be seen in *Fig.2*. Usually only three textures containing color, depth and normal information for the fragment are rendered, but to support motion blur we needed more information than the three textures could support, resulting in a *position texture* and a *previous position texture*. Since the depth can be calculated from the *position texture*, we do not render the depth to a texture separately. Also a fifth *data texture* stores if and how many lights affect the current fragment, as well as which material the object belonging to this fragment uses. Using multiple materials with deferred rendering is a common issue, which we resolved this way.

### 2.2 Motion blur

As mentioned before, we use two additional textures in the first render-pass to save the global position of the fragment in this frame and the last frame. With the projection and view matrices from both frames, the positions can be converted to screen-space coordinates

*andreas.bring@rwth-aachen.de
†tarek.chebbi@rwth-aachen.de
‡martin.josmann@rwth-aachen.de
§julian.schulz@rwth-aachen.de

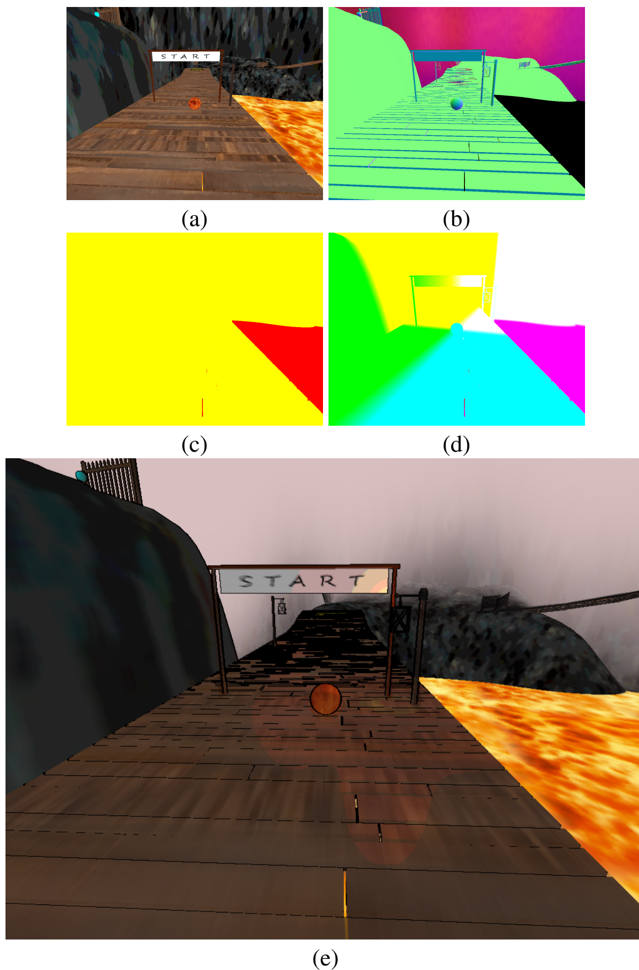


**Figure 2:** *Textures created in the first render-pass: (a) colors, (b) normals, (c) data, (d) position and previous position for every fragment. (Position and previous position textures only differ a little bit, unobservable in screen shots) And (e) the result of combining the five textures in the second render-pass and applying all graphic effects.*

and motion blur is applied, depending on if and how much the fragment has moved.

### 2.3 Light and shadows

We implemented support for two types of lights: directional and point lights. Each light has properties for ambient, diffuse and specular light emission as well as a shadowmap, which are combined with the information about the fragment and its material properties in accordance with the Blinn-Phong reflection model. To limit the area for which shadows need to be generated, we added fog, which starts after a definable distance and reaches its full opacity after another definable distance.

## 3 Game logic

### 3.1 Scenegraph

As a basic layout for all game objects we use a scenegraph structure, containing transformation-, light-, renderable- and special entity-nodes. Entities are designed to provide functionality for deriving classes, such as synchronization with the physics engine and automatically executed code, based on triggers (e.g. collision or every update). This way new entities with new functionality can be created very easily, and are kept separate from other game logic.

### 3.2 Performance and Space

While developing *Globus Ex* we faced the issue of low performance, due to not filtering and sorting the objects rendered. We overcame this issue, by implementing a renderer, which iterates over the scenegraph and registers all renderable objects with their associated data (textures, matrices). By sorting the objects by their normalmaps and textures we were able to boost the overall performance of the rendering, because the textures are not sent to the shader multiple times. The renderer also allows us to filter objects if they are not visible in the cameras frustum and it takes care of linking the lights and materials used in the scenegraph to the correct shader.

Because the renderer has all information on lights and renderable objects, it is also responsible for rendering the shadow maps of all directional lights to textures. Shadow map generation proved to be a bottleneck for point lights, which is why we decided not to implement point light shadows, also regarding the time, we had left.

Another issue were textures and geometry, which were loaded once every time they occurred in the level-file. To fix this, we implemented a resource manager, which loads resources only if they were not already loaded before.

### 3.3 Level files

For our level files we decided to use a XML-like structure, consisting of tags representing nodes in the scenegraph loaded from the file. It is read by a reader class, that converts the tags to methods which are then executed with the parameters of the tag. We chose this approach, to make the level files easily extendable, without the need of adding significantly more code. Additionally the various nodes of the scenegraph allow defining simple game logic in the level file, while the entities support the more advanced game logic by responding to events. E.g. the switch-node, which renders different scenegraph subtrees depending on the value of global variables can be defined in the level file without the need to alter existing or add code.

### 3.4 Third party libraries

In our game engine we utilized different libraries. Namely the bullet physics engine ([Bul October, 2013]) and the irrKlang audio library ([Irr ]). The former powers our real-time physics simulation as well as collisions, while the latter supports our game with sounds and music. Since audio was no requirement and physics and collisions are very complicated topics if they can not be simplified for certain applications, we decided to include these features as third party libraries only. Bullets ray-casting functionalities also power our smart camera implementation, by checking against blockages in the marbles and cameras vicinity, and calculating the best position for the camera on this basis.

## References

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

irrklang audio library. http://www.ambiera.com/irrklang/.