Developing a MarbleRace Game: StarBall

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

Felix Hermsen* Marius Wins† Pedro Louback Castilho‡

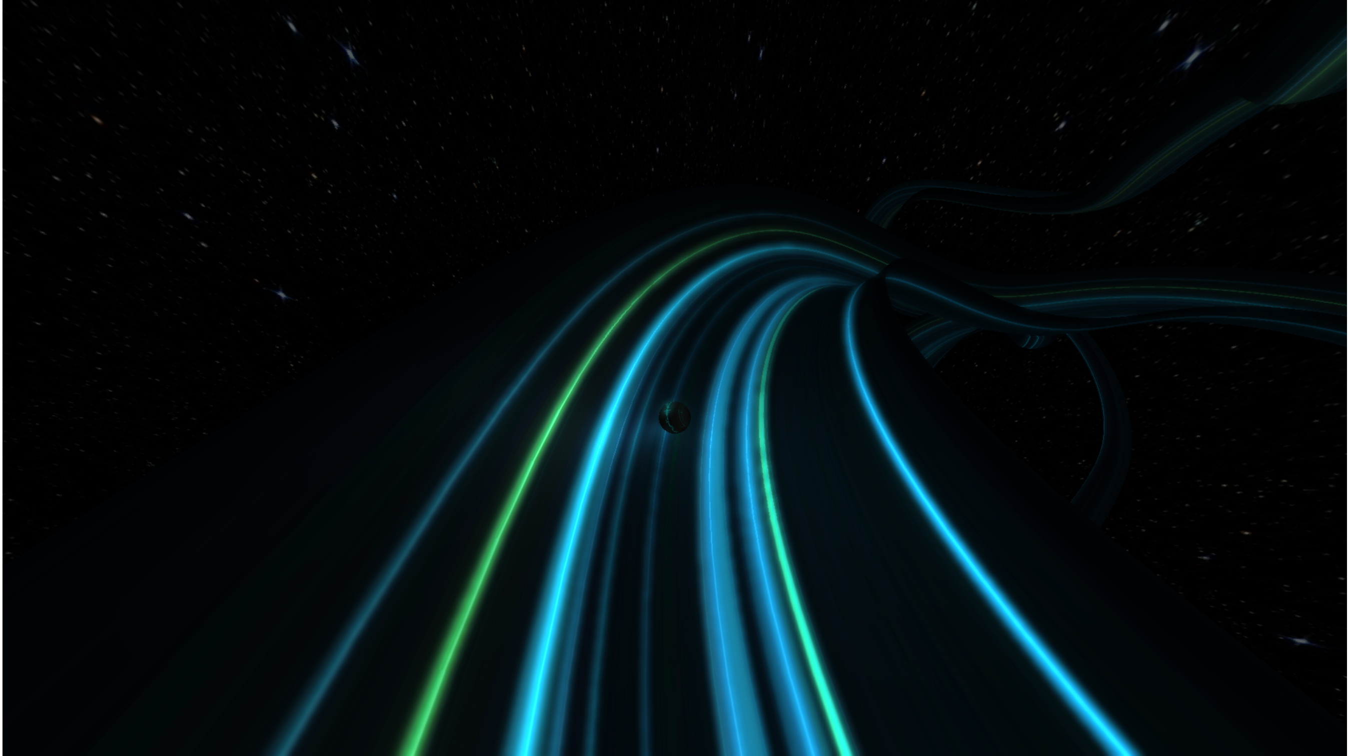


Figure 1: *An example of StarBall's gameplay.*

Abstract

This report describes the development of a game called StarBall as part of the "Developing a MarbleRace Game" practical project. Our game implements several techniques in order to achieve high graphical quality coupled with simple but challenging gameplay, while keeping the underlying design minimalistic. (cf. Figure 1). These techniques will be presented as part of this report.

Keywords: game programming, survival racing, glow, motion blur, deferred shading, real-time rendering

1 Gameplay Description

StarBall is a game of the *survival racing* genre, as exemplified by games such as *Nitronic Rush* [Nit 2011]. As such, the objective of the game is to reach the finish line safely. The challenge in achieving this objective is provided by the hazardous tracks of the game.

The player takes control of a marble-shaped mechanical being which must navigate complex tracks in a space-like environment by rolling. The tracks are typically flat or half-pipe shaped with many curves and ramps, and no guard rails are provided. Therefore, the player must be careful in their actions, lest they fall off the track.

2 Program Architecture

StarBall is structured as 3 main parts: The *graphics engine*, the *physics engine*, and the *interaction layer*. The functionality of each part shall be explained separately.

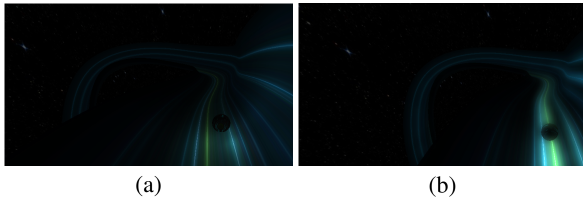
2.1 Graphics Engine Functionality

The Graphics Engine is responsible for generating the game's graphics. It is structured as a pipeline which takes inputs from the Physics Engine, and through many steps generates the image shown onscreen as output. The steps undertaken in the graphics pipline are clarified in section 4. The Graphics Engine is also able to automatically set the in-game camera without any player input.

2.2 Physics Engine Functionality

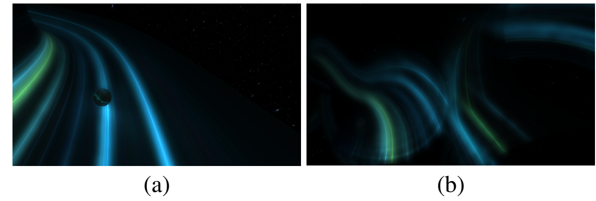
The Physics Engine computes the player's interactions with the environment. Using the Bullet Physics library, it runs a physics simulation where the player and the current track are present as solid objects and sends the motion state and position of the player to

*felix.hermsen@rwth-aachen.de
†marius.wins@rwth-aachen.de
‡plc@cin.ufpe.br



(a) (b)

Figure 2: *(a): Scene rendered with glow disabled. (b): Same scene with glow enabled.*



(a) (b)

Figure 3: *(a): Low level of motion blur. (b): High level of motion blur.*

the Graphics Engine, so that the game state can be rendered to the screen. The physics engine also handles eventboxes - rectangular box-shaped areas of 3D space where some effect is applied to the ball. Among possible effects are speed up, slow down, jumps, and player "death" (resetting the current level)

2.3 Interaction Layer Functionality

The Interaction Layer deals with the player's inputs. Using the GLFW utility library, it manages the game window, reads keyboard inputs and sends this data to the Physics Engine so that it might properly update the simulation.

3 Level Loading

Each level (track) is simply defined as a file containing the 3D polygon data for the track. The data from this file is used by the Graphics Engine to draw the track on the screen, and read by the Physics Engine using a custom loader in order to generate a solid object for the physics simulation. Eventbox data for each level may optionally be loaded. The polygon data used for the project was generated by us using Blender, while the eventbox data is entered by hand for each level.

4 Graphics

As previously mentioned, the graphics engine follows a pipeline structure. There are two main stages corresponding to the two main rendering passes, and several minor steps corresponding to graphical effects. Rendering uses deferred shading, specifically the G-Buffer approach [Saito and Takahashi 1990].

In this approach, we write scene properties needed for the lighting computation into a series of textures (the G-Buffer) and then in a second rendering pass the lighted scene is computed from these textures. The shading model employed is Phong Shading [Phong 1975]. After computing the lighted scene, motion blur and glow are added onto it following the processes detailed below.

We chose to use the G-Buffer approach not only in order to easily support arbitrary numbers of lights without a significant performance hit, but also to simplify the implementation of the graphics effects we use, as it becomes trivial to combine the output of several frames or renderings during the second pass.

4.1 Graphics Effects

StarBall makes use of two major graphics effects: Glow and Motion Blur.

4.1.1 Glow

The use of the glow effect is crucial for the surreal environments of StarBall. Our implementation of glow is based on the implementation described in Chapter 21 of "GPU Gems" [Fernando 2004]. Namely, the glowing parts of the scene are rendered into a smaller glow target and blurred. This technique leads to artifacts in distant regions due to these regions possibly not being rendered in a continuous fashion in the smaller target, we add a distance-based exponential fog to the scene. In order to further reduce artifacting and improve aesthetics, a small motion blur, analogous in implementation to that described in the next section, is applied to the glow target, which results in a pleasant visual effect. (cf. Figure 2)

4.1.2 Motion Blur

The motion blur effect is employed by us in order to increase the sense of speed imparted by the game to the player. Our motion blur effect is achieved by maintaining an accumulator texture to which previously rendered frames are added. Each frame is added to this accumulator with an opacity based on the current speed of the ball, effectively making the blur disappear below a certain "low threshold" speed and making the blurring not increase anymore above a "high threshold" speed. The amount of blur when the player's speed is between the threshold speeds is determined by cubic Hermite interpolation between a constant "low level" and a constant "high level" of blur. (cf. Figure 3)

4.2 Camera Control

As mentioned in section 2.1, the graphics engine is able to automatically set the camera without the player's input. This is done by taking the direction of the ball's movement and interpolating between the camera's current orientation and the ball's movement orientation, such that the camera orientation is always close to the ball's, but very fast orientation changes are filtered out.

The position of the camera is set such that the player is always looking at the ball from behind, and the distance from the ball to the camera is set according to the ball's speed - when the ball's speed is higher, the shot becomes wider, and conversely as the ball slows down the camera closes in.

References

FERNANDO, R. 2004. Real-time glow. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, GPU Gems. Pearson Higher Education, ch. 21.

2011. Nitronic rush. http://nitronic-rush.com.

PHONG, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June), 311–317.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 197–206.