Swinging Game Production Report

[Company name] | [Company address]

[Document subtitle]

Sergej Frank

2019

Contents

[Introduction 1](#_Toc535133796)

[Research 1](#_Toc535133797)

[Architecture 1](#_Toc535133798)

[Changes 1](#_Toc535133799)

[Testing 1](#_Toc535133800)

[Reflection 1](#_Toc535133801)

# Introduction

This document describes the production report of ‘Swinging Game’ within the scope of assessment 2 of the course Advanced Games Programming. As the task demanded, the game was produced using DirecX11 and object-oriented C++ with Visual Studio. It also describes changes that have been made differing from the original game design document and a documentation of tests and progress during the development process.

# Research

The only additional research besides information I learned from the tutorials was, how to create a lava shader. I read about HLSL shaders in “HLSL Development Cookbook” to have a better understanding about shaders in general. Also, I googled about moving vertices for lava and used this resource to adapt my own shader: <https://gist.github.com/Davidiio/7b7e2b59c7bf5c0e303174d7be6d1fd5>

Resulting in a lava shader using 3 different textures.

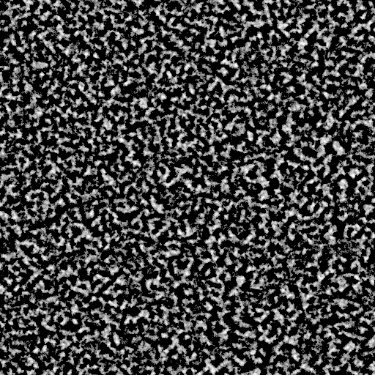
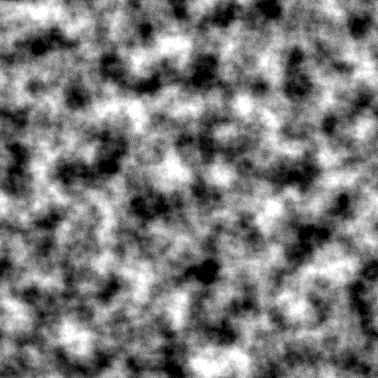
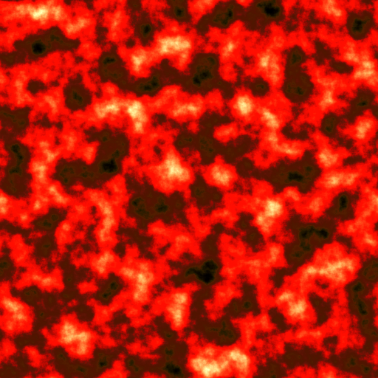


Figure 1.1 Lava Diffuse Figure 1.2 Lava Normal Figure 1.3 Lava Noise

# Architecture

Mainly the two key techniques I used were the Singleton pattern to ensure there was only one instance of DirectX, that can interact with several classes and scene management together with object pooling, as we have learned from the tutorials. All objects in the game derive from GameObject. The ones who can be drawn, have also a Model and two colliders, a bounding sphere and a mesh collider. The scene root created in the main class holds all other scene objects as children nodes and calls the update and draw methods for each child every frame. That way it is much easier to create a scene, because the objects need only to be loaded once and added as children. Simply one call for the scene update and draw is necessary to do the same for each child.

# Mechanics

One easy mechanic to implement was the death of a character touching the lava.

Every time a player dies, he gets plus one death on the death counter seen in the left upper corner, as can be seen in *Figure 2*.

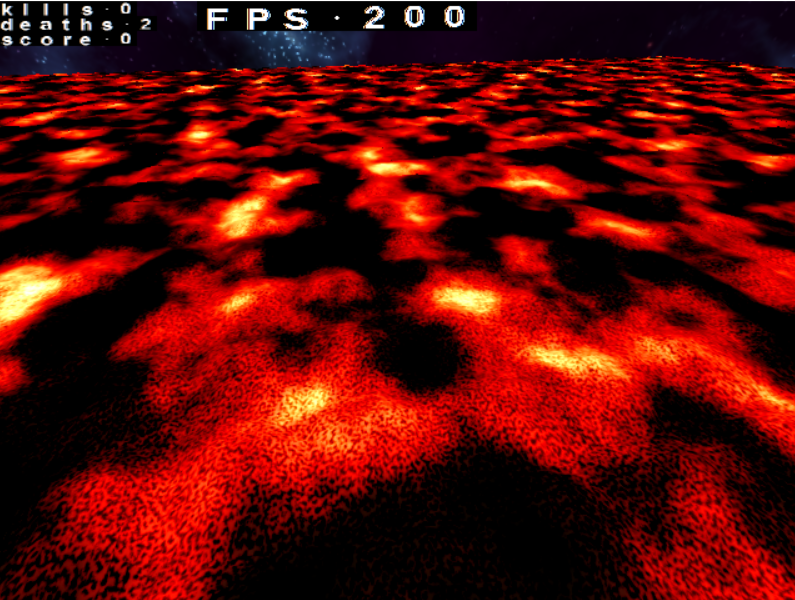


Figure 2, Lava and death

Every time a player dies, he gets plus one death on the death counter seen in the left upper corner, as can be seen in *Figure 2*. What also can be seen in *Figure 2* is the FPS (frames per second) counter, that show the average FPS, but not more than 200. The equation for that was easy to write, once the timer class was implemented.

FPS equation:

min((int(1.0 / delta\_time())), fps\_limit);

Another important mechanic is jumping. For that I used a constant value to apply gravity to every non-kinetic GameObject in its update method. I accumulated the time, the object did not touch the ground using delta\_time. Then I used a common gravity equation.

Gravity equation:

falling\_velocity = gravity \* (air\_time\_\*air\_time\_);

The only thing that I had to do now, was checking in the players update if he is airborne and execute the jump method when the spacebar was pressed.

What took me the most time was collisions. Sphere collisions were easy to implement together with scene management using a root object. Nevertheless, it was a problem for me to check the ray to triangle collisions. Every time I tried to check for the collision, it turned out the ray pointed in the wrong direction. In the end I could manage to do it with my own interpretation of ray equation.

Ray equation:

The main mechanic hooking would look like shown in Figure 2, where a similar game is shown called “Energy Hook”.



<https://bagogames.com/energy-hook-review/>

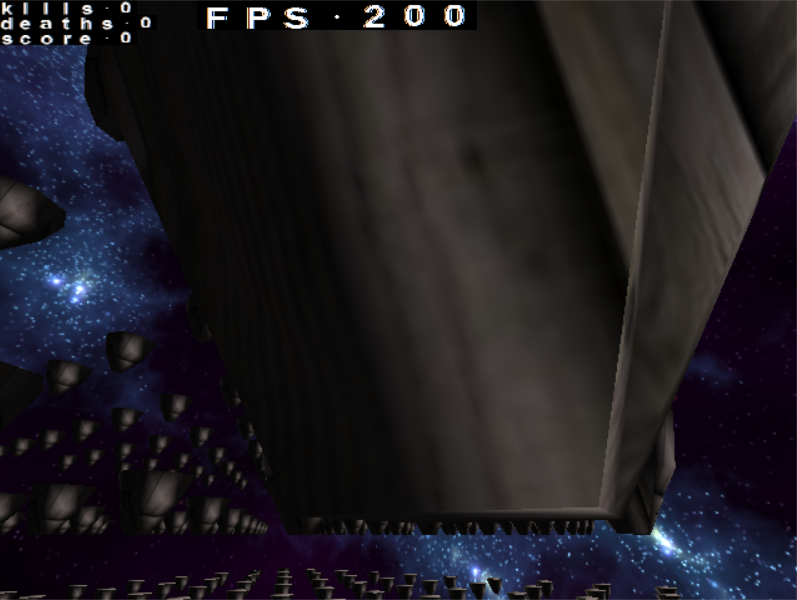
# Changes

I have done some major changes due to several problems that occurred and cost me a lot of time I did not have.

The three main game mechanics that describe the game were not implemented. Rather than trying to implement them, I had a hard time to implement per-triangle collisions. Although the player can move, look around the map and jump, there is no hooking, air acceleration or cutting.

# Testing

It was significant to test collision in order to get it work without any faults. I have tested it many times by moving the player with the first-person camera towards the objects from all different directions illustrated in *Figure 3*.



*Figure 3* Player collides with object using a ray and triangle-collision

# Reflection

The whole project was harder to develop than I thought it would be. The time we had to accomplish it was roughly enough for me. Most of the problems occurred while trying to implement collisions. Perhaps I should not have tried to work on one part for 2 weeks, because this was, what it took me to get collision with scene management to implement.

My biggest problem was, that I got ill for two weeks and could not concentrate to program fast enough.

Besides that, I had to decide what to do, when only one week for development was left and my code architecture was not good enough to easily add new features. So, I gave up on finishing the game to have the original game specifications and went on implementing things to get points for the assessment.

After I researched about how to smoothly implement interactive components with game object, I found that it would have been better to do it with the **component system**, but one week was left and I have not had enough time to change my whole code architecture.

In the end I tried to do the easier things left on my task list to get as much points as I could.

Github repository <https://github.com/radlog/SwingingGame>

Asset List:

<http://free-texture-site.blogspot.com/2010/10/free-wooden-crate-texture.html>

<http://www.custommapmakers.org/skyboxes.php>

<https://www.textures.com/download/substance0137/132297>