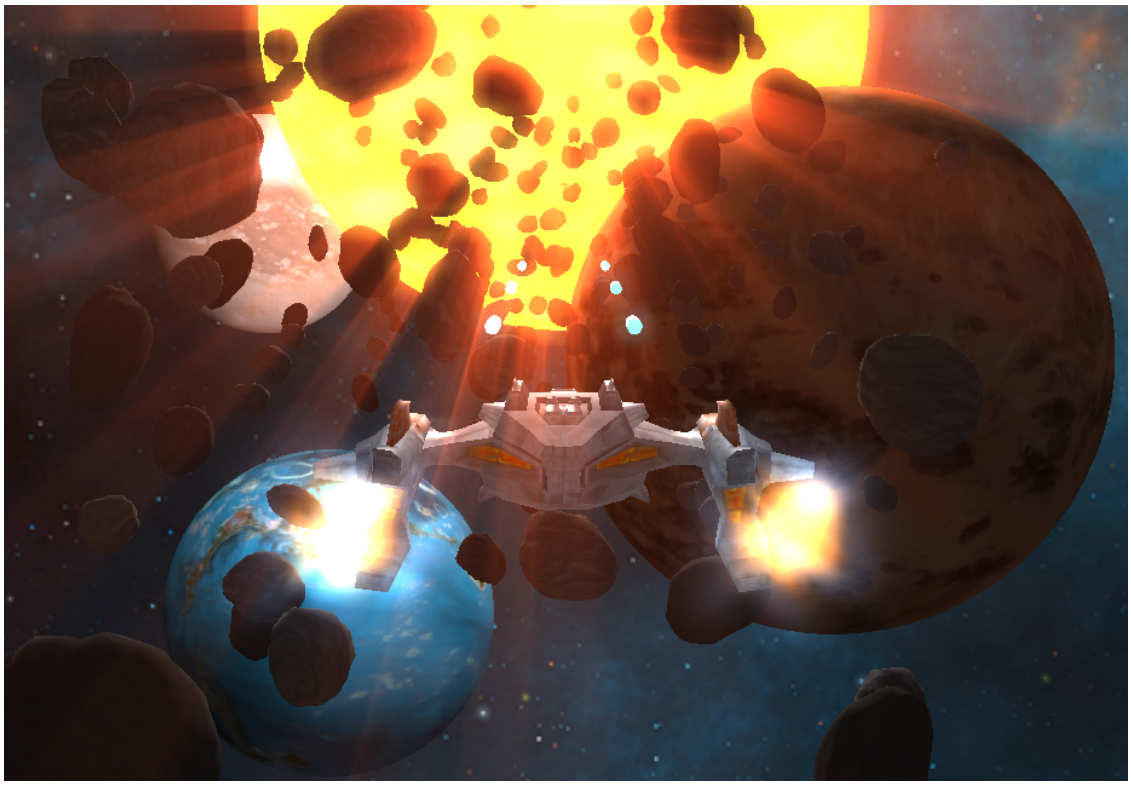

A Case Study of a 3D Space-Shooter Game

ULMO the Star Explorer

Bachelor of Science Thesis in Computer Science and Engineering

Oskar Dahlberg
Hampus Ericsson
Johan Hasselqvist
Anton Josefsson
Henry Ottervad

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2015

A Case Study of a 3D Space-Shooter Game
ULMO the Star Explorer

O. DAHLBERG,
H. ERICSSON,
J. HASSELQVIST,
A. JOSEFSSON,
H. OTTERVAD,

Examiner: ARNE LINDE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: An ingame screenshot of the game *ULMO the Star Explorer*.

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Abstract

This Bachelor's thesis treats the development of ULMO the Star Explorer, a graphically intensive space-shooter game set in three dimensions, which is developed using the framework Microsoft XNA. The project was carried out as a case study, and the thesis presents and compares different techniques that can be used to solve the problems we identified.

The main focus for this thesis is to present the graphical effects used to create a graphically intensive game within a limited time frame, but it also presents the different solutions chosen to create the game engine used by the game. The thesis covers several common graphical effects used by games, such as Bloom and Motion Blur, and presents two novel algorithms: one algorithm for lighting of spheres, and one algorithm to generate the game world including a path within the three-dimensional space of this world.

ULMO the Star Explorer does not contain all the features that were originally planned, but is still a fully functional game that can, given the time frame and the number of project members, be seen as a visually pleasing game.

Sammandrag

Denna kandidatrapport behandlar utvecklingen av ULMO the Star Explorer, ett grafikintesivt rymdskjutarspel i tre dimensioner, som är utvecklat med hjälp av ramverket Microsoft XNA. Projektet utfördes i form av en fallstudie, och rapporten presenterar och jämför därför flera tekniker som kan användas för att lösa de problem som vi har identifierat.

Rapportens huvudfokus ligger på att presentera de grafikeffekter som användes för att skapa just ett grafikintensivt spel inom en begränsad tidsram, men den innehåller även de lösningar vi valde för att skapa spelmotorn som spelet använder. Rapporten täcker flera vanliga grafikeffekter som används i spel, som till exempel Bloom och Motion Blur, och presenterar två egenutvecklade algoritmer: en för ljussättning av sfärer och en för generering av spelvärlden med en rutt i tredimensionell rymd genom denna värld.

ULMO the Star Explorer innehåller inte alla de funktioner som ursprungligen var planerade, men är trots det ett fullt fungerande spel och kan, med projektets tidsram och antal projektmedlemmar i åtanke, ses som ett visuellt tilltalande spel.

Contents

1

Introduction

This thesis discusses the development of a three-dimensional space-shooter game, and the implementation of several graphical effects and techniques that were used to make the game look interesting. When developing the game we decided that, if there is a conflict between visuals and realism, visuals will be prioritized if it seems likely to result in a more engaging game. This introductory chapter covers the background, purpose, problems, limitations, and the method of working that form the basis of this Bachelor's thesis.

During the development process two novel algorithms were developed: one algorithm for the generation of levels, and one algorithm to properly light the planets by the sun, which is in the center of the scene.

1.1 Background

The development of computer games has come a long way since it started. According to Moore's law, the number of transistors will double every two years, thus increasing the computing power, making it possible to create better effects. As power increases, the public's demands on the games are rising as well, and the cost to produce a modern game can stretch to several hundred million dollars with teams consisting of up to 500 people working on the same game [Cross 2014].

One of the first popular game genres was the space-shooter genre, with games such as Space Invaders in 1978, or Asteroids, released the year after. This project takes inspiration for the gameplay from games such as Star Fox, a space-shooter game for the Super Nintendo Entertainment System console that was released in 1993, where the player follows a predefined path in third person, shooting enemy ships in front of the player.

1.2 Purpose

The purpose of this thesis work is to conduct a case study, in which a group of five team members work for four months, in order to develop a 3D shooter game set in space, with emphasized focus on graphical effects.

1.3 Problem

When creating a game, there are a couple of problems that have to be considered. For this project, we decided to focus on three different parts, namely how to design the game engine, what to consider when implementing different graphical effects, and how to efficiently work as a group on a heavily time-constrained project.

Game Engine. How do we structure the code in order for five people to efficiently implement features iteratively, seamlessly, and concurrently? How can we solve common game-programming problems, such as collision, and level generation?

Graphics. How can we implement certain specific graphical effects in a visually pleasing, yet efficient, manner? Which of these effects are easy to implement, and which are difficult?

Collaboration. How do we collaborate as a group of five members on the same software? How do we select which features to implement, and which to skip, with regards to the time constraint set on the project?

1.4 Limitations

In order to create a game within the given time frame with five members, some limitations have to be made. Given that the main focus for the project is graphics, other areas, such as artificial intelligence, audio and advanced gameplay features have been more restricted than the implementation of graphical features.

Additionally, since no group member has any significant experience in designing 3D models, it was decided to mainly use free models from external sources, rather than design models on our own. This limits the amount of models we have at our disposal and also lowers the consistency of models, since most models come from different sources and have varying styles.

1.5 Method

While the thesis itself is going to explain and discuss the algorithms and thoughts, this section covers some of the tools used during the development of this project.

There are many ways to structure the working process of a group; the waterfall model, spiral model, or extreme programing are just a few alternatives. We decided to work with an agile process, a spiral model, with prioritized tasks. These tasks were implemented and tested one by one, iteratively, and structured using Trello, which is a billboard web application that helps groups easily overview and organize such tasks.

We chose to develop the game in Microsoft's framework XNA, rather than an existing game engine, such as Unity or Unreal Engine. The reason for this is that XNA requires the developer to write most of the game engine and rendering themselves, while still supplying most of the underlying code for things such as importing models and textures, gathering input, and setting up for the graphics pipeline. This allows for more freedom in the specifics of the implementation compared to when using Unity or Unreal Engine, while it also provides more learning possibilities.

2

Game Engine

This chapter presents the decisions made when creating the game engine. It discusses the differences between programming games and programming event-driven applications, and presents common game engine problems and solutions. It also covers the techniques used when generating levels for our game. Each section presents and analyses the results from being implemented into the game.

2.1 Complexity Management

Games are large software projects, and even very small games usually require more than a thousand lines of code [Schuller 2010]. Thus, it is imperative to use methods that help manage the resulting code complexity; two such methods are object-oriented inheritance and entity-component systems.

Standard object-oriented inheritance has the advantage of being familiar to all developers of the project and is a very well-documented method. However, it has a disadvantage in that it can, at times, make the code difficult to refactor in later stages of the project. An entity-component system, on the other hand, favours composition over inheritance and reduces this problem by separating objects into entities and functionality into components [Hall 2014], which adds flexibility; changing and adding functionality does not grow difficult at the same rate as for standard inheritance. However, one disadvantage is that more code may be required and, thereby, the implementation time could increase.

Results and Discussion. We decided to use standard inheritance, since the scope of the project was small and the project was initially believed to not require very complex logic. This decision turned out to be acceptable, and we did not run into many problems with the flexibility of the code. However, it may have been advantageous to implement an entity-component system in case we want to further develop the game in the future.

2.2 Game Loop

Unlike event-driven software, real-time video games are run continuously. This is because the game interacts with the player using a tv screen or a computer monitor, and these are updated at very high frequencies. A normal update frequency is usually between thirty and sixty frames per second, and when programming a real-time game that is supposed to run at a frame rate of sixty frames per second, the computer has only one sixtieth of a second to compute what should happen in the next frame. Real-time game engines are built around a central loop that is called a game loop [Schuller 2010], from which most code in the engine is called. This loop runs as fast as it can in order to reach the target frame rate.

In Microsoft's framework XNA, the game loop is responsible for calling two methods: update and draw. Update is responsible for all game logic, such as input and physics, while draw is responsible for rendering the game to the screen. The game loop can be set to run in two ways: with a fixed time step, or with a variable time step. If the game runs with a fixed time step, the game loop will be called at exactly the interval specified, and after update and draw has been processed, the loop waits until the next time step is reached and repeats the process. However, if update should take too long to process, the game loop will skip calling draw until it catches up. If the game runs with a variable time step, update and draw will be called sequentially as often as possible, which means that the game can run at wildly varying frame rates.

Results and Discussion. We decided to run our game with a fixed time step set to sixty frames per second. Unless complicated physics are needed, it may be worthwhile to run the game in variable time step, but we chose not to implement this, as it might lead to problems and makes code analysis more difficult. One of the goals of the project was to have a smoothly running game at this frame rate, which, depending on hardware, we largely accomplished.

2.3 Game States

A game can usually be divided into several different states, e.g. title menu or gameplay. This presents an architectural challenge; it should be easy to expand the code with a new game state, while also avoiding repeating code. Schuller [Schuller 2010] presents a naive implementation using a switch statement, but dismisses it, since it is neither easily extendable nor does it conform to the DRY (Don't Repeat Yourself) principle [Hunt 1999].

The only other approach we found documented is to use a state machine. An implementation of this could be to use an abstract class, with every state deriving this class, and having the main game loop keep track of the current game state. This design allows the developer to easily add new game states, and management of the state transitions could be delegated to the states themselves, allowing them to decide when a new state should be entered.

Results and Discussion. We chose the state machine approach and found it suitable for our needs. The game does not currently require many states, but one could speculate that transition management could quickly grow more complicated with more states, since each state might need to transition to multiple others, possibly resulting in an exponential growth.

2.4 Input Management

Good input management is important for several reasons;like most other game architecture problems, the code needs to be easily extensible and modifiable. It is also preferable if the input can be customized by the user.

There are two main ways for input to be handled: polling and interrupt requests. Polling is software driven, and means that the software repeatedly polls, or asks, the hardware if any input is occurring. In a video game, for example, the game loop could repeatedly ask the hardware if the space button is pressed, and if so, the game character is made to jump. The obvious drawback of this method is that the input is only checked at the rate the game loop is called, so if any input is made between the game loop iterations, the input will be lost. Fortunately, this should not occur frequently, since the game loop usually runs very fast. Interrupt requests, on the other hand, means that the hardware tells the software that something has happened. This means that input can not be lost, but the software loses some control. A situation where this might be desired could be when reading the motion of the mouse pointer, and the game needs to know if the mouse is hovering over a specific area; if the mouse is moved fast enough, polling might miss this input.

Assuming that polling is used, a common way to handle input might be the following example:

```
void HandleInput()
{
    if(keyDown(W)) moveForward();
    else if(keyDown(S)) moveBackward();
    if(keyDown(Space)) jump();
}
```

This is however not very flexible. For instance, there is no easy way to remap the input according to the user's needs. There is also no way to decouple the input from the functions being called. Nystrom [Nystrom 2014] presents a solution to the problem called the command pattern, which is a pattern that represents the function calls as objects. He also presents a comfortable way to associate the input commands with certain game objects, which can make input-based AI and handling several player characters easier.

Results and Discussion. We chose to use the polling method, since our game has no specific situations where input might be missed. We also implemented the command pattern and found it useful for modifiability of the input. For example,

we decided to program support for game controllers at a late stage of development, and the command pattern made this very straightforward since remapping input becomes trivial when using it.

2.5 Collision

In a game that focuses on shooting down enemies, or avoiding being shot down, a system to handle collision is essential. This section describes bounding volumes, a way to detect colliding objects, our reasoning behind collision in a space-shooter game, results from the game, as well as a discussion about what could have been done differently.

2.5.1 Bounding Volumes

A common way to detect collision is by using bounding volumes (i.e., shapes that enclose objects or other bounding volumes) [Akenine-Möller 2008, pp. 647-648]. Bounding volumes can be of any shape, such as a sphere, an axis-aligned bounding box, an oriented bounding box, or any polytope, where more complex shapes and higher precision come at the cost of computation time.

In a space-shooter game, where objects move fast and are far away from each other, the shapes of the bounding volumes do not have to be too precise, as any minor faults in the collision detection will likely go unnoticed. This means that most objects can have a bounding volume represented in a simple form, in our case a sphere, since sphere-sphere intersection is very efficient.

2.5.2 Results and Discussion

The approximate behaviour of our collision detection system is satisfactory, but still requires a lot of power from the CPU, due to all objects of one type being tested against all other relevant types (e.g., the player ship tests collision against all enemy ships, bullets, and asteroids) every update. We decided that, since the game does not perform any other heavy tasks on the CPU, this is acceptable. Figure 2.5.1 shows the bounding spheres used for collision in our game.

Currently, a lot of collisions are tested each time the game logic updates, and most of the collision tests are not very relevant. It would be possible to reduce the number of computations if we were to implement some sort of bounding volume hierarchy and only test for more relevant collisions.

Figure 2.5.1: A screenshot from the game, displaying bounding spheres around asteroids and enemy ships.

2.6 Procedural Level Generation

This section covers some basic procedural content generation concepts, our implementation of procedural level generation, followed by a discussion about the results.

Procedural content generation (PCG) is a concept that we have chosen to define as creating content through algorithmic means with limited human input, a definition which is discussed more thoroughly by Togelius et al. [Togelius 2011a], and has some advantages over manually creating content [Togelius 2011b]. Using PCG it is possible to, depending on the implementation, automatically generate huge amounts of content that would have taken a significant amount of time to create manually.

Togelius et al. [Togelius 2011b] propose a number of different classifications of PCG algorithms. We will focus on two sets of algorithms that can be said to be each others opposites: constructive algorithms and generate-and-test algorithms. Constructive algorithms make sure to generate the content in such a way that it always is correct according to the specifications. Generate-and-test algorithms, on the other hand, repeatedly generates content, checks it, and throws it away if it does not fulfill the specifications, until suitable content has been created.

2.6.1 Level Generation

Each level in the game guides the player along a path through a planetary system. It was decided early on during development that the path should be able to be placed closely around planets in order to achieve interesting visuals, such as a sun rising up above the planet's horizon. Generating levels where the path is placed close to planets became the main goal for the development of the level generation algorithm.

In order to create a functional level generation algorithm within the project's time

frame, the levels were kept pretty simple. A level consists only of these elements: waypoints representing the path the player is moved along, planets which the path moves around, planets and moons which exist solely for decoration, a sun that acts as a light source, asteroids for the player to avoid or destroy, and a background.

The final algorithm divided the level generation into three conceptual parts: main path generation, decoration and debris generation, and content randomization. Main path refers to the path and the planets it will be placed close to, and is the minimum requirement for a level. Decoration and debris refer to objects such as the sun, asteroid belts, and decorative planets that the player will not travel close to. In the context of content randomization, content refers to the models and textures used by the models in the level.

2.6.2 Main Path Generation

The main path generation algorithm was developed with some rules for correctness in mind: the path should not pass through a planet, the path should not move erratically, and the path should not move between planets in a strange order. An example of a strange order is when the path starts at a position, A, then moves to the planet furthest away from A, then back to the planet closest to A.

Two approaches for the main path generation were tried. The first approach was a generate-and-test algorithm, where all the planets were generated first, followed by the path. Randomly generating the planets presented no problem, but generating the path between the planets according to our rules of correctness proved to be difficult while keeping the complexity of the algorithm low, therefore, this method was abandoned.

The second approach uses a strictly constructive algorithm; each planet, and consequently the path, depends on the previously generated planet and corresponding path. Due to this, the position of the first planet is arbitrary and, unlike the previous approach, this method does not need to test the position of any planet or any part of the path.

In detail, the algorithm works as follows: first, the number of planets, that will be part of the path, is randomly chosen from an interval of three to five. The first planet is then placed at a fixed position, and waypoints are generated in a sine curve of random length and rotation around the planet. The next step is to generate a direction vector from the two last generated waypoints and an offset vector from the first planet's position to the last waypoint. The direction vector is then offset by individually randomized (x, y, z) magnitudes of the offset vector. The next planet is then generated at a bounded random distance in the direction given by the, now offset, direction vector, as shown in Figure 2.6.1, and waypoints are also randomly generated in a sine curve around the newly created planet. Finally, waypoints are generated between the last waypoint of the previous planet and the first waypoint of the new planet. This is repeated until there are no more planets to generate.

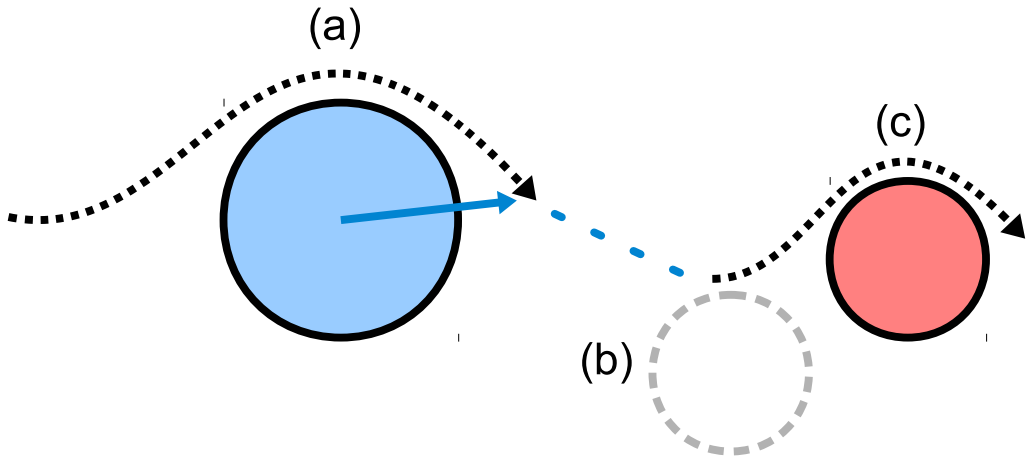

Figure 2.6.1: This figure shows how and offset vector, represented by the blue arrow, affects the planet placement. (a) shows the waypoints around the first planet. The black arrows shows the tangent of the last two waypoints. (b) shows where the next planet would have been placed if it had been placed on the tangent. (c) shows the where the planet is placed after offset.

2.6.3 Decoration and Debris Generation

After the main path has been completed, the rest of the level's objects are generated.

Sun and Decorative Planet Generation. The sun is generated by a generate-and-test algorithm, by randomizing the size and continuously randomizing the position within a boundary, specified by the previously generated planets, until the position is not too close to a planet or the path, after which the sun can be placed. If the position is not suitable after a certain amount of attempts, the boundary expands. The decorative planets are generated in the same way, with the only difference being that they cannot be too close to the sun, and the process repeats until a random number of planets has been generated.

Asteroid Belt Generation. The asteroid belt is generated in two steps. First, the asteroid belt is initialized, then the asteroids are positioned. The initialization function requires two points in space, representing a start point respectively an end point, as well as a density value between 0.0 and 1.0. Together, these values decide where the asteroid belt should be located and how many asteroids it should have. The number of asteroids depends on the distance between the two points and the density value (see Figure 2.6.2). After initialization, the asteroids can be positioned.

The asteroid positioning is done by generating a random point on the line between the start and end points, followed by randomly generating a position in a sphere, where the center is the random point. If the position is too close to another asteroid, the process is repeated. The radius of the sphere depends on how close to the middle of the line the random point is, and how many times the algorithm has generated a position too close to another asteroid. This process is done for all asteroids that should be created.

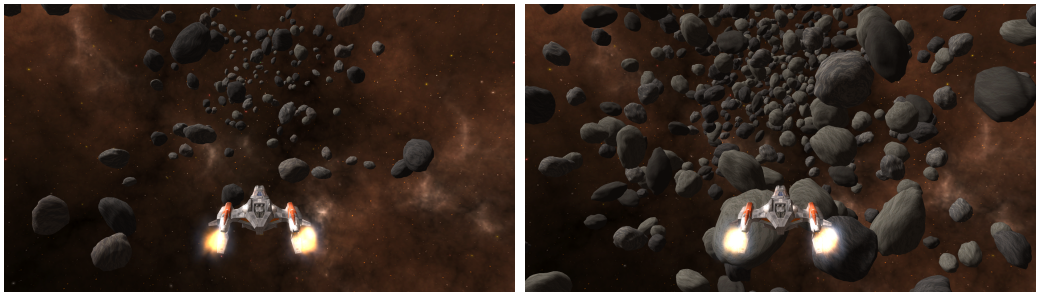

Figure 2.6.2: Two asteroid belts with different density values. *Left:* A belt with a density value of 0.0. *Right:* A belt with a density value of 1.0.

Orbital Asteroid Belt and Moon Generation. Orbital asteroid belts orbit in a ring shape around a planet, and the number of asteroids generated depends on the radius of the planet the asteroid belt should orbit. Moons, like the orbital asteroid belts, orbit around a planet.

2.6.4 Content Randomization

At this stage, the level has already been created, but the models and textures of the different objects are not set. Each planet, moon, and sun is assigned a model, and a texture from a pool of planet and sun textures. The models are scaled when the level is rendered, so that they will have varying sizes. The asteroids in the asteroid belts have to share models and textures as they may number into the thousands; having that many unique models and textures is not feasible, both due to the combined file size and the time that would have to be spent creating or finding the models and textures. The background texture, randomly chosen from a pool of backgrounds, is randomly oriented so that the player will not see the same part of the background if it appears in several levels.

2.6.5 Results and Discussion

Overall, the level generation works as intended; generated levels often produce interesting and pleasing visuals, as can be seen in Figure 2.6.3, and the levels are usually not strange looking. The main path fulfilled our rules of correctness (see Section 2.6.2 Main Path Generation), and, as a bonus, the levels are also generated very quickly.

Although attempts were made to make the levels visually distinguishable from each other, levels quite often look similar to each other. This is likely due to the lack of different types of objects in the level; having only a few types of elements to put in a level keeps the level generation algorithm fairly simple, at the cost of variety.

Basing the path of the level on waypoints was a bit of a shortcut. When the player passes a waypoint, the player is rotated to look directly at the next waypoint. This means that the path requires many waypoints in order to be sufficiently smooth

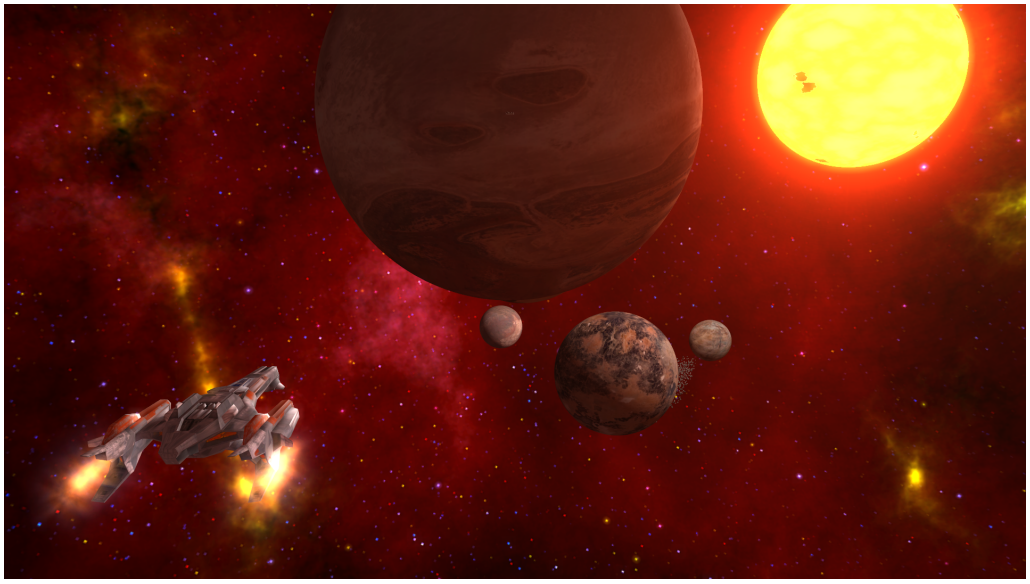

Figure 2.6.3: A visually pleasing level (screenshot taken during actual gameplay).

at curves, such as when orbiting a planet. We do have a very simple smoothing implementation that can be activated or deactivated on a per waypoint basis. The smoothing algorithm makes the player rotate slowly towards the next waypoint. However, the technique is not reliable enough for when orbiting around planets, as it works best when waypoints are far away from each other, and if the waypoints are too far from each other when moving around a planet, it is possible that the smoothing will let you pass straight through planets. We do use smoothing at the waypoints between two planets, as those paths are straight enough and do not need many waypoints. Representing the path using some sort of curve, like parametric curves or Béizer curves [Farin 2001], would have allowed the path to be smooth at all times and would probably require less waypoints.

Another limitation in the level is that the path does not branch. The player's action have no consequences on how the level will play out. Allowing a more dynamic level would likely make gameplay more fun and rewarding, as interesting scenarios could be created. One could imagine a situation where depending on if the player is able to destroy a planet or a giant enemy base, the path changes. Perhaps the path could loop until the player manages to achieve some objective, for example destroying an enemy base. However, this would mean that we would have to create the corresponding types of contents used in the objectives, which would take further time.

3

Graphics

This chapter will begin by briefly introducing some basic rendering concepts as well as a discussion on the models and textures used in the game, before going into detail about several techniques that we felt were relevant to explore in order to create interesting graphical effects. All sections, except for the rendering introduction, will also present and discuss the results after having been implemented in the game.

3.1 Brief Introduction to Real-Time Rendering

Three-dimensional graphics are typically represented in one of three ways: triangles, voxels, or curve surfaces. Triangles are probably the most common of the three, and the graphics hardware is optimized toward processing these. Each triangle is represented by three vertices, which are points in three-dimensional space. Voxels, in the context of real-time graphics, are essentially cubes on a grid. In contrast to triangles, which are very good at representing graphics with large amounts of empty space, voxels are superior for representing graphics with large amounts of non-empty space, which makes voxels common for terrain rendering. Curve surfaces are defined by curves, and can as such represent smooth detail several magnitudes more complex than triangles and voxels, but lack the flexibility of the previous two when the surface is not smooth.

When rendering three-dimensional graphics in real-time, there are several steps that are commonly referred to as the graphics pipeline [Marroquim 2009]. Since this project uses the XNA framework, and by extension, Direct3D, we did not have to implement this pipeline ourselves. Direct3D uses a so called shader language called HLSL (High Level Shader Language). This shader language deals with two steps of the graphics pipeline and consists of two parts: the vertex shader and the pixel shader. There is also a third type of shader, called the geometry shader, which will not be covered here. The vertex shader, as the name implies, deals with transforming and shading the vertices. This involves transformation of the vertices' positions from world space to screen space, calculation of normals, and the depth value of the vertices, among other things. This information is then passed on to the pixel shader, which is executed for each pixel on the screen that contains any geometry. The pixel shader outputs color data for each pixel on the screen, and can generate this color data in a multiple of ways, including looking up data from

textures that have been input, calculating complex shading operations based on the information passed on from the vertex shader, and even generating special effects that can be applied to the whole screen (post-processing).

3.2 Models and Texture Maps

This section briefly covers the basics of 3D models and textures. It also discusses the results of how models and textures were handled for our game.

Models. There exists many different 3D file formats [McHenry 2008]. The shape of models used in applications where quick rendering is important are commonly stored as collections of vertices, vertex indices and vertex normals, which together are called meshes. The way in which the vertex indices are structured defines the polygons, i.e., the faces, of the mesh. The detail of polygonal meshes depends on how many vertices the mesh contains. In addition to one or more meshes, a 3D model may include other data such as animations, material properties, textures, and UV coordinates that say how texture maps should be applied to the model. What is included in a 3D model file depends on the file format.

Texture Maps. Within 3D applications, a texture map is an image file that is usually mapped to a model according to the model's UV coordinates [Heckbert 1986]. Texture maps can be used for many different purposes, a common use is acting as color maps, providing color to a model. A texture is simply represented as an image, and can be painted in most painting software, and saved as any common image file format (e.g., PNG, JPEG, or BMP). Other uses for textures include bump mapping (see Section 3.5 Bump Mapping) to add surface detail, material mapping to change some material property, and alpha mapping to change the transparency of some part of the model.

Results and Discussion. No one in our group had any significant previous experience with 3D modeling or texturing, and since many tasks had to be worked on for the game, we decided to not have a dedicated modeler. This, combined with time constraints, forced us to use several free models and textures for items we could not create ourselves in a timely manner. Therefore, the style of the game is not very coherent, as finding suitable free models, that may or may not even exist, is difficult and time consuming. One of the models that was created by us is shown in Figure 3.2.1.

Figure 3.2.1: A rocket model, created by us, used in the game.

3.3 Shading

An important aspect of making a scene look graphically convincing is how the environment is shaded. This section goes through design decisions more specific to this game, where the light source is not only in the middle of a planetary system, but is also very large in relation to the other objects, as well as discusses the results.

3.3.1 Lights

There are several different ways to represent light sources, common ones being point lights, directional lights, spotlights, and area lights. Both the spotlight, and the directional light sources have defined directions [Akenine-Möller 2008, pp. 100, 220], so when the light source is in the middle of the scene, as in our game, they are not very suitable; an omnidirectional light source would be preferred. This leaves the point and area light sources as possible candidates.

A point light source is a light source where all the light comes from an infinitely small point. This approximates lighting from a light source that is either very small, or far away. However, when the light source covers a larger area, a point light source might be too inaccurate, as it will not illuminate the surrounding objects as well as a large light actually would (see Figure 3.3.1). An area light source would solve this issue by simulating the larger area, commonly using multiple point light sources, which results in more proper lighting, but at the cost of computational power. One way to circumvent this issue is to, for every fragment of the object on which to apply lighting, choose the best point of the light source, and use only this point, as a point light, for the light calculations.

In the special case of a spherical light source shading a spherical object, we found that it is possible to very efficiently approximate this point. This can be done by simply multiplying the normalized normal vector of the fragment to be lit, by the light radius of the light source, and finally adding this vector to the position of the light (see Figure 3.3.2).

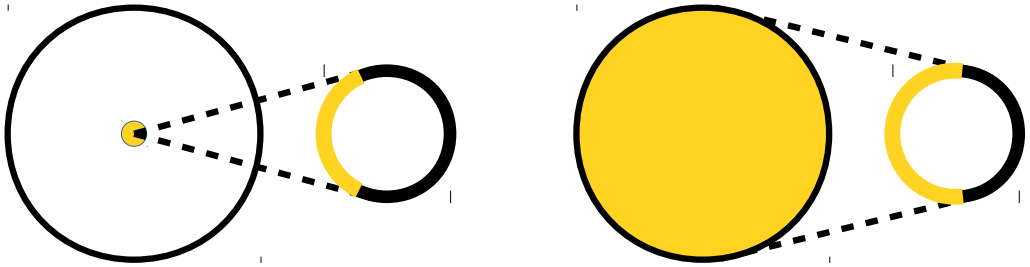

Figure 3.3.1: A large, spherical light illuminating a smaller sphere. *Left:* A light represented as a point light source. *Right:* A more realistic light, giving light to a larger area of the sphere. The dashed lines represent the outermost rays of light that hit the sphere.

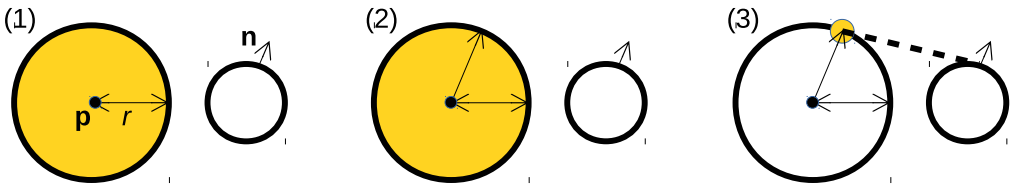

Figure 3.3.2: A sequence explaining a way for a spherical light to illuminate another sphere. The normalized normal vector **n** of the fragment to be shaded is multiplied by the radius r of the light source, which gives a vector from the center to the edge of the light source (2). This vector is then added to the position **p** of the light source in order to receive the point on the light which is most relevant to use as a point light for the shading (3).

3.3.2 Shadows

An object that is illuminated by light also casts a shadow, which in computer graphics is commonly achieved by the techniques Shadow Maps or Shadow Volumes. The Shadow Map algorithm, as introduced by Williams [Williams 1978], renders the depth buffer from the light source's viewpoint and stores it in a shadow map. When rendering the scene, it checks, for each fragment, whether the distance between the fragment and the light source is greater than the value stored in the shadow map or not. If the value is greater, the fragment is in shadow, otherwise it is in light. The first problem that comes with shadow maps in our case is that, when it comes to an omnidirectional light source, the shadow map is instead rendered to a cube map, resulting in rendering the shadow map six times rather than one. The second problem is that, the size of the shadow map directly influences the quality of the shadows, and the further away the shadow is from the light source in relation to the camera, the worse the quality of the shadow will be; in order to have high quality shadow maps in a game with great distances, such as our game, the shadow map textures would need to be very large for satisfying results.

Shadow Volumes was introduced to computer graphics by Heidmann [Heidmann 1991], and is a technique based on representing shadows with geometry. Assuming that we draw a line from the position of the light source through each of the corners

of a triangle, we would form a pyramid that extends to infinity. The pyramid beyond the triangle's corners is called a shadow volume, and any geometry that intersects this shadow volume is in shadow. Once shadow volumes have been generated, the scene is rendered as normal. For each pixel, a ray is cast from the light's position toward that pixel's position, and the amount of intersections with shadow volumes is counted. For each shadow volume entered, the counter is increased by one, and for each exited, the counter is decreased by one. When the pixel is reached, if the counter is larger than zero, the pixel is in shadow. Heidmann [Heidmann 1991] also proposes optimizations to this technique using the stencil buffer to do the counting. Shadow volumes do not have the resolution problems that shadow maps have, but can be more difficult to implement. In our case, they are also limited in that the can not comfortably handle area lights.

Due to the limitations of the two presented algorithms, we decided to simply check whether an object is in shadow or not, using ray-sphere intersection. The algorithm shoots a ray from the object to be shaded, towards the center of the light source. If this ray hits a planet in between, the object is considered to be in shadow, as illustrated in Figure 3.3.3.

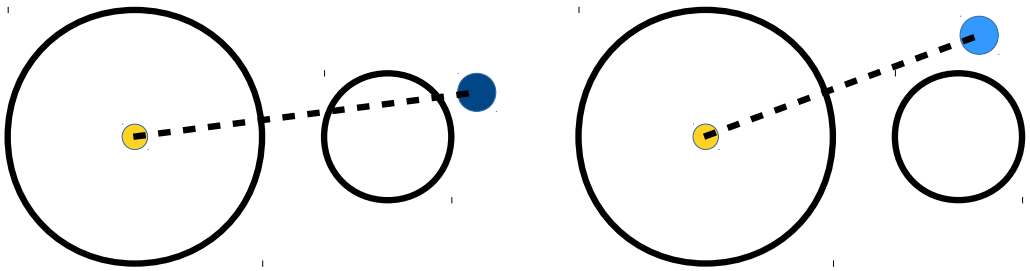

Figure 3.3.3: An illustration of how our shadows are determined. *Left:* An object shoots a ray towards the center of the light source, but it intersects a shadow caster in between; the object is determined to be in shadow. *Right:* The ray shot towards the light does not intersect anything, so the object is determined to be lit.

3.3.3 Results and Discussion

We currently use the lighting algorithm only for rendering the planets, since these are the only objects that are too large for traditional light calculations using a point light, successfully lighting a larger area of the planet, as can be seen in Figure 3.3.4.

The shadow algorithm gives a very crude approximation, and, in general, objects will commonly be rendered as if in shadow when they should not be. However, since the shadowing objects are considerably large in comparison to the objects being shadowed, the low accuracy of the algorithm is not very notable in practice; the most notable inaccuracy is the transitioning to and from being shadowed since the shadow is considered per object, not per fragment. The result of the game using this implementation is shown in Figure 3.3.5.

We are very content with the lighting of the objects, and we have no intention of

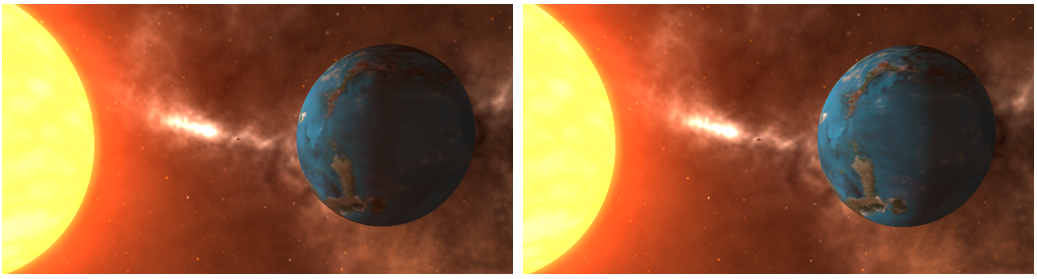

Figure 3.3.4: *Left:* Shading on a planet, with the light represented as a point light. *Right:* Shading on a planet, with the light represented as an area light, using our planet shading algorithm. As can be seen, the latter method illuminates a larger area of the planet's surface.

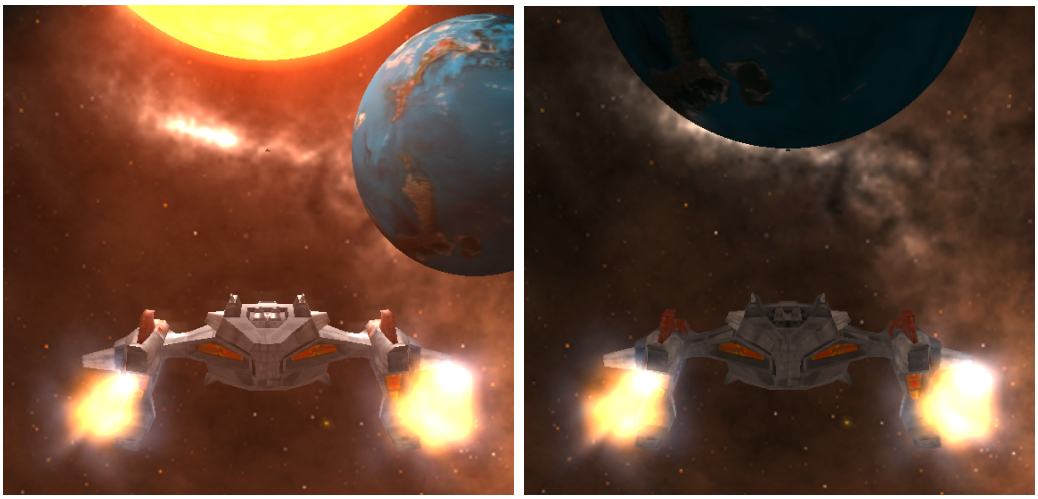

Figure 3.3.5: *Left:* The player ship illuminated by a light. *Right:* The light has been obstructed by a planet, casting a shadow upon the ship.

any changes there. The shadows, however, could be calculated more accurately. Our plan was to, after finding out the most relevant shadowing planet, check for each fragment on the shadowed object, if that specific fragment is in shadow or not. Due to some complications and time constraints, this was not prioritized, and has not been implemented. Furthermore, this implementation would not generate soft shadows, which would be ideal. We did have some ideas about how to implement this, but we decided to not explore this too deeply before we had, at least, per-fragment shadows.

3.4 Post-Processing

A post-process effect is an effect that is applied on already rendered images. There are a wide variety of post-processing effects could be implemented in order to highlight effects or amplify the player experience. For example, motion blur can be used to enhance the feeling of speed [Rosado 2007] and depth of field can be used to create

a greater feeling of depth [Demers 2004]. This section covers several post-processing effects used in the game, and discusses their results.

3.4.1 Motion Blur

When an object moves quickly across the field of view, it appears blurry. This effect is called motion blur, and is often simulated in computer graphics to gain more realism, feeling of speed, as well as smoother transitions between rendered frames.

Without post-processing, this can be achieved by, for example, solving temporal visibility and shading from an equation describing pixel intensity [Sung 2002]. Post-processing solutions to apply the effect have been done for example by time convolution [Potmesil 1983], convolution along a vector (Line Integral Convolution) [Cabral 1993], or by blending previously rendered images together with the current one [Chen 1993].

Time Convolution. Time convolution simulates the effect of an object moving during the time of exposure for one frame through a camera lens, as in an object moving during the time of which the shutter of a camera is open [Potmesil 1983]. For time convolution algorithms, sample points of moving objects are processed separately. The object path, the exposure time, and the exposure length of the camera are factors taken into account to calculate the motion blur to be produced from the moving objects.

Line Integral Convolution. The line integral convolution algorithms can, with a velocity buffer, generate motion blur [Cabral 1993]. This buffer can be generated by the shader, through passing a velocity vector for every vertex to the pixel shader, which then writes it out to the buffer [Akenine-Möller 2008, pp. 490-496]. By filtering one-dimensionally along the vectors, motion is apparent in the resulting image when applying the algorithm.

Image Interpolation. By storing images rendered from previous frames in an accumulation buffer, and interpolating them together with the image of the current frame, a resulting image with a motion blur effect can be achieved [Chen 1993]. Considering only images are required to use an image interpolation algorithm, the complexity of the scene does not necessarily affect the computation needed to apply the algorithm.

Results and Discussion. The algorithm that was chosen for our game makes use of Line Integral Convolution. This means that it makes use of a velocity buffer, which needs to be made for this purpose. In order to calculate the velocity of an object in a pixel, the shader compares the current position of that object with the position it had at the previous frame. To make this calculation possible for the shader, we pass in the inverse of the current model-view-projection matrix and the model-view-projection matrix it had at the previous frame into the shader as uniforms. This information, together with information from the depth buffer, is enough to calculate and compare these positions, and thereby decide the velocity between these two frames. Once the velocity map is created, the motion blur can

be done by taking samples along the velocity vector for the pixel, and interpolating the color of these samples with the color of the current pixel [Rosado 2007].

This algorithm was chosen because it is relatively easy to implement, and our scene is not very complex, so an accumulation buffer approach (image interpolation) would not be as fitting as for complex scenes. Since our shaders already output to multiple textures, and since the depth buffer is also needed for other techniques that we have implemented, not much further implementation is needed for a line integral convolution algorithm. An approach to image interpolation was briefly looked into, but line integral convolution was decided to be more fitting in this case.

Sadly, the algorithm was only partially implemented due to time constraints, and as such, we can not show any pictures of the results.

3.4.2 Bloom

Bloom, also commonly referred to as glow, is a post-processing effect that simulates the glowing nature of bright light [James 2004]. In natural circumstances, it is caused by the scattering of light in the atmosphere or the eye; however, it is worth to note that this effect does not occur when bright light is displayed on a computer screen [Nakamae 1990]. There are several techniques that can be used to provide a bloom effect, two such techniques are presented by James and O'Rorke [James 2004]; one suitable for small objects, using blurred billboards, which is not suitable for our game due to the many large objects and the fast movement through the world, and one post-processing algorithm suitable for dynamic objects.

The post-processing algorithm has three steps: isolation of glowing areas, blurring, and combining. When isolating the glowing areas, a glow source mask is included in the alpha channel of the rendered object's texture, which allows copying the already rendered screen into a second render target, that will now contain the glow areas and nothing else. The next step is the blurring step. This is done using a gaussian blur approximation, and can be done efficiently using a two-pass blur filter at a lower resolution [Akenine-Möller 2008, pp. 482-486]. The final step combines the blurred image of the glow areas with the originally rendered image, using additive alpha blending, to produce the glow effect.

Our implementation is similar to the one described above, but is different in the isolation step. Instead of using a glow source mask to identify what areas to blur, we use a shader-pass that discards all color with less than a certain brightness. This allows us to also apply bloom to things that are rendered with transparency, such as the fire from the engine of the space ship. It does, however, require an additional render pass, which is more computationally expensive.

Results and Discussion. The bloom effect increased the visual quality of the game by a large amount. It helped make other effects, such as the fire from the engine and the laser look more believable, as can be seen in Figure 3.4.1. The effect is versatile and can be adjusted to convey different moods and settings, and it was also not particularly difficult to implement. Bloom is not very computationally

expensive due to downsampling, but it does require an additional render pass, which is noteworthy. We did, however, have some problems with the bloom effect applying to objects we did not want to glow, such as the background. This not only caused the background to become unintentionally bright in some areas, but also caused it to sometimes bleed over onto other objects, such as the planets.

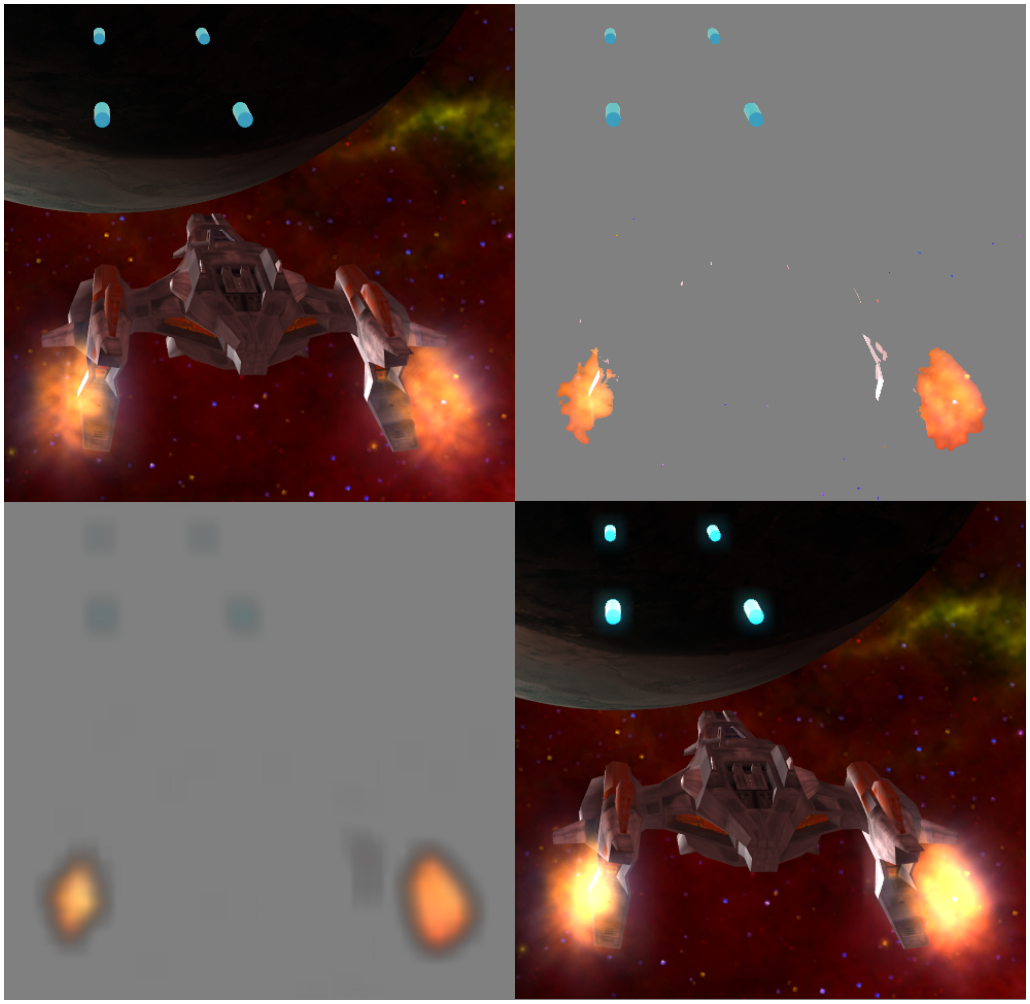

Figure 3.4.1: *Top Left:* An image with no bloom effect. *Top Right:* The isolated bright areas of the original image. *Bottom Left:* The previous image, blurred. *Bottom Right:* The combination of the blurred and original images, showing the result of the bloom.

3.4.3 Volumetric Light Scattering

Volumetric light scattering, also known as god rays, is an effect used to imitate how rays of light can sometimes be seen in dense air particles. There are several ways of achieving this, common methods using ray marching to solve the light scattering integral described by Nishita et al. [Nishita 1987]. One way to implement volumetric light scattering without the heavily computational ray marching, is proposed by

Billeter et al. [Billeter 2010]. This method, however, requires a shadow map, which is something we chose to avoid, as discussed in Section 3.3.2 Shadows.

Since we do not have a shadow map, we chose to implement a technique using ray marching. First, all occluding objects, as seen from view space, are rendered into a pre-pass frame buffer. This can then be used, when rendering the scene, to, for each pixel, cast a ray from the pixel to the view space light position, and check at a number of sample points along the ray, whether it is occluded or not. The number of unoccluded samples determines how much lighting will be additively blended into the pixel. This method is explained in more detail by Mitchell [Mitchell 2007]. While this algorithm does not require a shadow map, it is less accurate than if it were, since it does not take into account if an occluding object is in front, or behind, of the light source, resulting in rays of light occasionally appearing in front of objects in the foreground [Mitchell 2007].

Results and discussion. Our implementation of volumetric light scattering performs well with computers with mid range or better graphics hardware, but not as well with low-end computers, or computers without a dedicated graphics card. The effect in action can be seen in Figure 3.4.2. We are content with the results we achieved, but due to performance issues, we decided to implement the option to toggle volumetric light scattering on or off.

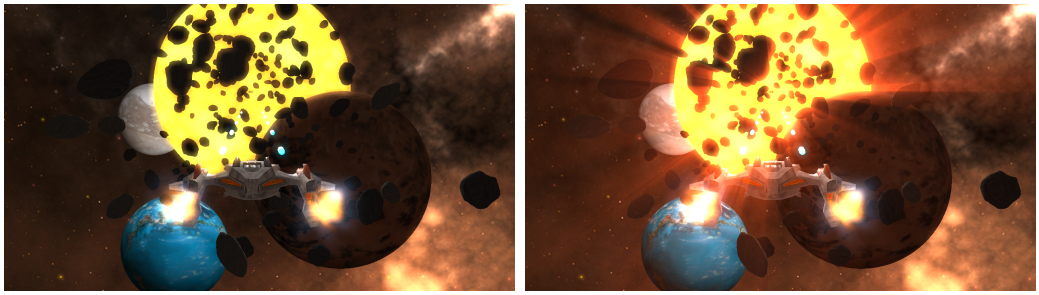

Figure 3.4.2: *Left:* A screenshot from the game without the volumetric light scattering effect. *Right:* A screenshot showing the same scene, but with volumetric light scattering activated

3.4.4 Depth of Field

Depth of field is a graphical effect that is applied to simulate the functionality of a real lens, such as a camera lens or the lenses in our eyes, in regards of focusing on a set distance. There are several different techniques and approaches used to achieve this within computer graphics, for example by using ray-tracing [Cook 1984] or an accumulation buffer [Haeberli 1990], as well as algorithms that divide the scene into layers [Scofield 1992]. When it comes to applying depth of field by post-processing, there are algorithms making use of mapped depth buffers [Potmesil 1982].

The depth-of-field effect occurring in reality is caused by the physical properties of a lens. In this context, we will treat these lenses as convex, and because of this they

redirect the light passing through them inwards towards a so called image plane; for example in the case of an eye this would be the retina. Let us, to exemplify, assume a point of an object on the other side of a lens projects light hitting all over the whole lens. In order for said point of the object to be in perfect focus, the case has to be that the lens converges the light to hit one single point of the image plane. This is only the case for a specific distance from the lens to the object, depending on the properties of the lens and the distance between the lens and the image plane. These properties of the lens also affect how far away something has to be in a scene in order to be in focus. The plane in focus is the plane which is in perfect focus for the lens; any light coming from further away than the plane in focus will not be converged sharply enough by the lens to hit the image plane at a single point. Instead, this light will hit the image plane spread evenly over a circular area on the image plane, with the radius of this circle depending on how much out of focus the light source is. This circle is called the circle of confusion. Likewise, if the light source is closer to the lens than the plane in focus, the lens will not make all of the light hit a single point on the image plane, but will instead spread it over a circle of confusion. The circle of confusion phenomenon is visualized in Figure 3.4.3.

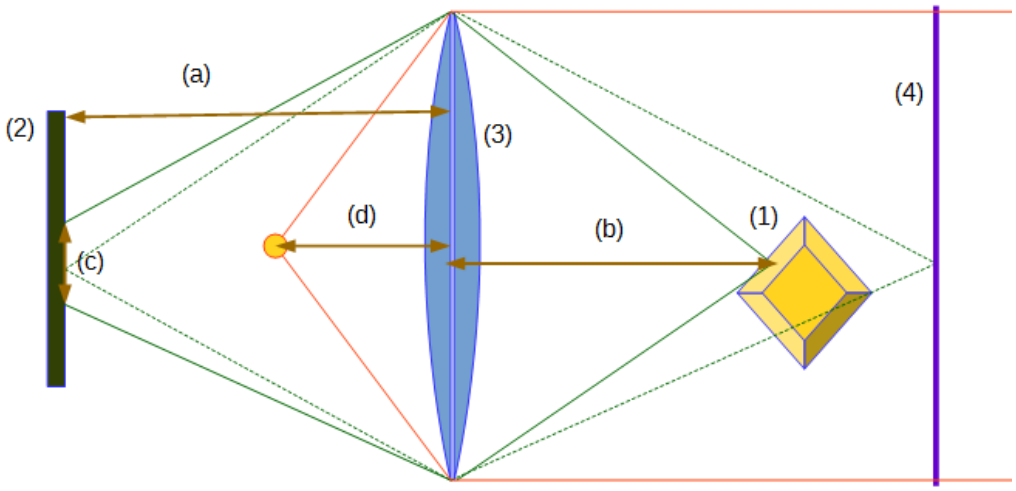

Figure 3.4.3: An object (1) being projected onto an image plane (2) through a lens (3). The image plane has a certain distance (a) from the lens. Because the distance (b) from the lens to the object is shorter than from the lens to the focus plane (4), the light emitted from the object through the lens will not get converged onto a single point onto the image plane. Instead, it will be projected over a circle of confusion with a diameter (c) proportional to the distance between the object and the focus plane. The focal distance (d) depends on the properties of the lens.

The focal point is defined as the point behind the lens, for which parallel rays of light hitting the lens orthogonally would converge into. The distance between the lens and this focal point is called the focal distance. Assuming the properties of a lens do not change, the focal distance is constant for that lens. Let D be the distance from the lens to the light source, V be the distance from the lens to the image plane, and F be the focal distance; then there is a relation such that $\frac{1}{D} + \frac{1}{V} = \frac{1}{F}$. This is the thin lens equation.

In computer graphics, shortcuts are often made when implementing the depth of field effect in order to scale down the required calculations and balance physical accuracy with performance. For example, a lot of algorithms assume a pinhole camera lens with zero diameter, causing a light source to have exactly one path for its light to travel through the lens. When doing this, physically correct calculations of depth of field can not be done, so approximations are used instead [Demers 2004]. This section will now explain a few techniques used to simulate depth of field.

Ray-Tracing Algorithms. Depth of field can be implemented by tracing rays that are cast from the image plane through the lens. While these algorithms are relatively realistic, and are relatively accurate in terms of physics, they are also not processing the images at a speed suitable for real-time graphical computations [Cook 1984].

Accumulation Buffer. Another technique is based on using an accumulation buffer. This buffer is created by rendering the scene from slightly different positions through the lens and adding every resulting image to that buffer [Haeberli 1990]. The images are then blended together, resulting in an image with a depth of field effect. Naturally, the more images rendered and added to the accumulation buffer, the more accurate the resulting image will be, but the computational cost increases accordingly.

Layered Depth of Field. With Layered Depth of Field, the scene is required to be divided into layers [Scofield 1992]. These layers can then be blurred separately, based on how their depths relate to each other. Though these algorithms aren't taking the physical properties of the lens into account, the resulting images appear to have the effect of depth of field. Dividing the scene into layers can be quite tricky, as there might not always be obvious depth ranges to divide it into.

Forward-Mapped Z-Buffer Depth of Field. With this post-processing technique, a source image, containing pixel colors, is used [Potmesil 1982]. A Z-buffer is also needed to determine the depth for each pixel. Then for every pixel, a circle, with a size based on the calculated circle of confusion, is rendered to a target image. This circle is blended together with the circles drawn based from all the other pixels in the source image, so that the result is an image blurred to represent depth of field. This requires a lot of rendering since one circle is rendered per pixel.

Reverse-Mapped Z-Buffer Depth of Field. This technique is similar to the previous one; the difference between the reverse-mapped technique and the forward-mapped, is that this one does not create a blurred circle based on each source pixel [Demers 2004]. Instead, for each pixel it samples the surrounding ones in order to calculate how much they affect the current pixel by blurring into it. This allows the GPU to bypass having to render out a circle for every pixel.

Results and Discussion. The depth of field algorithm that we used is a variant of the Reverse-mapped Z-Buffer Depth-of-Field technique. It is divided into stages that together practically layer the scene in a layer behind the focus plane, and a layer in front of the focus plane [Hammon 2007]. First, the thin lens equation is applied in a pixel shader to evaluate which pixels are too far away to be in focus

and which ones are too close to be in focus, as well as how much out of focus that they would be. From this, a so called confusion map is created. The confusion map is then blurred out by applying a gaussian blur. The pixels in the original image which are too far away to be in focus according to the non-blurred confusion map, are blurred out. In a separate stage, pixels that are too close to the camera according to the blurred out confusion map are blurred out with a specific algorithm in order to achieve a blur that looks relatively realistic close up. The results from these two blurs are then merged into one resulting image.

A reverse-mapped Z-Buffer Depth of Field algorithm was chosen for our game because it yields good performance relative to the other algorithms without sacrificing too much quality [Demers 2004]. An interesting choice of algorithm could also have been one that implements depth of field by straight off layering the scene a bit more. Due to the vast differences of distance between objects in our scenes, the layering itself could most likely be implemented with relatively simple effort. The necessity of spending extra computations to accurately blur objects close to the camera can also be discussed, considering objects rarely get between the player and the camera. If they do, they tend to not be there for a longer extent of time.

3.5 Bump Mapping

This section covers the concept of bump mapping, several bump mapping techniques, and evaluation of our results from implementing normal mapping.

The basic concept of bump mapping is to use data stored in textures to fake surface detail, like wrinkles and bumps, by modifying the normal used during per pixel lighting calculations [Blinn 1978]. Rather than interpolating the normal from the vertices, as is normally done, the normal is modified, or replaced, by the bump map, as shown in Figure 3.5.1. Since only the normals are changed, the bumps will not occlude what is behind them, nor will they cast shadows or modify the silhouette of the object. Rendering a model with bump mapping is less computationally expensive compared to rendering geometry with similar surface detail, as the model would necessarily have to be significantly complicated.

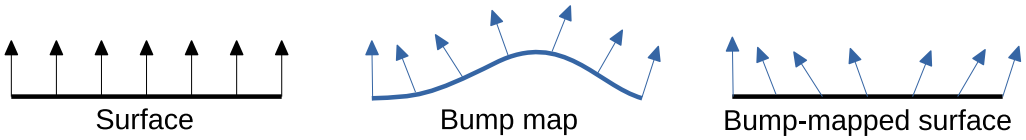

Figure 3.5.1: This figure shows a bump map being applied to a surface, by replacing the surface's normals with normals from the bump map.

Bump Mapping. Bump mapping was introduced by Blinn [Blinn 1978]. He showed two methods for bump mapping, vector offset bump mapping, and bump mapping using a heightmap. Vector offset bump mapping uses a texture that stores two values, which represent offsets in two directions. These values modify the normal

used during lighting calculations by offsetting it accordingly. Bump mapping using a heightmap, which we will refer to as standard bump mapping, uses a monochrome texture where the gray values represents the height. By looking at the neighboring texture values, the slopes, and thus the offsets, can be derived and used to modify the normal. See Figure 3.5.2. for an example of standard bump mapping.

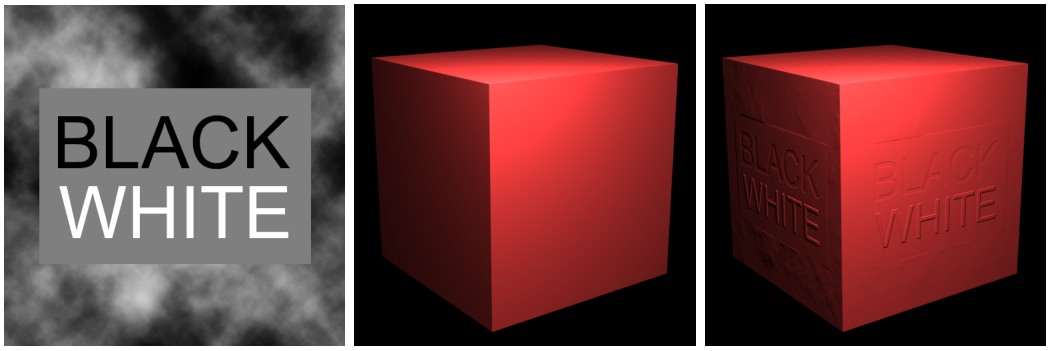

Figure 3.5.2: *Left:* A bump map. *Center:* A red cube without any bump mapping. *Right:* The red cube with the bump map applied.

Normal Mapping. Normal mapping [Cohen 1998] is a commonly used bump mapping technique where, unlike the previously mentioned bump mapping techniques, the normals are stored directly in the bump map, henceforth called normal map. A normal map can be seen as a precomputed bump map, and should, depending on the implementation, produce the same result. Due to this connection, we can convert bump maps into normal maps (see Figure 3.5.3) where the (r, g, b) values of the normal map represent the (x, y, z) values of the normal. For an 8-bit normal map, the components of the normals are mapped from [-1, 1] to [0, 255] onto the colors in the image, where (128, 128, 255) represents an unchanged normal (0, 0, 1). Storing the normals directly takes extra memory compared to a heightmap, but has the benefit of simplifying the rendering calculations.

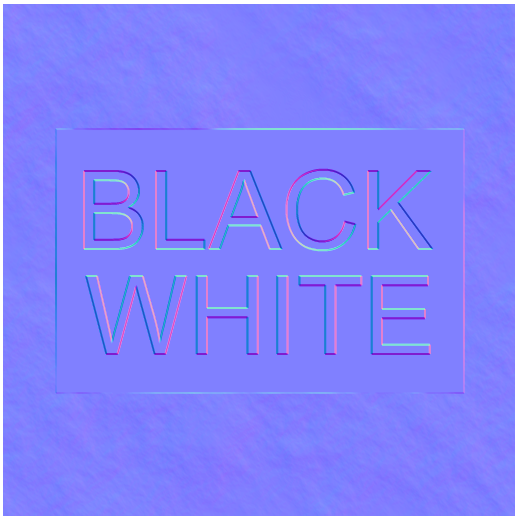

Figure 3.5.3: A normal map conversion of the bump map shown in Figure 3.5.2.

Parallax Mapping. Parallax mapping [Kaneko 2001] is a bump mapping technique which attempts to approximate parallax, that is, when moving, objects further away appear to move more slowly than objects closer. As parallax can be observed in non-flat surfaces, simulating parallax further enhances the surface detail by making bumps observable. In contrast to normal mapping, rather than changing the surface normals, parallax mapping simulates bumps by offsetting the sampling of textures by taking the viewing angle and height in an associated heightmap into account. Depending on the angle and the height, the texture is sampled at different locations. Parallax mapping, as introduced by Kaneko et al. [Kaneko 2001], has problems at shallow viewing angles. The texture sampling can be offset so much that the result does not at all resemble the original texture [Welsh 2004], so Welsh proposed limiting the offset, which reduces the parallax effect but produces less artifacts.

Relief Mapping. Relief mapping [Policarpo 2005] is an advanced bump mapping technique that samples a ray, shot from the camera position to the surface, at regular intervals to see if the ray intersects with a heightmap. Relief mapping supports self-occlusion, self-shadowing, and variations of relief mapping can produce correct silhouettes [Oliviera 2005].

Both parallax mapping and relief mapping are more computationally expensive than normal mapping, and relief mapping in particular is quite expensive due to the ray tracing. We chose to implement normal mapping due to the simplicity, speed, and availability of helpful resources.

3.5.1 Results and Discussion

Generally, objects in the game are either too far away or move too quickly for the effect to be noticeable. For this reason, the number of game objects for which normal mapping would significantly increase the visual quality is very limited. Normal mapping is therefore only implemented for the spaceships, as at least the player's ship is visible at all times, and even then it is not particularly noticeable (see Figure 3.5.4). Normal mapping could have been implemented for the planets, however, the planets use very high resolution textures, and adding another set of textures for the planets would increase the memory usage considerably.

Implementing some other, more advanced, bump mapping technique, like relief mapping, would likely have yielded greater effects, at the expense of performance. However, as game objects would still be far away, or be moving quickly, considering the performance loss compared to what is gained graphically, it would likely not have been worth it.

Figure 3.5.4: *Left:* A spaceship without normal mapping. *Right:* The spaceship with normal mapping. Note the added detail by the exhaust.

3.6 Particle System

Certain effects, such as smoke, fire and explosions, are in general difficult to create due to their irregular, and perceptually random, shapes and behaviour. A common way to create convincing effects in real time is to generate and transform them using a particle system, as introduced by Reeves [Reeves 1983]. This section explains the algorithm used in our game, as well as the results, and discusses our decision.

3.6.1 Algorithm

A particle system can be designed to run on the CPU, the GPU, or in a combination of both. For this project, the particle system has been decided to mostly run on the GPU due to significant difference in calculation speed between the two units.

The basic concept of a particle system is that, in order to generate an effect, it creates a number of particles, small images, with different properties such as velocity, gravitational force, color, size, some of which are constant, and some which change depending on the desired results. The idea is to make all of these individual particles behave slightly differently from each other, and together make up an effect that behaves coherently, but varying in time.

In this particular project, it was decided that Microsoft's 3D particle system sample for XNA would be used. Every particle effect that is to be implemented into the particle system requires a sub-system of its own, defining its properties. Some of these attributes are global for the whole effect, while some are randomized per particle, once the particle effect is instantiated. Examples of global properties are the texture to be used, the maximum number of particles than can be active, and how to blend this particle effect into the scene. Some particle-specific attributes include velocity, color, rotation, and size, and while chosen randomly, their possible ranges have to be specified as part of the sub-system.

When rendering the particles, these particle-specific attributes are passed along with the vertices to the shader, which then computes the position, size, and rotation of

the particles, using a uniform variable that holds the time.

3.6.2 Results and Discussion

The effects that are being generated by the particle system are explosions with smoke, fire from the jet engines of the player ship, and laser collisions (see Figure 3.6.1).

Figure 3.6.1: A screenshot showcasing the particle effects we use for explosions, laser collision, and fire from the ship's engines.

Due to the large amount particle effects generated at times, and due to the player moving towards and through these effects, there are occasions when hundreds of overlapping images that cover most of the screen have to be rendered in a single frame. This results in notable framerate drops, something that has been slightly improved, but not completely removed, by reducing the lifetime and amount of particles generated and instead making the individual particles more prominent.

It seems that the large amount of texture lookups that a particle system can result in can be difficult for most computers without a dedicated graphics card to render, so it might have been a good idea to look more into other available particle systems, and see if they would have performed better.

3.7 Terrain Generation

This section discusses a feature that was not completely implemented in the game due to time constraints, but that we still find interesting enough to discuss in this thesis. Initially, we wanted to allow players to occasionally descend to a planet's surface and fly over varying terrain; this would give players a change of scenery. In order to accomplish this, we researched a few different techniques to procedurally generate terrain.

There are several techniques that can be used to generate terrain. The most common approaches are heightfield based [Fishman 1980], often created using fractal subdivision techniques [Fournier 1982, Gardner 1984] as well as Poisson Faulting [Mandelbrot 1982]. Noise [Perlin 1985] can be used to break the repeating nature of these techniques [Miller 1986, Musgrave 1989]. However, heightfields hold a limitation in that they cannot represent arbitrary three-dimensional curvature, e.g. caves or overhangs; to overcome this limitation, one must step into a third dimension. This can be done by using an algorithm called Marching Cubes [Lorensen 1987]. Marching Cubes takes volumetric data, which in turn can be generated from noise, and polygonizes the surface [Geiss 2007].

3.7.1 Heightfields

A heightfield, also commonly referred to as a heightmap, is a mapping from a two-dimensional coordinate to a height value. This value can then be used to create terrain, where the height value corresponds to how high up the surface of the terrain should be from some pre-determined base value. Creation of this heightfield can be accomplished in several ways, and, considering that many features of terrain are fractal in nature [Mandelbrot 1982], it makes sense to take a fractal approach to procedurally generating them. Fractals can be modeled by, among other things, subdivision techniques and combinations of Perlin Noise [Perlin 1985].

One of the most common subdivision techniques for creating fractal terrain is the Diamond-square algorithm. The algorithm is divided into two steps: the square step and the diamond step. The square step takes a square of points with a scalar value at each corner, and places a new scalar value, based on the corner values, in the middle of the square. The diamond step uses the middle point from the square step to form diamonds, and places a new scalar value, based on the corner values, in the middle of the diamond. The algorithm starts with a square grid with a seeded value at each corner. Then it recursively applies the square step and the diamond step to assign values to each point. Once a sufficiently large amount of iterations have been made, or the grid has been filled, the algorithm stops. The process can be seen in Figure 3.7.1.

Miller [Miller 1986] argues that the diamond-square algorithm is flawed, criticising the visible creasing problem and the artifacts it produces. Musgrave et al. [Musgrave 1989] proposes a technique based on summing band-limited noise, which removes

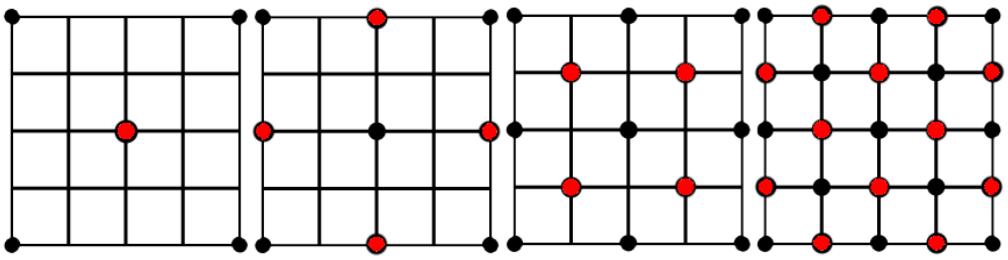

Figure 3.7.1: An image showing the diamond-square algorithm. The red points are added in each step, using the already present black points. Note how in the diamond step shown in the second and last image, the algorithm wraps around the edges to find the black points needed.

these issues, and this will be presented in the next section.

3.7.2 Perlin Noise

Perlin noise is a gradient-based noise function that maps a relation between $R^n \rightarrow R$, invented by Perlin [Perlin 1985], that can be used to generate height fields. It has several useful characteristics, one of the most important being that it is pseudorandom; the output is only dependent on the input, and will not vary between executions. Another important feature is that several samples of noise can be summed together to create fractal noise.

The parameters of fractal noise are usually called frequency and amplitude. The frequency of the function is a multiplier of the input, where higher frequencies means that it varies faster, while the amplitude is a multiplier of the output, where a higher amplitude means a larger output. A single sample of the noise function with some frequency and amplitude is called an octave. High amplitude combined with low frequency gives the shape of large hills and mountains, while low amplitude combined with high frequency gives the details of rocky texture; summing up several octaves of noise, with varying frequencies and amplitudes, can yield convincing terrain, and is highly customisable for different scenarios. The output of these octaves can be rendered in the same way one would render a heightfield, but can also be used for a wide range of techniques. For more information about Perlin Noise and its uses, see [Perlin 1985, Perlin 1989].

Heightfields, regardless of method used to generate them, do however have an innate problem. Since every two-dimensional coordinate corresponds to exactly one height value, there is no way to represent concavities such as caves and overhangs. This is not a trivial problem to overcome, and we will now present a solution called Marching Cubes.

3.7.3 Marching Cubes

The Marching Cubes algorithm is an algorithm that creates a polygonal mesh from a scalar field. It was originally developed by Lorensen and Cline [Lorensen 1987] to visualize data from medical scans such as CT or MRI scans, but has later been used to create polygon representations for isosurfaces of any origin. In our case, we are interested in using it to create the surface of terrain.

When generating terrain, the scalar field can be seen as the density values of the terrain, where a negative value for a point represents non-terrain volume, such as air or water, and a positive value for a point represents solid material, such as rock or soil. We want to create a polygonal representation along the border between positive and negative points, which is what the Marching Cubes algorithm does. The algorithm uses the scalar field as input and traverses it as a cubic grid. At each cube, the vertices are looked at, and it is identified what edges, if any, have one vertex in solid material and the other in empty space. Triangles are then generated, with one triangle vertex on each of the identified edges. This will, however, only be a rough approximation, since the actual surface value might be closer to one of the vertices on the edge than the other. To solve this, we need to interpolate the triangle vertex positions using the scalar values corresponding to each vertex.

Lastly, the triangles need a unit normal for shading. The normal for a triangle can be obtained by using the cross product of two of the edges, but this will only produce hard edges between triangles. Lorensen & Cline [Lorensen 1987] showed that smooth edges can be obtained by finding the derivative of the scalar field at the point; this can be estimated using numerical differentiation. Lorensen and Cline [Lorensen 1987] stated that "since there are only two states for each vertex and eight vertices per cube, there are $2^8 = 256$ ways that the surface can intersect the cube", and also showed that these 256 cases can be reduced to 14 patterns. The vertices for the cube can be arranged into a byte and used to access an edge table to get the edge intersections. This vastly improves the running-time, since accessing a table is always faster than computing the information again.

We have now described an algorithm for generating a surface based on volumetric data. However, we must obtain this volumetric data somehow. Fortunately, this can be done using the same perlin noise techniques described in the previous section [Geiss 2007]. However, since Marching Cubes requires a three-dimensional scalar field, we need to sample noise in three dimensions. This technique has a few major drawbacks. It produces visual artifacts, as noted by Lengyel [Lengyel 2010]. It's also rather computationally expensive compared to the heightfield techniques since it operates in three dimensions instead of two.

3.7.4 Results and Discussion

For our game, we wanted to have the possibilities of overhangs, arches and cavities. For example, we thought it would be beneficial for the game's visual fidelity to be

able to fly through canyons with arches overhead. Because of this, we decided to put further research into the Marching Cubes approach.

This was fully implemented to work on the CPU, and we were able to generate relatively complex geometry using this. However, since the game moves at rather high speeds, the world size of the terrain generated needs to be rather big, and the Marching Cubes algorithm does not scale as well as a heightfield. Our implementation is able to generate geometry consisting of 120k triangles, for a scalar field with a resolution of 2.8M voxels, in approximately nine seconds using a 1.8GHz Intel i3 processor. This can be made to run faster; for example, we could utilize the GPU. Since the Marching Cubes algorithm operates on one cube at a time, and each step is independent of all other cubes, it is highly parallelizable [Geiss 2007]. Our results are shown in Figure 3.7.2.

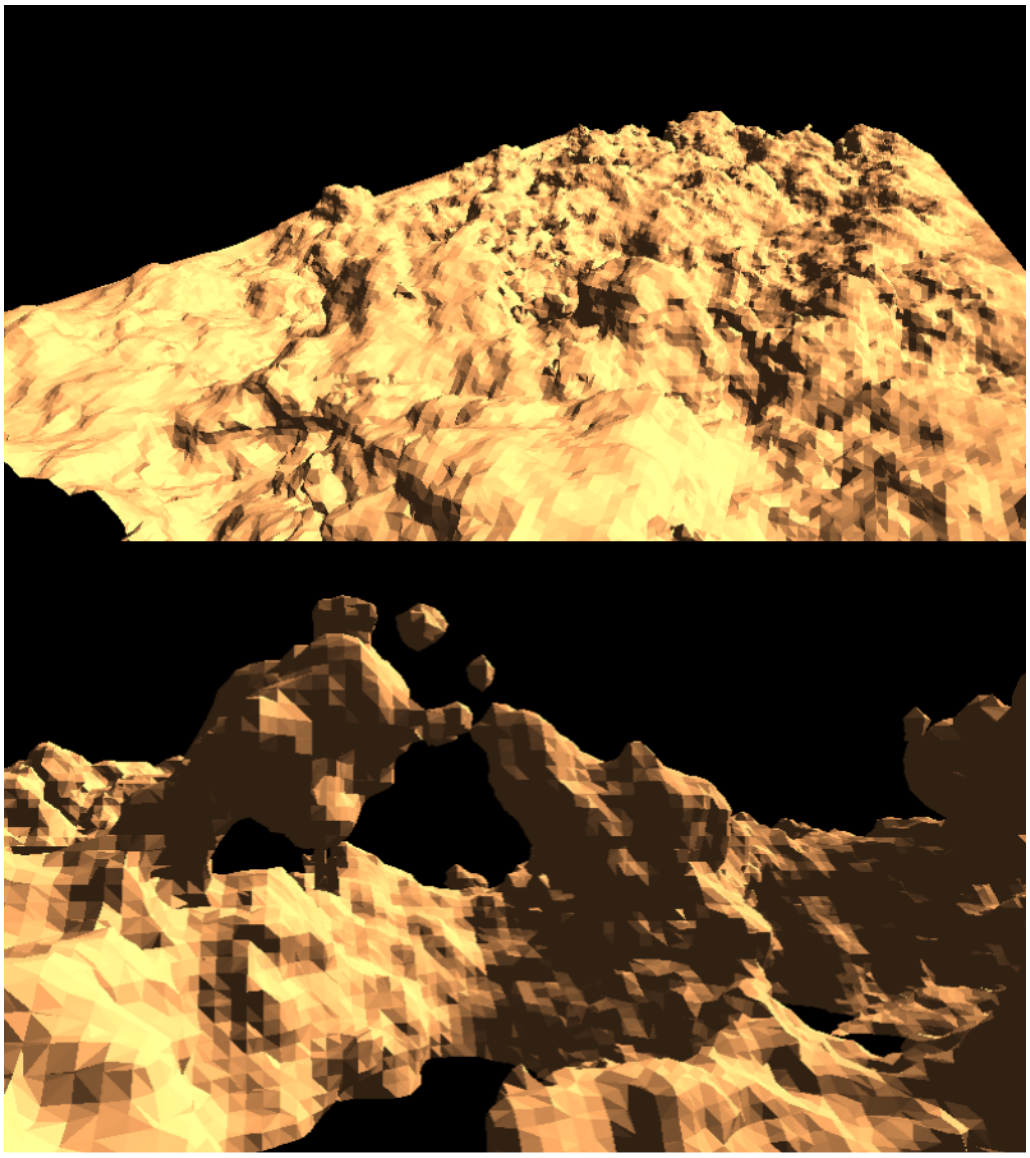

Figure 3.7.2: *Top:* A terrain field using Marching Cubes. *Bottom:* The kind of terrain that can not be represented by heightfields.

The terrain generated by this algorithm is, however, only large enough to cover the playable area and does not span a large enough area to be used for the surrounding world. Expanding the scalar field would be too costly, so we would have to use another technique to generate this. Since we do not specifically require arbitrary concavities, we could use heightfield methods.

The problem with generating the terrain for the planet then lies in seamlessly stitching these solutions together. Since there is no way for the heightfield solution to conform to the potential edges of the Marching Cubes terrain, we have to make the edges of the playable terrain be within the range of the heightfield. The easiest way to do this would be to have the amplitude of the terrain fade toward the base value on the sides of the terrain, i.e., when x is close to zero or the width of the terrain. This can be done using the power function and gives the result shown in Figure 3.7.3.

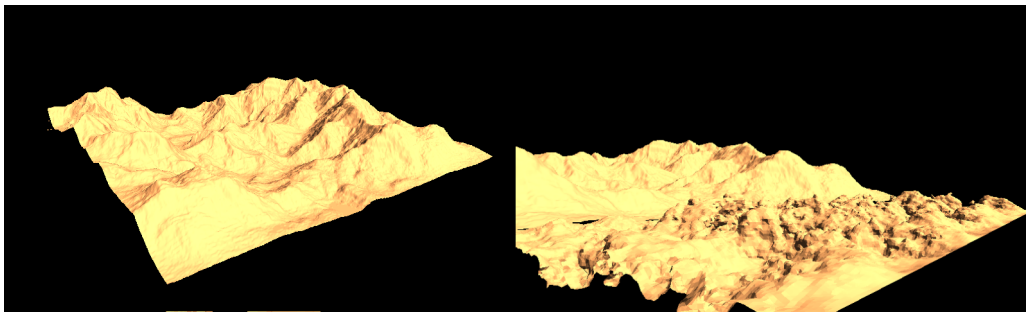

Figure 3.7.3: *Left:* A height field rendering. *Right:* A combination of a height field and Marching Cubes terrain.

We did find issues with artifacts, particularly noticeable floating rocks, which can be seen in Figure 3.7.4. Lengyel [Lengyel 2010] proposes some solutions to these problems, but we did not have time to implement this. We also had problems finding suitable normals for the triangles generated, which gave the terrain a very low resolution look.

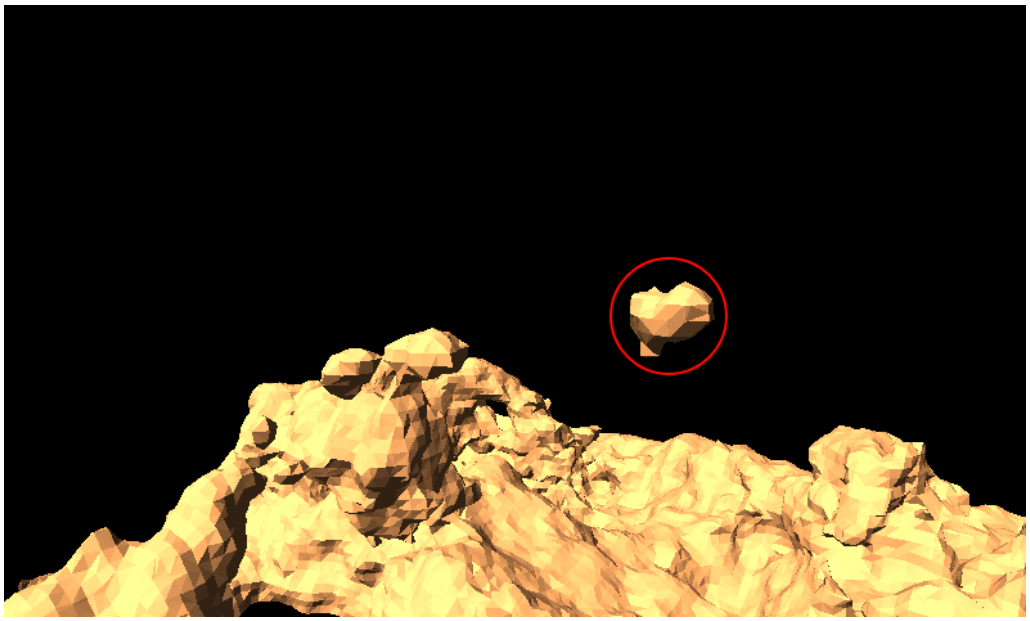

Figure 3.7.4: A floating rock artifact.

4

Conclusion

There has been a lot of explanation of what has been and what has not been implemented, followed by short discussions. The last part of the thesis is going to focus solely on the group's thoughts of the results, as well as our thoughts regarding further development.

4.1 Results

After four months of work, a visually pleasing space shooter game has been created.. Even though the game might not keep up with the standards of many commercial games, given the resources and time frame, we are still proud of the outcome.

The game is easy to play and the visual effects make the game look good. Each effect contributes to the look of the game, for example, the Bloom effect makes light sources seem brighter/glow, and the particle engine provides explosions and gives the spaceship a good looking thruster effect. Something that turned out exceptionally well was the volumetric light scattering the sun emits, especially when travelling through an asteroid field, this makes for a very good looking effect as well as a better feeling for traveling in space.

Unfortunately, the game does not run as smoothly as it was initially supposed to do. The goal was to have a steady 60 frames per second on most hardware, but unless the computer at least has a mid-range graphics card this might be hard to achieve without lowering the quality of some graphical effects, something that can be done from the settings menu in the game.

4.2 Discussion

At the start of this project the ambition might have been somewhat optimistic, and since problems usually take longer than expected to solve, some features had to be dropped during development. Some of these dropped features will be discussed in more detail in Section 4.3 Future Work.

Microsoft XNA has been quite easy to work with and we encountered relatively few problems, and even though the framework is a few years old, and might not be able

to keep up with modern game engines such as Unity and Unreal Engine, the end result is quite good. Had we used such a game engine instead, the game would likely have been developed faster; however, we would probably have missed several learning opportunities.

One thing that was decided in early development was that the game had to be procedurally generated to increase replayability; it is much more efficient to let the game generate a new level and increase difficulty of the enemies as the game progresses instead of letting one team member design lots of levels. The levels generated are quite similar to each other, which is primarily due to the lack of content; there are only a few enemies and other obstacles, such as asteroid fields implemented, which does not make for much variation.

The group did not spend much time together when implementing this project, and the reason for this was that not every group member had their own laptops. There are a lot of positive aspects to working together: when problems occur everyone can help, ideas can be discussed right away, and you get to know each other better, making it more motivating and fun to work on a project. In hindsight, we should probably have utilised the rooms with computers provided by Chalmers in order to increase productivity.

4.3 Future Work

There are, of course, features that had to be dropped during the development of this project due to the lack of time. If the group were to continue further development, some of these features would be reconsidered and implemented into the game.

More content is something that would be the first thing to be added to the game, since this is something that the game is lacking at the moment. For example, we would like to have greater diversity in enemies with different behaviors and formations they attack in, as well as in their ship and weapon models. Some sort of boss or tough enemy, with a wider range of weapons and higher health than the normal enemies, would be nice to have implemented as well; difficult enemies would give the player a tougher fight and a sense of achievement when beating the stronger opponent.

A proper progression system, where the player receives better weapons or more health over time, has also been discussed. This could be implemented in a number of ways, either by collecting upgrades dropped by enemies or by receiving an upgrade when a certain number of enemies have been defeated by the player. Regardless of the specifics, this will give the player a feeling of progression and thus will inspire them to keep on playing.

Another thing that we would like to implement, but did not have the time for, is generated terrain, something that has been mentioned earlier in this thesis but is worth mentioning again. Not only would entering a planet's atmosphere and flying

through the hills and valleys heighten the gameplay experience, it would also add new good-looking environments as well as diversity to the standard space environment.

Having the camera angle change during gameplay was something that we discussed for a long time, before it was finally dropped due to lack of time and the potential problems that comes along with the perspective changes. Nevertheless, the idea itself is still solid and would add a range of new possibilities in terms of gameplay; screenshots of how it might look can be seen in Figure 4.3.1.

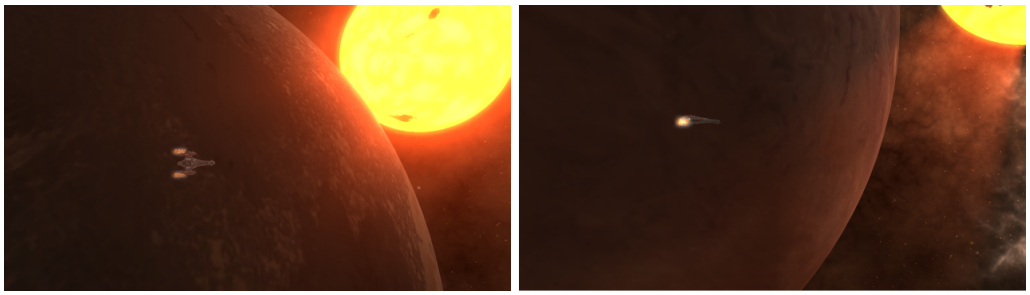

Figure 4.3.1: *Left:* An image showing the game from above. *Right:* An image showing the game from the side.

There would also be more time put into the user interface since it is quite basic at the moment; it shows the information needed, but not in an aesthetically pleasing way. There would also be more time spent on models, textures, and sound. This is due to a lack of variation at the moment, with limited resources repetition is inevitable and it does not make for good gameplay.

Bibliography

AKENINE-MÖLLER, T., HAINES, E., and HOFFMAN N. 2008. *Real-Time Rendering*. Third Edition, Boca Raton, CRC Press.

BILLETER, M., SINTORN, E., and ASSARSSON, U. 2010. *Real Time Volumetric Shadows using Polygonal Light Volumes*. High Performance Graphics (2010), pp. 39-45.

BLINN, J. 1978. *Simulation of Wrinkled Surfaces*. ACM SIGGRAPH Computer Graphics, vol. 12, no. 3, pp. 286–292.

CABRAL, B., LEEDOM, L.C. 1993. *Imaging Vector Fields Using Line Integral Convolution*. ACM SIGGRAPH '93 Proceedings of the 20th annual conference on Computer graphics and interactive techniques, p. 264.

CHEN, S. E., WILLIAMS, L. 1993, *View Interpolation for Image Synthesis*. ACM SIGGRAPH '93 Proceedings of the 20th annual conference on Computer graphics and interactive techniques, pp. 279-288.

COHEN, J., OLANO, M., and MANOCHA, D. 1998. *Appearance-preserving simplification*. ACM SIGGRAPH '98 Proceedings of the 25th annual conference on Computer graphics and interactive techniques, pp. 115-122.

COOK R., PORTER T., and CARPENTER, L. 1984. *Distributed Ray Tracing*. ACM SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques, vol. 18, no. 3, pp. 137-145.

CROSS, T. 2014. *Why video games are so expensive to develop*. The Economist, [Online] 24th September, Available from: www.economist.com, [Accessed 18th May 2015].

DEMERS, J. 2004. *Depth of Field: A Survey of Techniques*. GPU Gems, Boston, Addison-Wesley, ch. 23, pp. 375-390.

FARIN, G. E. 2001. *Curves and Surfaces for CAGD: a Practical Guide Fifth Edition*. Morgan Kaufmann, ch. 5.

FISHMAN, B and SCHACHTER, B. 1980. *Computer display of height fields*. Computers & Graphics, vol. 5, no. 2, pp. 53-60.

FOURNIER, A., FUSSELL, D., and CARPENTER, L. 1982. *Computer rendering of stochastic models*. Communications of the ACM, vol. 25, no. 6, June.

GARDNER, G. Y. 1984. *Simulation of Natural Scenes Using Textured Quadric Surfaces*. ACM SIGGRAPH Computer Graphics, vol. 18, no. 3, July, pp. 11-20.

GEISS, R. 2007. *Generating Complex Procedural Terrains Using the GPU*. GPU Gems 3, Boston, Addison-Wesley, ch. 1, pp. 7–37.

HAEBERLI, P., and AKELEY, K. 1990. *The accumulation buffer: Hardware support for high-quality rendering*. ACM SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques, vol. 24, no. 4, pp. 309-318.

HALL, D. 2014. *ECS Game Engine Design*. California Polytechnic State University, June.

HAMMON, E. 2007. *Practical Post-Process Depth of Field*. GPU Gems 3, Boston, Addison-Wesley, ch. 28, pp. 583-606.

HECKBERT, P. S. 1986. *Survey of texture mapping*. Computer Graphics and Applications, IEEE, vol. 6 no. 11, 56-67.

HEIDMANN, T. 1991. *Real shadows, real time*. Iris Universe, no. 18, pp. 23-31.

HUNT, A., and THOMAS, D. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Boston, Addison-Wesley, ch. 2.

JAMES, G., and O'RORKE, J. 2004. *Real-time Glow*. GPU Gems, Addison-Wesley, ch. 21.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., and TACHI, S. 2001. *Detailed Shape Representation with Parallax Mapping*. Proceedings of ICAT 2001, pp. 205-208.

LENGYEL, E. S. 2010. *Voxel-based terrain for real-time virtual simulations*. University of California, Davis.

LORENSEN, W. E., and CLINE, H.E. 1987. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*. ACM SIGGRAPH Computer Graphics, vol. 21, no. 4, July, pp. 163–169.

MANDELBROT, B. B. 1982. *The Fractal Geometry of Nature*. New York, W. H. Freeman and Co.

MARROQUIM, R., and MAXIMO, A. 2009. *Introduction to GPU Programming with GLSL*. 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing, October, pp. 3-16.

MCHENRY, K., and BAJCSY, P. 2008. *An overview of 3d data content, file formats and viewers*. National Center for Supercomputing Applications.

MILLER, G. S. P. 1986. *The Definition and Rendering of Terrain Maps*. ACM SIGGRAPH Computer Graphics, vol. 20, no. 4, pp. 39-48.

MITCHELL, K. 2007. *Volumetric Light Scattering as a Post-Process*. GPU Gems 3, Addison-Wesley, ch. 13, pp. 275–285.

MUSGRAVE, F. K., KOLB, C. E., and MACE R. S. 1989. *The Synthesis and Rendering of Eroded Fractal Terrains*. ACM SIGGRAPH Computer Graphics, vol. 23, no. 4, July, pp. 41-50.

NAKAMAE, E., KANEDA, K., OKAMOTO, T., and NISHITA, T. 1990. *A lighting model aiming at drive simulators*. ACM SIGGRAPH Computer Graphics, vol. 24, no. 4, pp. 395-404.

NISHITA, T., MIYAWAKI, Y., and NAKAMAE, E. 1987. *A shading model for atmospheric scattering considering luminous intensity distribution of light sources*. SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pp. 303-310.

NYSTROM, R. 2014. *Game Programming Patterns*. Genever Benning, ch 2.

OLIVIERA, M. M., and POLICARPO, F. 2005. *An efficient representation for surface details*. Instituto de Informatica UFRGS.

PERLIN, K. 1985. *An Image Synthesizer*. ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, July, pp. 287-296.

PERLIN, K., and HOFFERT, E. M. 1989. *Hypertexture*. ACM SIGGRAPH Computer Graphics, vol. 19, no. 3, July, pp. 253-262.

POLICARPO, F., OLIVIERA, M. M., and COMBA, J. L. 2005. *Real-time relief mapping on arbitrary polygonal surfaces*. ACM SIGGRAPH Proceedings of the 2005 symposium on Interactive 3D graphics and games, April, pp. 155-162.

POTMESIL, M., and CHAKRAVARTY, I. 1982. *Synthetic image generation with a lens and aperture camera model*. ACM Transactions on Graphics, vol. 1, no. 2, pp. 85-108.

POTMESIL, M., and CHAKRAVARTY, I. 1983. *Modeling Motion Blur in Computer-Generated Images*. ACM Transactions on Graphics, Vol. 17, no. 3, pp. 389-399.

REEVES, W. T. 1983. *Particle Systems - a Technique for Modeling a Class of Fuzzy Objects*. ACM Transactions on Graphics (TOG), vol. 2, no. 2, April, pp. 91-108.

ROSADO, G. 2007. *Motion blur as a post-processing effect*. GPU Gems 3, Boston, Addison-Wesley, ch. 27, pp. 575–581.

SCHULLER, D. 2010. *C# Game Programming: For Serious Game Creation*. Cengage Learning PTR, ch. 2.

SCOFIELD, C. 1992. *2 ½-D Depth-of-Field Simulation for Computer Animation*. Graphics Gems III, San Diego, Academic Press, pp. 36–38.

SUNG, K., PEARCE, A., and WANG, C. 2002. *Spatial-Temporal Antialiasing*. IEEE Transactions on Visualization and Computer Graphics, vol. 8, no. 2, pp. 144-153.

TOGELIUS, J., KASTBJERG, E., SCHEDL, D., and YANNAKAKIS, G. N. 2011a. *What is procedural content generation?: Mario on the borderline.* ACM SIGGRAPH In Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, June, p. 3.

TOGELIUS, J., YANNAKAKIS, G. N., STANLEY, K. O., and BROWNE, C. 2011b. *Search-based procedural content generation: A taxonomy and survey.* IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 172-186.

WELSH, T. 2004. *Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces.* Infiscape Corporation, pp. 1-9.

WILLIAMS, L. 1978. *Casting Curved Shadows on Curved Surfaces.* ACM SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques, pp. 270-274.