CHALMERS

A Case Study in Rapid Game Development using XNA Game Studio

Bachelor's Thesis
Computer Science and Engineering Programme

FREDRIK BERGGREN ANDERS JOSEFSSON

ELIAS HOLMLID STEFAN MIKAELSSON

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2008

A Case Study in Rapid Game Development using XNA Game Studio

Department of Computer Science and Engineering

Chalmers University of Technology

SE-412 96 Göteborg

Sweden

1 ABSTRACT

This thesis presents a case study on how a complete game can be developed during a limited timeframe. The game development is done using modern programming languages and frameworks, which deviates from the current industry standard. Specific agile software engineering development models, such as rapid iterative prototyping, are also incorporated into the development process.

It is shown how these above-mentioned factors aid and contribute to a complete, playable and enjoyable game within the specified timeframe. There has also been emphasis on constructing a visually compelling game. Thus, also a large portion of this study is dedicated to graphical effects, game physics and game logic.

2 INTRODUCTION

2.1 BACKGROUND

The video games industry is today a very successful one. A report entitled *Global Entertainment and Media Outlook: 2007-2011* (Scanlon, 2007) anticipates that the market will grow at a compound annual rate of 9.1% over the next five years. This makes the games industry the third fastest growing segment of the entertainment and media market after TV distribution and Internet advertising (Scanlon, 2007).

It all begun in 1962 when the game *Spacewar!* was created by a group of students at MIT, becoming the first widely available video game. *Spacewar!* is often credited as the first true video game and it has influenced games ever since. During the 1970's the commercial industry was born and the market expanded until 1983, when it collapsed, mainly because of a hyper-saturation of the market. A few years later, the industry revitalized, mostly due to the success of the Nintendo Entertainment System (History of video games, 2008).

During these early years of computer games, many of the released games were crafted by just a single person who did all the programming, graphics and sound. Today games are huge productions, often involving hundreds of people. This has made it necessary to employ sound software engineering methods and more detailed planning of the projects (Finley, 2007).

Computer science is a relatively new discipline, and even if many methods have been developed in order to assure the quality of software and to decrease development times, released software often contain bugs and deadlines are not kept.

To avoid reinventing the wheel for each project, companies often license ready-to-use engines which they can modify to suit their particular games. On a lower level, several frameworks exist for graphics, for example OpenGL and Direct3D. These APIs are intended to make it easier for the programmer to interact with the graphics hardware without having to target a specific graphics card. They are not suitable for games only, but are intended to be used for all types of applications relying on graphics for their presentation.

In this thesis we use a relatively new framework from Microsoft which goes by the name XNA. XNA is by no means a ready-to-use 3D game engine, but the abstraction level is somewhat higher than APIs like Direct3D and OpenGL (Microsoft XNA, 2008).

3 PROBLEM

The task of implementing an entertaining game is an interdisciplinary one. It poses a vast number of questions and challenges ranging from questions such as "What is a game?" and "What is entertaining?", at one end, to "What are the benefits of Microsoft's game developing framework XNA?" at the other. Between philosophy and practicalities we encounter difficulties of physics simulation-, computer graphics- and software engineering.

Using the technical advances in game design, real-time rendering and modern game development frameworks we investigate what can be achieved in terms of a limited, but complete, functioning game with high quality graphics and performance within the limits of a bachelor thesis.

4 PURPOSE

The main purpose of this study is to develop a complete and visually appealing game during the timeframe of four months. As such, the following aspects of development, design and software engineering will be taken into consideration:

- Establish what impact modern programming languages, frameworks and development environments have on a final product. This study will try to demonstrate the contribution these factors present in terms of quality and progress.

- Evaluate current software and game design pattern's efficiency on a basis of team development. Also, a presentation of how agile software engineering methods aid product completion according to established requirements while remaining on schedule will be included.

- Make use of well documented methods for physics and graphical effects in order to quickly evaluate if they fit our needs, and then modify them to suit our particular requirements.

- Incorporate ideas and concepts from game enjoyment studies to create a fun and challenging game while maintaining playability.

5 DELIMITATION

Game Development is often a lengthy process with schedules ranging from six months to two years. Few games are developed in less than six months (Bates, 2004). Due to our limited time frame, this thesis will take the form of a case study. This means that development will only be evaluated in a single modern language and accompanying framework; methods that are implemented will be evaluated exclusively and methods not used will not be as thoroughly examined.

Emphasis will be on game engine design, game mechanics and graphics. Thus, we will not engage too deeply into game concept design.

6 METHOD

We use an iterative process where a single iteration can be split into the following stages:

1. **Team brainstorming**

We usually set up limits for what is allowed to be brought up. For example, when discussing a new level design, we might only permit a limited number of planets.

2. Research

The time put into research is very subject dependent. Research is sometimes done before Team brainstorming, but usually very draft, as we will go back to brainstorming after the first iteration has been completed.

3. Implementation

Each implementation task is split between team members, and a deadline is set. A new iteration on another subject will sometimes start here, as all team members might not be needed to implement smaller tasks.

4. Testing

Testing is to be done by all members of the team after a deadline has been reached. There are multiple iterations of Implementation and Testing within an overall iteration cycle.

5. Documentation

Document arguments of choices, interface implemented, how it was done and what problems were encountered.

7 The Game Megachile Pluto

7.1 Introduction

In the game developed, entitled Megachile Pluto, the player controls the rabbit Chow (see Figure 1), who is viewed from behind in a 3D solar system. The aim of the game is to save a dying planet from extinction by bringing them the items they need to survive. In each solar system, a multitude of planets orbit a burning and hostile sun which the player needs to avoid. The player will need to visit the different planets by jumping to them since they contain the items the dying planet need. Some of the items sought after are not to be found in the solar system, but the player can combine items at certain planting spots. If blended together correctly, new forms of life will appear. For instance, on the example level developed for this bachelor thesis, a red flower combined with a mushroom spawns a melon. The dying planet's life force diminishes constantly and the player will need to act quickly and wisely if he intends to succeed in his task of saving the planet.

Figure 1. The title screen for Megachile Pluto, with Chow in the lower right corner.

8 Software Engineering

8.1 Key Construction Decisions

8.1.1 Programming Tools

Instead of the two more commonly used APIs, OpenGL and Microsoft Direct3D; we choose to use XNA Game Studio, which is a new game development framework produced by Microsoft. XNA was first introduced to developers at the Game Developer's Conference in 2004, and is thus

a more recent addition to graphical software development than OpenGL, which was introduced in 1992 (Silicon Graphics Inc., 2008) and Microsoft Direct3D, which was introduced in 1994.

XNA might be the future game development standard for PC and Xbox. It was conceived to integrate a development environment across all Microsoft platforms, and also free developers from doing repetitive mundane implementations (Microsoft Corporation, 2004). XNA includes the latest version of DirectX 9, in addition to various Xbox Tools. The more prominent of these tools being the Cross Platform Audio Creation Tool (XACT) (Irish, 2005, pp. 173-174)

While features such as portability between PC and Xbox was appealing, the main reason for using the XNA Game Studio is that it supports C#. Working with a high-level language, like C#, improves productivity, reliability, simplicity and quality (Jones, 1998), (Boehm, et al., 2000).

8.2 Development Method

8.2.1 Background

In the 1980s and early 1990s, the most common software engineering model was the waterfall model. This model consisted of a single development cycle, where every aspect of the development was completed in a sequential order and only a single time. Approaches of this kind involve a significant overhead in planning. Moreover, documentation is costly and final deployment of the product may take several years (Sommerville, 2007, p. 396).

Sommerville concludes that while this development model works well for systems that are large, long-lived and critical, the overhead in planning and documenting such systems is too large when applied to small and medium-sized business systems. This led to new development models known as agile or rapid development methods, which all revolve around iterative approaches. One of the most common agile methods is extreme programming (Beck, 2000). It has been suggested that this type of development is ideal for game development:

"Rapid iterative prototyping is the best development model for most new games." (Bates, 2004, s. 226)

8.2.2 Development approach

A software engineering model with an iterative approach was chosen. Guidelines for choosing between sequential and iterative design approaches (McConnel, 2004, ss. 35-36), were used, as well as principles defined by (Beck, o.a.). These principles are essential in all agile methods. The reasons for choosing an agile development method were the following characteristics:

- Requirements are loosely defined, not well understood and have a high probability to change in the future.
- The development team has limited experience with this type of development.
- The product's long-term predictability is not important.
- Early and frequent delivery of working software has high priority.

All these points advocate the use of an iterative approach, and as such, an agile development model was chosen. However, we decided to maintain a flexible approach to agile development and no specific method was chosen. Instead, elements of several methods that integrate well into the study have been used.

8.2.3 Agile Development Aspects Used

8.2.3.1 Incremental delivery

A working version of the game was delivered every two to three weeks, after which the development cycle began anew. In the beginning of each increment, we specified the requirements to be included in the next increment, after which construction, design and testing followed. This is a characteristic of agile development that is common to most variations of it (Sommerville, 2007, p. 391), (McConnel, 2004, pp. 58-59). This process was chosen because it guarantees a working final product. Figure 2 below depicts the release cycle used for this study.

Figure 2. The release cycle used in this study.

8.2.3.2 Tickets

In order to make problems less complex, the requirements can be divided into smaller, more manageable tasks. These are called tickets, which are distributed among the developers (Bates, 2004).

8.2.3.3 Software Prototyping

Prototyping is an approach that can be used when creating a software system, such as a game engine (Sommerville, 2007). In our case, the throw-away prototype that was created was an implementation of the game engine itself, and both continued to be developed in parallel for continued use throughout the course of development. This prototype was used to evaluate the new requirements needed, experimenting with various solutions as well as testing other aspects of the engine during its development.

8.2.4 Component-Based Software Engineering

8.2.4.1 Background

In addition to the rapid and evolutionary development model previously outlined, component-based software engineering (CBSE) has been used. The following summarizes the essence of component based software engineering:

"The primary role of component-based software engineering is to address the development of systems as an assembly of parts, the development of parts as reusable entities, and the maintenance and upgrading of such systems by customizing and replacing such parts." (Crnkovic, 2003).

8.2.4.2 Method

The methods used are drawn from ideas outlined by the CBSE project, which is being researched at Andersen Consulting. The CBSE project aims to combine ideas from research and prominent methods currently used in industry that support component-based development (Ning, 1997). Here follows some of the methods presented by Ning.

8.2.4.2.1 Interfaces

Interfaces or abstract classes are created to define functionality of certain components before they are implemented. The parts in our implementation that show this most clearly are the BasicObject and BasicSystem abstract classes, which define the functionality of all other objects and systems within the game engine.

8.2.4.2.2 Wrappers and Integration

We did not acquire any third-party components, and all components were developed internally by the development team. Because of this, no integration or wrapper problems were encountered.

9 COMPUTER GRAPHICS

9.1 LIGHTING

9.1.1 USING THE SUN AS LIGHT SOURCE

In the game, each level is built with a sun positioned in the middle of the solar system. We decided that we would also use it as the primary light source in our lighting model. Nevertheless, there is one drawback here, being that the planets will only be exposed to the lighting on the parts facing the sun. Essentially, two different alternatives were discussed:

- Use up to six different light sources in order to properly light up all parts of the world objects at the same time.
- Bind a light to the player which will always light up the area where the player is located.

The main reasons for finally choosing a single light source were that we felt it would be more convincing to use the sun as a light source, and also because using multiple lights would be rather costly in terms of performance. In our minds, a light following the player would also appear awkward. Given the tight schedule, we had no time to test and evaluate all the different approaches.

9.1.2 POINT LIGHTS

In real time computer graphics, three types of lights are mainly used – point lights, spotlights and directional lights. A point light spreads light in all directions, and a spotlight spreads light only within the cone of the light. A directional light approximates all incoming light rays as parallel, and is often used in outdoor scenes, where the rays from the sun are approximately parallel (due to that it is so far away it could be considered infinitely far away). The sun we are using in our game is modeled as a point light since it is near the player and the player can be on any side of it.

9.1.3 THE LIGHTING MODEL

We utilize a somewhat simplified version of the Phong lighting model. This model is widely used in games, and gives convincing results while being well suited for the

parallel nature of modern graphics processors. It has been sufficiently explored elsewhere, so we will not treat it here. A good and complete explanation can be found in (Akenine-Möller & Haines, 2002, ss. 67-84). The difference in our model is that we use a constant factor for all ambient light instead of calculating it from the ambient components of the light and material. We did this because we were using about the same ambient component for all materials throughout the scene. Due to the fact that we only have one light source, we could then get rid of some parameter passing to the shaders.

9.2 Normal Mapping

We make use of normal mapping to a great extent in the game. Normal mapping is a technique to fake surface depth, like a brick wall for example, without having to add extra geometry. It would require a significant amount of additional polygons if this fine detail should be modeled geometrically. Since the extra detail is small, it mostly affects how the object is lit. Therefore, for each texture applied to the objects, an additional texture is passed to the shader with information about how the normals are pointing on the surface. So, instead of interpolating the normals across the faces of the triangles and thus getting a normal corresponding to each pixel, a lookup is done in the normal map instead (St-Laurent S. , 2005, ss. 60-62).

Constructing the normal map from a texture could be done using a tool, like for example the NVIDIA Normal Map Filter (NVIDIA, 2007). Then, for each vertex a tangent space has to be created, which is used to construct the matrix necessary for transforming all components involved in the lighting calculations into the coordinate system the normals reside in. This base can be constructed automatically for us in XNA, and we will not treat its derivation here. A thorough explanation of the tangent space derivation can be found in (Gath & Dreijer, 2006). Since the components of the normals also need be in the [-1, 1] interval and the components of the textures are in the [0, 1] interval, a simple transformation also have to be computed at each lookup (Luna, 2006, s. 542).

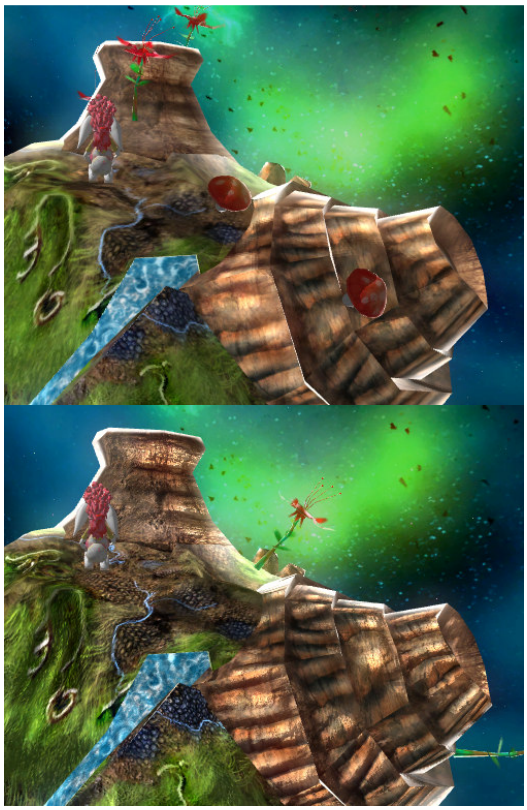

Figure 3. The upper image is without normal mapping. The lower image illustrates the same planet with normal mapping applied. Especially note, on the volcano in the bottom image, how the cracks appear to have more varied depth than in the top image.

9.3 PARTICLE SYSTEMS

Particle systems are often used in games to simulate small objects which move in a similar, yet slightly random, way. Once the particles are emitted, they are pretty much independent, and can be affected by forces like wind and gravity. Their color and opacity may also change over time, for example. Some particle systems take into account collision between other world objects, but collisions between particles are not very common (Hall, 2008, ss. 507-511). Examples of phenomena suitable for simulation by particle systems include smoke, rain and explosions.

9.3.1 PARTICLE SYSTEM FRAMEWORK

In order to get a flexible particle system framework, we constructed an abstract base class which all particle systems should implement. We also implemented a custom vertex format used by all particles. This vertex format includes the start position of the particle, its velocity, its initial size and other variables. At each frame, this information is passed to the shader that the particle system is configured to use. Since most particle systems involve the motion of particles, the new position is calculated in the shader using the starting position, the velocity and the time passed since the particle was created.

9.3.1.1 Point sprites

We use point sprites for the particles. Unlike an ordinary point primitive, a point sprite can have an associated texture and can vary in size. Point sprites were introduced in Direct3D 8.0. Before that, *billboards* were often used for particle systems. A billboard is a quad which is oriented so that it is always facing the camera. This requires four vertices instead of only one, so using point sprites is clearly preferable (Luna, 2006, ss. 483-484).

9.3.1.2 Pooled resource

Since new particles are frequently created, using a pooled resource is useful. This means that a maximum number of particles is set for the system and that a list is created that hold these particles. Two additional lists are also created, which index into this list. In this way, we need not allocate new memory at any time during the lifetime of the particle systems (Hall, 2008, s. 511).

A new particle is created only if the following criteria are fulfilled: that a specified time has passed, and the maximum number of particles has not been reached. During each update of the particle systems, the following is executed on the CPU side:

1) Increment the age of the particle system.
2) Fill the two arrays indexing into the particle pool, one with the indices of dead particles, and one with the indices of the alive particles
3) Emit new particles if the criteria stated above are fulfilled. Every time a new particle is created, it is not truly created, but is initialized at a position in the pool pointed to by one of the indices in the array of dead particles
4) Render the particles by passing them with a single draw call into the preferred shader.

9.3.2 *Results*

In Megachile Pluto, we use particles for several different effects, for instance the fire burning at the sun surface, the stars marking out the planting spots and the smoke rising from the volcanoes. All of them use some kind of transparency blending, which have some on performance. A drop in frame rate can in this context mostly be observed when looking at the sun or volcano smoke up close, since the particles will then be large and overlap to a great extent (Latta, 2004).

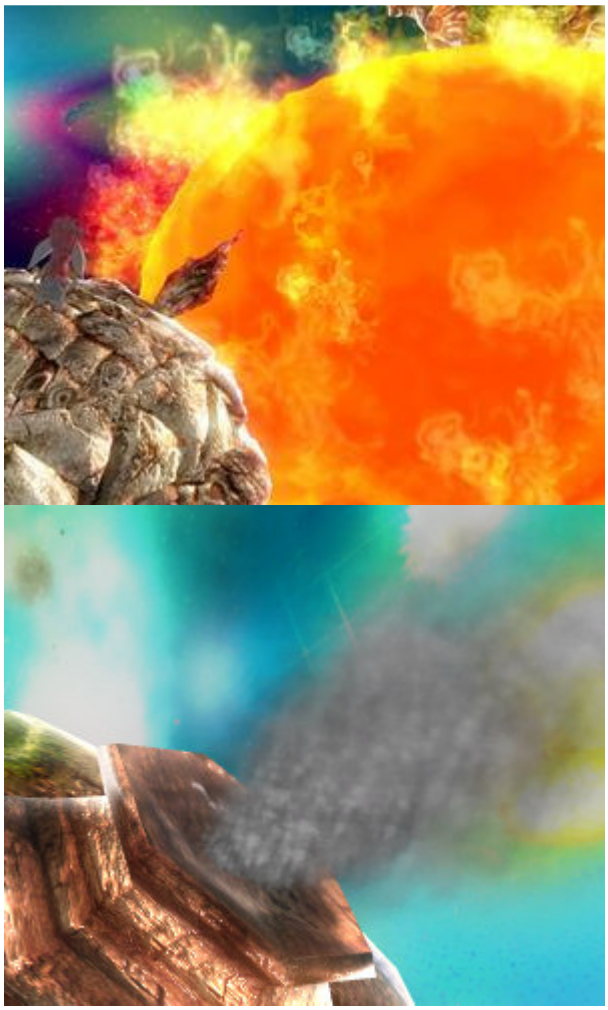

Figure 4. The fire surrounding the sun and volcano smoke particle systems

9.4 Shadows

Shadows help the viewer of a 3D-scene to convey the relative positions of objects (Luna, 2006, s. 567). We decided early on that we would utilize shadows in our game. Given that we have a world with no up and down which initially might be hard for the player with respect to orientation, using shadows seemed even more important. As described earlier, our only light source is the sun which is positioned at the center of the world coordinate system. Our first approach was to cast shadows realistically in all directions from this light source, but in the end we settled for a simpler solution. A description of the basic techniques used follows below, thereafter followed by a discussion about our experiences with the techniques, what decisions we took and why.

9.4.1 *Shadow Mapping*

There are basically two major techniques used in modern computer graphics to render a scene with shadows – shadow mapping and shadow volumes. We use shadow mapping, an image based technique invented by Lance Williams in 1978 (Williams, 1978).

Another technique worth mentioning is *projected planar shadows*, but this method has the limitation the name implies; it can only cast shadows onto planes. We tried several different approaches for how to use shadow maps in our engine. However, all of these were rooted in the most basic shadow mapping technique.

Shadows are created in the areas occluded by objects when viewing a scene from the

position of the light. This fact is used in the shadow mapping algorithm. First, an off-screen buffer is rendered, which afterwards will contain the depth values inside the viewing frustum of the light source. This is accomplished by transforming each vertex from world space into the view space of the light. From there, the vertex is transformed with the light's projection matrix, and the depth value is stored. By taking advantage of the Z-buffer algorithm, when finished rendering, the buffer will contain the z-value closest to the light.

When rendering the scene, each pixels z-value (in light space), is compared to the z-value written into the shadow map. If the value in the shadow map is less than this value, the pixel is in shadow. Since the map is sampled, the technique can result in aliasing artifacts. This is mostly because objects shadow themselves, a problem called *self-shadow-aliasing*. As the name implies, the depth comparison results in that an object incorrectly shadows itself. One common solution is to add a little offset to the comparison, a *bias factor* (Akenine-Möller & Haines, 2002, ss. 271-272).

9.4.1.1 Shadow mapping for omni-directional light sources

Shadow mapping is not particularly well suited for point lights. This is because of the limited *field-of-view* when generating a shadow map for the point light source. This has been one argument for using *shadow volumes* instead. Using multiple shadow maps is one solution to the problem. In order to capture completely the surroundings, up to 6 different shadow maps might have to be used. There are other methods also, like *Paraboloid Shadow Maps* (Brabec, Annen, & Seidel, 2002).

9.4.1.2 Soft shadows

The shadow cast by an object can be split into two parts, the *umbra* and the *penumbra*. The *umbra* is the region which is in full shadow, and the *penumbra* region is partially in shadow. A realistic soft shadow is sharper near the occluding object. However, the more advanced techniques for generating realistic soft shadows are often computationally expensive, and therefore simpler techniques are frequently used (Shastry, 2005).

PCF

One basic technique for achieving soft shadows is *percentage closer filtering*. Here, several samples of the shadow map are fetched, and the current pixel depth is compared to each one of them. After that, an average of the result is calculated. This results in that some pixels will be only partially in shadow (Luna, 2006, ss. 573-575).

9.4.2 RESULTS

Our initial approach was to use the sun as the shadow casting light and to use 6 shadow maps written into a cube texture. The reason we choose not to look into more advanced techniques was simply that we wanted to get the shadows up and running as soon as possible so that we could begin to focus on the actual gameplay.

Our first approach was to use a cube texture to capture the occluders in all directions. It is worth noting that the texture lookups in

HLSL uses a left-handed coordinate system (like in Direct3D), while XNA uses a right-handed coordinate system by default. (Scharl, 2007). Unexpected results can be produced if the implementer is not aware of this.

We experienced some performance problems after the full implementation of the cube map technique and resorted to a less complicated solution which required only one shadow map. Since the player in the game will mostly concentrate on the planet he is currently visiting, we enable shadows only for this planet. The switch occurs when the player lands on a planet and when he leaves a planet. At each frame the matrices of the shadow map are updated, so that they always encapsulate the current planet.

For the soft shadows, we first implemented a technique described in (Shastry, 2005). This technique uses the shadow map to first render the shadowed parts of the scene into a texture. The texture is then blurred. Thereafter, the whole scene is rendered, and the blurred shadow texture is projected onto the scene. This resulted in very smooth (although uniformly soft) shadows, but the impact on the frame rate was too severe for us at that point, so in the end we resorted to only using a *percentage closer filtering* with a kernel of 9 samples.

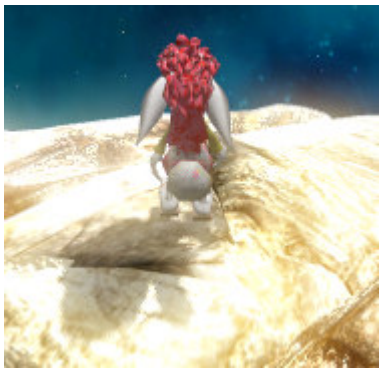

Figure 5. The shadow cast by the main character.

9.5 Post Processing

Post processing, or screen effects, refers to the technique of performing per pixel manipulations on a complete rendered scene. Examples of applications range from simple color manipulations, such as saturation/desaturation, to blurring, image perturbation and advanced lighting (St-Laurent S. , 2004).

9.5.1 Bloom and Lens flare

The light intensity range of the computer monitor does not mimic very well the intensity range between full shadow and direct sunlight found in a real environment. A number of methods can however be applied to give the impression of brightness and thus increase the level of realism in a rendered scene. One common strategy is to focus on image artifacts which the human eye and cameras cause when exposed to bright light. By exploiting the fact that these artifacts are associated with brightness the impression there of can be achieved (Akenine-Möller & Haines, 2002).

When looking towards a window from a dark room, the bright light from the window often produces a glow around it. The light appears to “bleed” outside the window

frame and contrast is perceived as dimmed. This occurs because the light coming from the window is scattered in the eye on its way to the retina. The specific image artifacts are commonly referred to as *blooming*. As previously mentioned, the computer monitor does not allow setting an intensity level where this bleeding occurs naturally. Explicitly rendering the artifacts is, however, possible and can be carried out in a number of ways (St-Laurent S. , 2004).

9.5.1.1 Bloom

The implementation used in the game to simulate bloom consists of four passes. First, the bright colors of the original scene are extracted by rendering them to a texture. The texture is then blurred horizontally and vertically using Gaussian blur. Finally, the original scene texture and the blurred texture are combined while adjusting the saturation level.

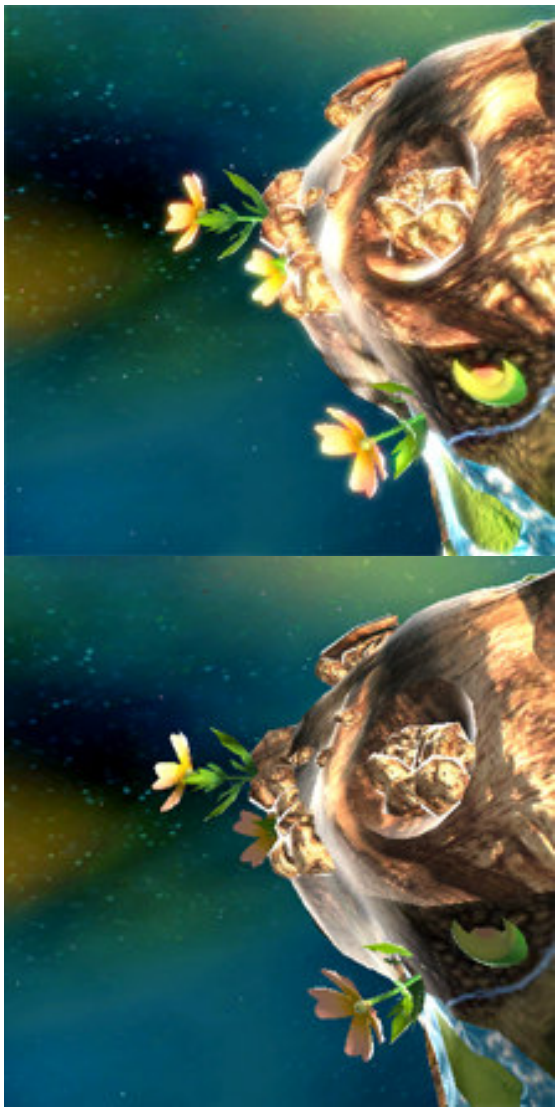

Figure 6. Upper image with *bloom, bottom* without.

9.5.2 Heat Haze

The air rising above a hot surface or an open flame sometimes makes the image of objects above it seem distorted. The phenomenon is known as heat haze, or heat shimmer, and can often been noticed when looking above an asphalt road on a hot summer day (St-Laurent S. , 2004).

Two main physical properties of air are accountable for this effect (St-Laurent S. , 2004). The first important property is that

light seems to go faster through hot air than through cold air, and the second is that the density of air decreases with temperature.

The degree with which the velocity of light, or other waves, decreases when leaving one medium for another is known as refractive index (Refractive index).

The fact that air density decreases with temperature makes the heated air above the flame or surface rise and small pockets of varying density and refraction index form. Because the light refracts non-uniformly above the heat emitter, the image seem hazy (St-Laurent S. , 2004).

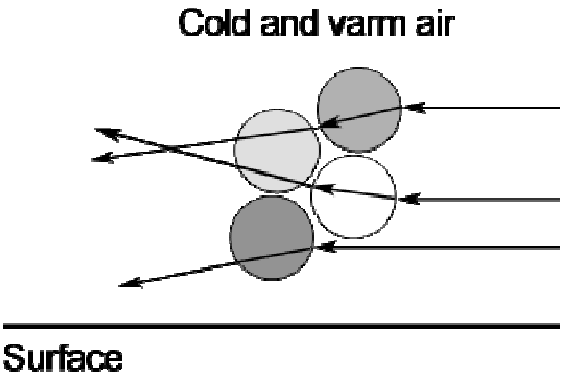

Figure 7. Light travelling through air above a hot surface.

9.5.2.1 Optimal Implementation

Whenever light from an object travels through regions of turbulence before it reaches the viewer, the rendered image representation of vertices needs to be distorted. In our simulated environment, this means that if we imagine a sphere around the sun where there is intense heat, we need to distort an object whenever light passes through the surface of this sphere on its way between an object and the viewer.

9.5.2.2 Actual Implementation

The algorithm used for the simulation of heat haze can be described as follows;

1. Render a distortion amount for each pixel to a texture, hereby referred to as heat texture, taking into account the position of rendered vertices, the position of the heat source and the position of the viewer. *See below for a complete description*.
2. In a second pass, use a noise texture to look up an offset vector for each pixel.
3. Sample the heat texture using an expandable Poisson disc expanding the radius by distortion amount.
4. Calculate the average of the distortion amount samples and add to the offset vector.
5. Use the resulting offset vector for lookup in the original scene texture.

9.5.2.2.1 Determining the pixels to be distorted and the degree of distortion

The pixel shader used for rendering the heat texture has two branches. If the object being rendered is behind the plane positioned at the center of the sun parallel to the projection plane, then the distort amount is based on the distance from the sun in the described plane. To give an appropriate distortion amount, the distance is scaled with the distance between the viewer and the sun and damped. The amounts rendered can be seen as the filled glowing circle in figure 8.

If the object being rendered is not behind the sun, the distortion amount is zero if the vertex is outside a sphere defined by a maximum heat range, and between zero and one depending on the three dimensional distance from the surface of the sun.

Figure 8. Heat texture

9.5.2.3 Results

The purpose of adding post processing to the game was to make the environment vivid, colorful and appear less artificial. The result is fairly pleasing but certainly does not come for free in terms of performance. The game is clearly fill rate bound and tradeoffs between quality and speed have to be made even when using quite powerful graphic accelerators. Luckily, there are a multiple of ways to trade in quality for frame rate in the context of post processing, for example by disabling multi-sampling for intermediate buffers, lower the resolution of buffers or rendering buffers less frequently.

Not surprisingly, the heat haze has the same influence on fill rate as bloom even though to a lower degree.

The purpose of implementing a heat haze effect was for it to act as a visual cue for the hazard of getting too close to the sun. It is in fact lethal, in real life as well as in the solar systems of our game. Whether the implementation serves its purpose or not is hard to anticipate but the effect behaves in an expected way, with some exceptions;

- Steep differences in distortion amount between adjacent regions in the heat texture cause unnatural glitches in the final image.

- When part of an object is rendered in the first branch, in the heat haze implementation, and another part of the same object is rendered via the second branch the object can be partly distorted even though all parts of the object is located at the same distance from the sun.

10 PHYSICS

10.1 BACKGROUND

For a game to be interactive and dynamic, objects need to have some kind of motion. One way to accomplish this is by manually animate all objects in a modeling program and then play them back in the game engine. Animating every object in a game world containing hundreds or thousands of different objects is not often a feasible option, and it is rarely the best solution for a fully dynamic object. In our case, imagine animating every curve the character or the camera follows for a single game instance.

Modern PC and console games usually have some kind of physics simulation to make the game world more realistic and believable.

One of the top selling games of 2007, Portal (developed by Valve), makes heavy use of physics and is using physics in a new and creative way. Eight of ten latest reviewed games at (Gamespot, 2008) are clearly using some kind of physics simulation. At a quick glance, the only two that does not use physics are both puzzle games.

There are plenty of third party physics engines available. Two of the most common ones are PhysX and Havok. A manager/wrapper can be created around both PhysX and Havok, but doing so will break the compatibility with the Xbox 360 due to the fact that the Common Language Runtime (CLR) on the Xbox 360 does not allow execution of native code. For C# and XNA, the third party physics engines for 3D simulation of physics are limited. There is a fully managed version of Bullet named BulletX in the works, but it is in alpha stage and may contain errors which render it unusable for use in a game that needs to be stable.

Due to the nature of our game, we needed a physics engine that could simulate rigid bodies in real-time with dynamic directional gravity.

10.2 TECHNIQUES

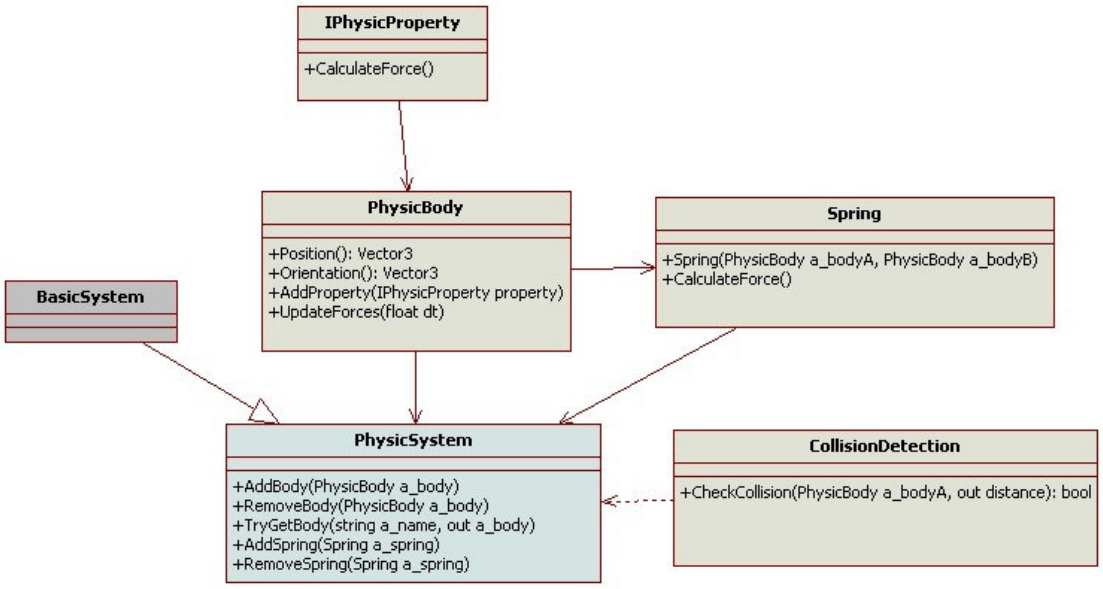

Figure 9. UML diagram of the physics implementation.

The above UML-diagram is the public interface, where the private functions and properties are excluded. The physics system inherits from BasicSystem, which gives it all the needed basic functionalities, such as for example the possibility to use an accumulated constant time step. It will also be easy to plug in and out of the engine and will be easy to access from Core.

IPhysicProperty is used to give a PhysicBody different properties. Planet, for example, uses the GravitySender (inside ref) property, which adds a gravitational force acting on all PhysicBodies with the GravityReciver property.

10.2.1 INTEGRATION

A fundamental difficulty with physics simulation is that computers use discrete time steps. Therefore, we need to do a numerical integration. The most basic implementation of numerical integration is Euler integration (Krantz, 2004), which is a first order procedure of solving ordinary differential equations. There are, however, limitations to this technique which are illustrated by the following example;

We have a scalar time step, t, which in our case is 0.02s (50Hz), a velocity vector that changes over time, v, and a position vector, s. To step the simulation one step, one can calculate the following equation to get the new position:

$$s = s0 + v * t$$

This simple approach works fine for this rather simple equation, if t is small enough, but it can generate huge errors if t is too big and the velocity changes over time (acceleration, a). Consider this:

(The following example is in one dimension for simplicity)

First test:

```
s0 = 0 m, v = 10 m/s, a = 100m/s^2,
dt = 1s

t = 1s: s = 0 + 10 * dt = 10,
        v = 10 + 100 * dt

t = 2s: s = 10 + 110 * dt = 120m
```

Second test:

```
s0 = 0 m, v = 10 m/s, a = 100m/s,
dt = 0.5s

t = 0.5s: s = 0 + 10 * dt = 5,
          v = 10 + 100 * 0.5 = 60

t = 1s:   s = 5 + 60 * 0.5 = 35,
          v = 60 + 100 * 0.5 = 110

t = 1.5s: s = 35 + 110 * 0.5 = 90,
          v = 110 + 100 * 0.5 = 160

t = 2s:   s = 90 + 160 * 0.5 = 170m
```

Changing the time step from one second to the half of a second generates a difference of 50m for a time interval of two seconds!

10.2.1.1 Runge-Kutta

More sophisticated algorithms have been developed which minimizes the problem described above. One of those is an integration procedure called Runge Kutta 4th order. Runge Kutta 4th order can detect the difference in the curvature of a function, thus making it more stable than Euler Integration. It takes four samples within the integration time-step frame and takes a weighted average of those samples.

When implementing Runge Kutta 4th order it is important to step the whole simulation simultaneously for each of the four samples. Otherwise, an object acting on a second object will have a difference of one to four internal Runge-Kutta 4th order step. Thus, the simulation will be less accurate (Krantz, 2004, s. 210).

10.2.2 Collision Detection

Our collision detection algorithm is rather primitive, but works well and gives sufficient performance for the needs of this project. Collision is detected for planets vs planets and player vs planets. For planets vs planets, we only use bounding spheres. For player vs planets, however, we use a combinition of a bounding sphere test and a ray-to-triangle test. First, a test to verify if the player can collide with the planet is executed using bounding spheres. If so, a ray is shot from the player into the center of the planet to find out if a collision occurs. For the ray-to-triangle test, an algorithm described by Tomas Möller and Ben Trumbore was implemented (Akenine-Möller & Haines, 2002).

10.3 Results

10.3.1 Performance

The collision detection implementation has not been optimized, but it uses less than 2% of the CPU in a given update cycle on our test level. Decreases in overall game performance have therefore not been an issue for these simulations. There are, however, optimizations that could have been done. Collision tests between planets are now done for every planet. This could easily have been optimized using an Octree (Akenine-Möller & Haines, 2002).

10.3.2 Believability

Due to the nature of a computer, physics is hard to simulate in a mathematically correct way, but also in a way consistent with the physical reality. We do not use real-world

values for mass and gravity and the player character has a damping force that acts on its body if velocity is too high. Orientation has also been tweaked to match the gameplay better. We found that a player playing the game can foresee what is going to happen, and that was our main concern when implementing the physics simulation.

11 Game Engine Design

11.1 Techniques

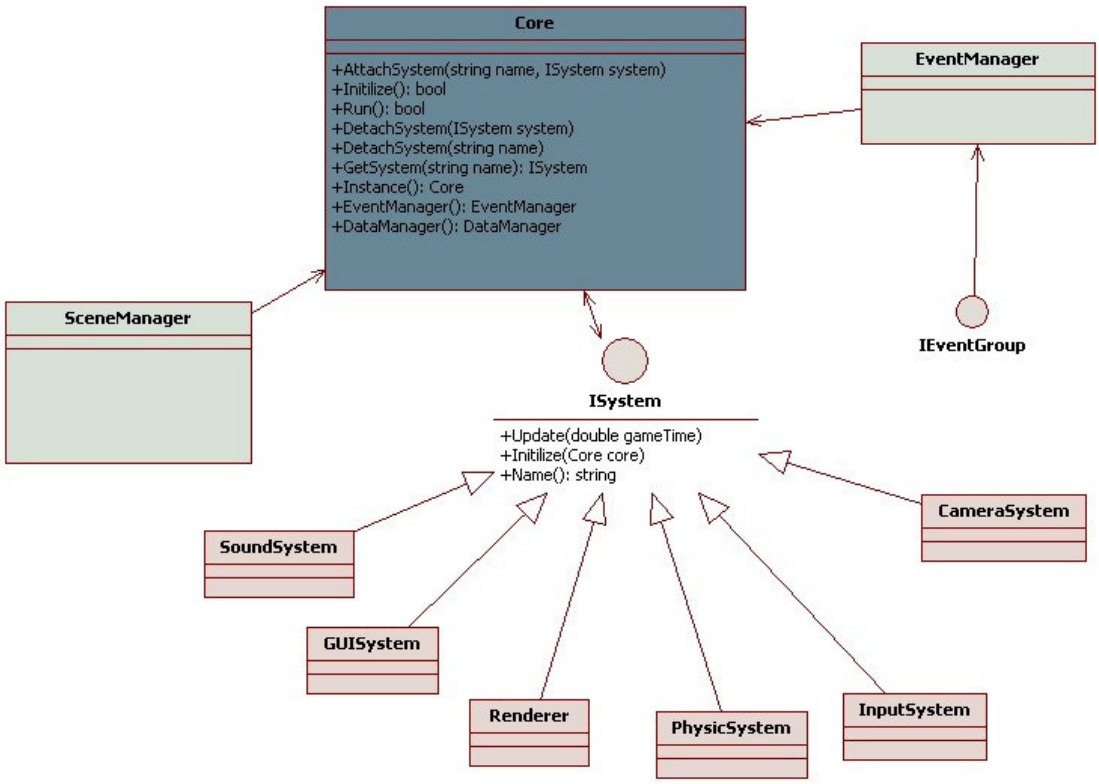

Figure 10. The core of the engine.

The core of our engine is the Core class which implements the Singleton pattern, thus making the Core interface reachable across the engine. Core takes care of updating all systems that have been added to it. A System has some basic functionality of its own; all systems can use a static time-step, and a profiler is applied to them which make it easy to measure CPU time in their update method. The ISystem interface is based on Julian Gold's (Gold, 2004) module design and makes it possible to add, remove, pause and fetch different system at runtime, thus making the testing of a system easy by just removing or adding the system.

11.1.1 Events

The EventManager can be accessed through a property in Core, making it accessible

throughout the engine. The main idea behind the EventManager is to decouple systems from each other. If, for example, a System should execute a function when a key is pressed, it is enough for the system to subscribe to the KeyPressed event in the EventManager, without knowing anything about the functionality of the InputSystem. One important aspect of the EventManager is that the InputSystem does not need to be present when a System subscribes to the KeyPressed event. It is therefore easy to remove the InputSystem if such a need should arise.

11.1.2 Scene

11.1.2.1 Method

A scene is handled by the SceneManager, which keeps track of all scene objects in the engine, and can be thought of as the game world. It can be, as the EventManager, accessed through a property in Core. All objects in the SceneManager inherit the BasicObject abstract class, which has basic functionality for orientation, position, and scale.

The physics simulation is updated at a speed of 60Hz and accumulates time to be able to guarantee a time-step of 60Hz. The main update loop, which also draws the objects, is updated at a variable speed. Here follows a demonstration of the problem:

A sphere at the center of the screen at position [0, 0] that is moving at a speed of one unit per second in the X-axis and the update speed of the game engine is somewhere near 40Hz. The physics simulation is using a static time-step of 60Hz. Delta time will be zero for the first frame and the sphere will be drawn at position [0, 0]. The next frame, the sphere will be moved 1s / 60 * 1 units on the X-axis, and the accumulated time for the physics simulation will be 1s/40 - 1s / 60= ~8ms. The third frame, the sphere will be moving 1s / 60 * 2 units, due to the accumulated time adding up to 1s / 60, which will make it move double the length of the last update even though the time between the frames are the same. Clarification is shown in Figure 11.

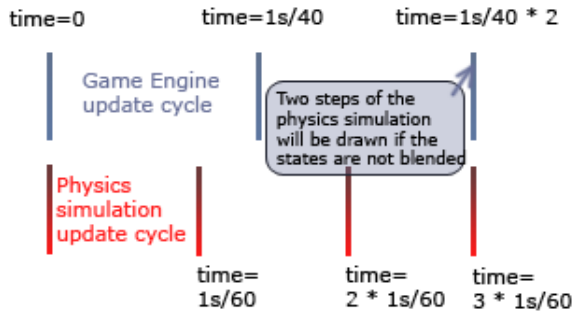

Figure 11. The fixed time step physics update

To overcome the above problem, the SceneManager interpolates between the previous and the current world matrix for all the objects. This is done using the following code:

float alpha = `AccamulatedTime / FixedTimeStep`

```
object.WorldMatrix = object.PreviousWorldMatrix * (1.0 – alpha) + object.CurrentWorldMatrix * alpha
```

Threading

To thread a game engine is not a trivial task, but is nevertheless an important one due to the recent year's development of CPUs. The Xbox 360 (Microsoft Corporation, 2006), Playstation 3 (Gorder, 2007) as well as the newer CPUs by Intel (Intel Corporation,

2008) and AMD (Advanced Micro Devices, Inc., 2008) all use a multi core architecture. The two most CPU intensive tasks in our engine are drawing calls and physics simulation, so letting them run in separate threads is a natural solution.

The most basic approach to data synchronization between threads is by using locks. However, it is difficult to thread a game engine that is not initially designed for it. Modules may share data on different levels, all of which must be considered when threading the engine. It is possible to thread at a later stage of development if all data bindings are well documented, but a complete redesign may be needed otherwise.

If performance is critical, threading may be a solution. It is not always the case that it will increase performance, so it is important to determine if threading is needed. If the bottleneck is the GPU, threading will most likely decrease the performance due to the overhead data synchronization creates. If the bottleneck is CPU, however, threading the engine will most likely increase performance if data synchronization is limited (Intel Corporation, 2003), (Intel Corporation, 2005), (Intel Corporation, 2005).

11.2 Results

The design of the engine works well for the problem at hand, but for a more complex game where decoupling of Systems is important, a less coupled design would be better. One solution would be to remove the Singleton pattern from Core, thus making it impossible for Systems to get access to other Systems without explicitly allowing them. Decoupling data and classes would make the engine easier to multi-thread but also easier to maintain (McConnel, 2004, ss. 100-102, 142-143).

12 Game design

12.1 Background

Within the scope of this project lies not only the challenge of implementing an able game engine but also the task of putting that engine to work by using it to create a new game. There are a few characteristics of this task which are fundamentally different from software engineering, computer graphics and physics simulation.

There are multiple definitions of what a game is and in addition to that there exist many definitions of gameplay (Rouse, 2004). This implies that defining what is a *good* game or even what *good* gameplay is not trivial.

It has also been suggested that computer game design is an art form (Crawford, 1984) and if games are art it is effortless to imagine the difficulties inherent in game design.

If considered an art form, taking an algorithmic approach from the basis of firm definitions in order to create an exciting game is analogous to taking an algorithmic approach to creating a piece of music. It may produce an adequate result, but it is debatable whether this procedure would bring a new Beethoven's Fifth Symphony.

Nevertheless, game design have been decomposed and analyzed and guidance for

computer game designers does exist (Björk & Holopainen, 2005).

12.2 BASIC GAME IDEA

A single idea is often the basis of a game and this idea is developed into a game concept. The idea may be of various kinds such as style and art, characters, new technology or gameplay (Bates, 2004).

Rather than focusing on theoretical aspects of games and entertainment we have taken. Instead of delving into definitions, and from them derive an optimal game idea, we narrowed the scope to a few handpicked games and briefly analyzed them. Common to these games were that they were either critically acclaimed or personally favored, or a combination of the two. We tried to analyze what makes them superior rather than trying to define what makes an arbitrary game stand out in the scope of all possible games.

Richard Rouse III supports this approach in the book Game Design Theory and Practice and argues that being able to recognize which characteristic is the foundation for a games success is an important skill of a game designer. Rouse also points out that such an analysis brings a more complete understanding of game design.

12.3 CONCEPT DEVELOPMENT

The *high concept* of a game, as described by Bob Bates (Bates, 2004) , is the one or two sentence summary of a game that is the realization of the basic idea. Bates states that many publishers believe that if a game cannot be translated into such a description the game cannot be successful, and he supplements that publishers do have a point in believing this. It needs however be added that this is expressed from a marketer's point of view.

12.3.1 GAMEPLAY

As mentioned above, *gameplay* is a commonly used term but comes with many definitions. Richard Rouse III describes gameplay simply as game interactivity (Rouse, 2004). That is, the way the player gives input to the game and the way the game world responds to these input. In line with the definition of gameplay he defines game design as determining the form of gameplay.

The primary goal of the game design of Megachile Pluto was to create a conflict, a balanced challenge for the player to overcome.

The developed *high concept* was the following: An action/puzzle game where the player collects and delivers a certain types of items or combinations of items to a dying planet. This is logically linked to the environmental imbalance and depletion of resources on the planet Earth, and the player's motivation is to save planets from this fate. The player is presented with all information needed to solve the puzzles in the game. This is to retain a sense of fairness. The challenge lies rather in the player's dexterity and ability to plan ahead. This was designed using guidelines defined by (Bates, 2004, pp. 107-134), for creating balanced game levels and puzzles.

We wanted to present a unique feel to our game by designing the physics around the fact that every interplanetary body within a solar system has its own gravity. This means

that player orientation is more loosely defined than in a game that is designed using a 2D plane. Some practical changes had to be done to sustain playability, such as modifying the gravitational pull from different bodies. Whilst doing this we prioritized not making any inconsistent changes, as well as keeping the world generally consistent with reality in order to not distract the player.

"Consistency is crucial for keeping players immerse in game environments; it's okay to be able to run 100 miles per hour and jump 20 feet—as long as the rest of the world reacts appropriately." (Hecker, 2000)

Our guide mark during the game design sessions was simplicity. If an idea was too complicated for the other team members to grasp immediately, the idea was discarded. Furthermore, an idea which introduced extra buttons or an additional graphical user interface item was considered less favorable. However, simplifying the user interface led to a non-intuitive interaction and more buttons were added later.

12.4 Music and Sound

12.4.1 Background

When LucasFilm Ltd was evaluating the new THX sound standard, it became apparent that good sound can actually fool the brain into thinking that the quality of the picture is better. When a group of people who previously watched a movie was shown the same movie a second time with improved sound quality, they commented that the picture seemed sharper. This implies that good sound not just is able to enhance the overall experience, but can also make other components of a game seem more polished (Simpson, 2000).

12.4.2 XACT

XACT is a tool which ships with the XNA and DirectX SDK:s which is used for integrating audio resources into games. The first step is to load the wave files into the editor. These can then be assigned as Cues, which means that they can be accessed through a unique name from the application. A Cue can either be played back as it is, or can be attached to an Emitter. If a Listener is also specified, XACT automatically uses this information to apply 3D-effects to the Cue. In the XACT tool, information about how XACT should apply 3D effects can be specified.

12.4.2.1 Earcons and Auditory Icons

An earcon is a synthetic sound, which often consists of notes arranged in different structures to symbolize objects and actions. Studies suggest that users can learn to recognize earcons even if their musical ability is not well developed (Dix, Finlay, Abowd, & Beale, 2004). Earcons should generally be short in order to be memorable and are constructed from basic building blocks called *motives.* Different motives can be combined to represent new actions. The most important characteristics of motives are *rhythm*, *pitch*, *timbre*, *register* and *dynamics*. As an example, an earcon for opening a file can be a three note ascending sequence. When destroying a file, the sequence can be played backwards, so that the notes are descending instead (Brewster, 2008).

An auditory icon is also linked to an object or action, but its purpose is to resemble something that the user can relate to from everyday life. A standard example is the waste basket sound when emptying the trash can in Microsoft Windows. An earcon for the same action could for example be a descending note sequence, more underlining the sense of *drop* or *throw* rather than *emptying*.

One of argument for using earcons instead of auditory icons is that there are many computer related actions that has no or little resemblance to actions in real life, thus making it hard to design auditory icons that are suiting that particular action. The arguments against earcons are of course that they are quite abstract, and will not be as intuitive to understand for the user.

12.4.3 RESULTS

12.4.3.1 Music

The music was created using Logic Platinum for PC and mastering was done in WaveLab. Mostly software samplers and software synthesizers were used, but some hardware synthesizers were also utilized. Some own audio recording were also executed, like for example the choirs in one of the songs.

The beginning of the main title music exposes also the main musical theme of the game. This twelve note long sequence appears in various musical cues in the game, although in different disguises. When the player dies, the theme appears in a slower tempo. The first eight notes are identical, although the harmonization is different. The remaining notes of the theme are played in a downward motion instead, although the note value remains intact, thus still being quite similar to the original theme. Being played on a church organ instead of brass, the mood is quite far from that in the main titles. When a level is completed, the theme appears in a higher tempo, but lands on a major chord after a quick upward movement, emphasizing the player's triumph.

The music for the game levels have been composed so that they can be looped easily. The music in the first level of the game starts out with only drums, then builds up, and after that gradually minimize to drums-only again. This makes for a seamless looping, without any explicit cue about exactly where the music is played back from the beginning.

12.4.3.2 Sound Effects

The sounds were mainly put together in Logic Platinum. The trimming down of the wave files and the setup of loop-points were also here done in WaveLab. In order to make unique and interesting sounds, layers of samples were often tweaked and played back together with different effects applied. Some samples were recorded, like for instance the knocking sound when the player character feels that the player has been idle for too long.

We have used a combination of earcons and auditory icons in Megachile Pluto. Although we do not use earcons in their developed sense (i.e. we do not combine basic earcons to represent other actions), we have used some of the basic ideas. For example, we are using ascending sequences for creating items, and descending for destroying and dropping items respectively. Another example of how earcons are used is the

planting sound, which is a looping sound in a certain pitch. When the player plants a second time, the same sound is played, although in a higher pitch.

Since many parts of the game are not related to activities in everyday life, much thought were spent on the sound effects which should give the player the impression that the sounds actually are emitted in the game world.

13 Discussion and Conclusions

13.1 Programming Language

The language used for the game was C#. Due to the fact that C++ is the most common language for developing games at the time this thesis is written, a comparison with C++ is unavoidable.

C# is a managed language that is executed within a virtual machine, named Common Language Runtime (CLR), which, among other things, takes care of memory management. Based on earlier experience, we found that fewer bugs were encountered when using C# than using C or C++.

Most of the time, the managed runtime is a blessing, but we did encounter problems which showed that in some cases a managed environment is not always ideal. For example, the garbage collector on Xbox 360's version of the CLR works differently than on the PC and often creates a stall when executed. This forced us to re-design some parts of the engine to work better with the CLR on the Xbox 360. In C++, for example, you can put classes on the stack, which is much faster than allocating memory on the heap. In C#, on the other hand, a struct that is initialized within a local scope is put on the stack, while classes are always stored on the heap. Instantiating a new class in C# is very fast, but the garbage collector will get slower the more classes that are allocated, especially so on the Xbox 360. This is comparable to the defragmentation of the heap that C++ suffers from.

Some of the positive things about C# is due to Microsoft Visual Studio and the functionally it provides; auto completing properties and functions (IntelliSense), fast compile times, and a good debugger.

13.2 XNA

The use of the XNA framework proved to be very successful, especially for getting a quick start on the project. One major part of game development is the handling of content, and the content pipeline integrated in the XNA framework proved its value here. The pipeline has support for most common texture formats and two model formats (.X and .FBX). They can be treated in the same way by the application programmer, regardless of format.

It is also relatively simple to extend the content pipeline by inheriting the classes processing the files. When we included normal maps into the engine for instance, we did a simple extension which calculated the tangent-space basis for all vertices on the models using this pipeline extension.

In essence, XNA is a wrapper around Direct3D, but some of the time consuming parts of Direct3D has been removed. For

example, one needs to worry less about releasing and reallocating memory for assets, when a reset on the graphics device happens. Every game also needs to have loop which takes care of messages happening in the operating system. This is readily implemented in XNA, similar to the game loop in glut for OpenGL. While implementing a game loop is not very difficult, having it built-in definitely makes for a more efficient start.

13.3 XACT

Using XACT for the sound integration worked very well. There were some minor problems with it, being mostly that new wave files had to be deleted and reloaded into the tool when external changes had been made to them. Not a major issue of course, but sometimes small changes were made to the audio files outside XACT which could be hard to notice. Using the "Rescan Wave Bank Files" should accomplish this (Corporation, 2008), but often it seemed like the operation did not make any changes. This issue might have been solved in a later release of XACT.

13.4 Software Engineering

The most desirable consequence in regards of software engineering came from the rapid software development's iterative approach. During early development, we could continuously release working builds of the game. This added a sense of readiness to deliver the product at anytime as the possibility to regress to the last working release was always possible. This enabled us to continue to polish and add new features without worrying about integrating all different parts successfully at the end, which obviously could have caused problems unless properly planned, had another method been used.

Due to the lack of a software testing group during development, it was not possible to emphasize quality reassurance at the end of product development, which would be based on testing feedback. Because of this, it has been of great importance to evaluate and specify new requirements and prerequisites throughout the process in order to ensure software quality, as specified by the iterative cycle that was used.

During the course of the construction of this game, many requirements changed, removed or reworked. We believe this has led to a higher-quality product, compared to what it would have been if the specification had been frozen at the beginning of development. Testing is a very important and widely used quality-assurance strategy, while prerequisites are often neglected. This case-study shows that desirable results can be achieved when a greater emphasis is placed upon prerequisites.

Another aspect of software engineering, more specifically component-based software engineering was prominent in increasing productivity. The ability to detach and attach specific components during development, and also in the final product, aided the joint team development effort. Incomplete or erroneous parts of the project could be detached by members working on other parts of the project.

14 BIBLIOGRAPHY

(2008). Retrieved April 27, 2008, from Gamespot: http://www.gamespot.com

(2008). Retrieved April 27, 2008, from Intel: http://www.intel.com/multi-core/index.htm

(2008). Retrieved April 27, 2008, from MSDN: http://msdn2.microsoft.com/en-us/library/bb204834(VS.85).aspx

(2008). Retrieved April 27, 2008, from AMD: http://multicore.amd.com

Advanced Micro Devices, Inc. (2008, May 21). *AMD*. Retrieved May 21, 2008, from AMD: http://multicore.amd.com

Akenine-Möller, T., & Haines, E. (2002). *Real-Time Rendering.* Wellesley: A K Peters, Ltd.

Bates, B. (2004). *Game Design, Second Edition.* Course Technology.

Beck, K. (2000). *Extreme Programming Explained.* Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., et al. (n.d.). Retrieved April 26, 2008, from Manifesto for Agile Software Development: http://www.agilemanifesto.org/

Björk, S., & Holopainen, J. (2005). *Patterns in game design.* Hingham: Charles River Media.

Boehm, B., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., et al. (2000). *Software Cost Estimation with Cocomo II.* Addison-Wesley.

Bouvier, D. J. (2002). *From pixels to scene graphs in introductory computer.* Elsevier Science Ltd.

Brabec, S., Annen, T., & Seidel, H.-P. (2002). *Shadow Mapping for Hemispherical and Omnidirectional Light Sources.* Retrieved May 13, 2008, from Max-Planck-Institut für Informatik: http://www.mpi-inf.mpg.de/~tannen/papers/cgi_02.pdf

Brewster, S. (2008, March 31). *Earcon Experiments.* Retrieved May 20, 2008, from The Glasgow Multimodel Interaction Group: http://www.dcs.gla.ac.uk/~stephen/earconexperiment1/earcon_expts_1.shtml

Corporation, M. (2008, March). *Building an XACT Wave Bank*. Retrieved May 17, 2008, from MSDN: http://msdn.microsoft.com/en-us/library/bb172313(VS.85).aspx

Crawford, C. (1984). *The Art of Computer Game Design.* Berkeley: McGraw-Hill/Osborne Media.

Crnkovic, I. (2003). *Component-based Software Engineering - New Challenges in Software.* Univ. Zagreb.

Dix, A., Finlay, J., Abowd, G. D., & Beale, R. (2004). *Human-Computer Interaction.* Essex: Pearson Education Limited.

Euler Integration. (2008). Retrieved April 27, 2008, from Wikipedia: http://en.wikipedia.org/wiki/Euler_integration

Feynman, R. P. (2006). *QED the strange theory of light and matter.* Princeton: Princeton University Press.

Finley, A. (2007, December). Sk Games' Bioshock. *Game Developer* , pp. 20-26.

Gamespot. (2008). *Gamespot*. Retrieved May 14, 2008, from Gamespot: http://www.gamespot.com

Gath, J., & Dreijer, S. (2006). *Derivation of the Tangent Space Matrix.* Retrieved May 14, 2008, from Blacksmith Studios: http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php

Gold, J. (2004). *Object-Oriented Game Development.* Addison Wesley.

Gorder, P. (2007). *Multicore processors for science and engineering.* IEEE Comput. Soc.

Haines, E. (2002). *Real-Time Rendering.* A K Peters, Limited.

Hall, J. (2008). *XNA Game Studio Express: Developing Games for Windows and The XBox 360.* Boston: Thomson Course Technology PTR.

Hecker, C. (2000). *Physics in computer games.* ACM.

Heidrich, W. (2002). *Shadow Mapping and Shadow Volumes: Recent Developments in Real-Time.* Retrieved April 27, 2008, from http://www.nealen.com/projects/ibr/shadows.pdf

History of video games. (2008). Retrieved February 11, 2008, from Wikipedia: http://en.wikipedia.org/wiki/History_of_video_games

Intel Corporation. (2005). *Developing Multithreaded Applications: A Platform Consistent Approach.* Intel Corporation.

Intel Corporation. (2008, May 21). *Intel*. Retrieved May 21, 2008, from Intel: http://www.intel.com/multi-core/index.htm

Intel Corporation. (2008, May 21). *Intel*. Retrieved May 21, 2008, from Intel: http://www.intel.com/multi-core/index.htm

Intel Corporation. (2005). *Multi-threaded Rendering and Physics Simulation.* Intel Corporation.

Intel Corporation. (2003). *Threading Methodology: Principles and Practices.* Intel Corporation.

Irish, D. (2005). *Game Producer's Handbook.* Course Technology, Incorporated.

Joll, A. (2008). *Shadow Mapping.* Retrieved May 14, 2008, from Ziggyware: http://ziggyware.com/readarticle.php?article_id=161

Jones, C. (1998). *Estimating Software Costs.* McGraw-Hill.

Krantz, S. G. (2004). *Differential Equations Demystified.* MacGraw-Hill Professional Publishing.

Latta, L. (2004, July 28). *Building a Million-Particle System*. Retrieved May 14, 2008, from Gamasutra: http://www.gamasutra.com/view/feature/2122/building_a_millionparticle_system.php

Luna, F. D. (2006). *Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach.* Plano, Texas: Wordware Publishing, Inc.

McConnel, S. (2004). *Code Complete, Second Edition.* Microsoft Press.

Microsoft Corporation. (2006, August). *Coding For Multiple Cores on Xbox 360 and Microsoft Windows*. Retrieved May 21, 2008, from MSDN: http://msdn.microsoft.com/en-us/library/bb204834(VS.85).aspx

Microsoft Corporation. (2004, March). *Next Generation of Games Starts With XNA*. Retrieved May 14, 2008, from Microsoft Corporation: https://www.microsoft.com/presspass/press/2004/mar04/03-24xnalaunchpr.mspx

Microsoft XNA. (2008). Retrieved February 11, 2008, from Wikipedia: http://en.wikipedia.org/wiki/Microsoft_XNA

Möller, T., & Trumbore, B. (2000). *Fast, Minimum Storage Ray/Triangle Intersection.* Chalmers University of Technology & Cornell University.

Newton's laws of motion. (2008). Retrieved April 27, 2008, from Wikipedia: http://en.wikipedia.org/wiki/Newton's_laws_of_motion

Ning, J. Q. (1997). *Component-Based Software Engineering (CBSE).* IEEE Comput. Soc. Press.

NVIDIA. (2007, 11 02). *Adobe Photoshop Plug-ins.* Retrieved May 14, 2008, from NVIDIA Developer Zone: http://developer.nvidia.com/object/photoshop_dds_plugins.html

Playstation 3. (2008). Retrieved April 27, 2008, from Wikipedia: http://en.wikipedia.org/wiki/PlayStation_3

Refractive index. (n.d.). Retrieved May 20, 2008, from Encyclopædia Britannica: http://search.eb.com.proxy.lib.chalmers.se/eb/article-9063034

Rouse, R. (2004). *Game design: theory & practice, second edition.* Plano: Wordware Pub.

Runge-Kutta Method. (2008). Retrieved April 27, 2008, from Wolfram Mathworld: http://mathworld.wolfram.com/Runge-KuttaMethod.html

Scanlon, J. (2007). *The Video Game Industry Outlook: $31.6 Billion and Growing*. Retrieved February 11, 2008, from http://www.businessweek.com/innovate/content/aug2007/id20070813_120384.htm

Scharl, J. (2007, 07 28). *Solution for cubemap handedness problem?* Retrieved May 14, 2008, from XNA Creators Club Online: http://forums.xna.com/thread/18309.aspx

Shastry, A. S. (2005, January 18). *Soft-Edged Shadows.* Retrieved May 13, 2008, from GameDev.net: http://www.gamedev.net/reference/articles/article2193.asp

Silicon Graphics Inc. (2008). *OpenGL Overview*. Retrieved May 14, 2008, from Silicon Graphics Inc.: http://www.sgi.com/products/software/opengl/overview.html

Simpson, J. (2000, July 17). *3D Sound in Games*. Retrieved May 15, 2008, from GameDev.net: http://www.gamedev.net/reference/articles/article1130.asp

Sommerville, I. (2007). *Software Engineering, Eighth Edition.* Pearson Education Limited.

St-Laurent, S. (2004). *Shaders for game programmers and artists.* Boston: Thomson Course Technology PTR.

St-Laurent, S. (2005). *The COMPLETE Effect And HLSL Guide.* Redmond, United States of America: Paradoxal Press.

Williams, L. (1978). Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78),vol. 12* , pp. 270–274.