CHALMERS

3D Racing Car Game

Project Lloyd

Bachelor's Thesis

SVEN ABELSSON RUNING
MATHIAS HÄLLMAN
NICOLE ANDERSSON
SVEN ANDERSSON
FREDRIK TOFT
RICKARD NILSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2009

3D Racing Car Game
Project Lloyd

Sven Abelsson Runing
Sven Andersson
Mathias Hällman
Fredrik Toft
Nicole Andersson
Rickard Nilsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Gothenburg
Sweden

Abstract

This bachelor thesis describes a case study, where we focusing on developing a 3D racing car game, using a process based upon *agile development*; an *evolutionary development* method.

The thesis will cover implementation of real-time graphics, physics engine, network support, as well as sound effects and background music.

In the end, our case study will show that this development process was an appropriate choice for our game development project.

Sammanfattning

Denna slutrapport beskriver en fallstudie gjord på utvecklingen av ett 3D-bilspel. Utvecklingen använder sig av en process baserad på *agile development*, som är en metod inom *evolutionary development*.

Slutrapporten kommer att täcka implementering av realtidsrenderad grafik, fysikmotor, nätverksstöd, liksom ljudeffekter och bakgrundsmusik.

I slutändan kommer fallstudien att visa att utvecklingsprocessen var ett lämpligt val för vårt spelutvecklingsprojekt.

Table of Content

1. Introduction

Developing software applications is a time-consuming process, and with time-consuming processes come high costs. During the last years, several software development methodologies, often known as *agile software development*, have become widely used by software developers to address this issue. Many different development methodologies can be more or less good, depending of the task and application type.

One of the software development methodologies is the *evolutionary software method*, which, as the name hints, takes on an evolutionary approach to the problem, and allows the project to evolve through different stages of the project. Our case study will show how well this evolutionary approach worked on our project where we choose to develop a 3D graphic computer game. Some requirements for the computer game were given from the beginning, such as:

3D graphics – The game must contain 3D models, and render these in the game. 3D environments were never a requirement, and platform games with 2D environment could still open up for 3D objects.

Impressive result – The game result must impress whoever plays the game. It should last long, and make the players come back and play it over and over again.

Graphical effects – To achieve an impressive result, we would need to add modern graphical effects, such as real-time rendered soft shadows, motion blur, and ambient occlusion.

Working with these requirements, we decided to use Microsoft XNA as our platform to develop our 3D game with. This decision was made with regard to that the platform had many in-built tools and provided a good framework for us to get started with the development as fast as possible. The fact that Microsoft XNA also used C# as development language was also in consideration, since we wanted to learn this newly developed C-based object-oriented language.

The requirement for the game to contain 3D graphics introduced an interesting challenge for the project group, since all had none or little experience in 3D modelling. Spending time learning how to model proper 3D models for our game was therefore necessary. During the research to find out what 3D modelling program to use, we found that we could use different studios to create models that we could later import to our game project. The complete game contains models made in both Blender and 3D Studio MAX.

With these choices made, we soon had our development environment set to use Microsoft Visual Studio 2008 with Microsoft XNA Game Studio 3.0 supporting the framework, and Blender and 3D Studio MAX for modelling the graphical components. For some of the sound effects we also made use of Adobe Audition 2.0.

2. Game Design

2.1. Game Design and Concepts

In this project, we were left free to decide what type of game we wanted to develop. The suggestion was that a racing game would be suitable, since such a game usually do not depend on advanced assets, e.g. animated models. After some brainstorming, it was decided that a racing game should be developed. However, there were two different racing game ideas, which will be described below.

In compliance with our development process, the game concept evolved, as more and more features were added. To further explain how the game concept evolved, the development has been divided into three parts.

2.1.1. Initial Concept

Our first concept had a game play similar to Remedy's game in 1996, called *Death Rally*[1], where each player had a car equipped with a weapon, and the goal was to hunt down the opponent's cars. The other idea was to create a simple race track where players could collect coins in order to win. Since the first idea involved much greater work than the latter one, it was decided that the first idea were to be given a low priority, whereas the second were to be given a high priority.

2.1.2. Second Iteration

We decided that a joint solution would be best, where players are able to collect coins and then use the money to make upgrades. Example of upgrades was better weapons, faster engine, and heavier car body. These upgrades would be similar to the concept of levelling, something that many players appreciate. It was also decided that the racing track should be a garage or a store-house, filled with containers, boxes, and miscellaneous objects suitable for that environment. Since driving a real car around a garage is virtually impossible, it was decided that we should make remote controlled cars instead.

2.1.3. Final Concept

We kept most of the ideas from the second iteration with one exception for the death rally concept. It appeared that our first instinct, to give this idea a low priority, was right and in this last iteration, the idea was dropped. However, a racing game where players only may collect coins and nothing more, sounded a bit boring. Influenced by games like Super Mario Kart, it was decided that power-ups should be added. There are a number of different power-ups, and they could be divided into two groups; those that affect the car that hit the power-up, and those that affect the opponents' cars. A more profound description of the different power-ups follows.

In addition to the other improvements, one change was made to the points system. It should be possible to collect three types of coins, worth 50, 100 and 200 points.

Power-up	Affects opponents' cars	Description
Nitro		The player's car moves faster.
Punch	★	A punch from a random direction hits the opponents' cars.
Double points		The player's points are doubled.
Slow	★	Slows down the opponents.
Reverse	★	The opponents' steering controls are reversed.
Low Friction	★	Causes the car to lose friction to the ground.

Figure 1: Punch object.

2.2. Development Process

In a software development project, the resulting product is required to fulfil many different qualities. Examples of such quality requirements are: robustness, availability, maintainability, dependability and usability. To meet such varying demands, it is important to base the work on a well prepared strategy. In software engineering, the term for such a strategy is commonly known as *software process,* which is built on one or several *software process models*.

What Is a Software Process Model?

A software process model is a theoretical philosophy that describes the best way of developing software. Based on one or several models, a *software process* is formed providing guidance on how to operate. A software process model may also be described as an abstract representation of a software process. The concept of the process model is similar to an abstract java class, which can not be instantiated, but it can be implemented by another class, thus providing basic guidelines for that other class.

A model may for example demand customer involvement, but it does not state exactly how. A process implementing that model should involve the customer in the process' activities, but is free to choose how[2].

There is not only one type of process model, but two. The first one is the most common, and described above. The second type of process model is called a *process paradigm*, which is a model even more general than an ordinary process model. Such a model does not hold any details on how the activities that lead to the completion of a project should be performed, but what it does hold is basic guidelines of how to develop software, and assump-

tions about what sort of project could benefit from implementing a particular model. With this in regard, one can conclude that a process paradigm provides a framework that may be adapted to form a process which suits a particular project.

There are three major process paradigms that are commonly used today in software engineering practice; the waterfall model, component-based software engineering and evolutionary development[2].

The Waterfall Model

The waterfall model is recommended for large and complex systems that have a long life-time[2]. Some systems which carry these attributes are also *critical systems*. This means that if a fault would occur, it may result in:

- Threat to human life (death or injury)
- Economic loss
- Environmental damage[2]

It is believed that the waterfall model would be an appropriate choice when developing a critical system, since the model emphasises on thoroughness[3].

The basic concept is to take all the activities and treat them separately. One activity is always followed by another, in the same way water travels down some falls. This description becomes even more obvious when looking at a visualization of the model (Figure 2).

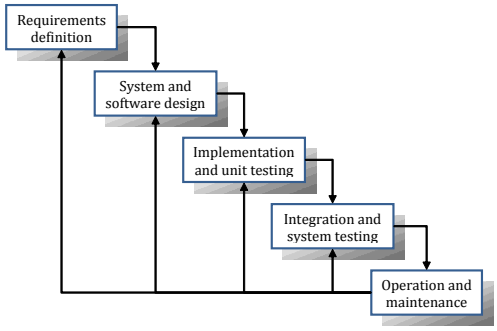


Figure 2: The waterfall model[2].

1. *Requirements definition* All requirements on the system are found by talking to system users. Example of requirements can be services, constraints and goals, such as "We want a webpage that colour-blind people can enjoy".
2. *System and software design* In this activity, the overall architecture of the system is established.
3. *Implementation and unit testing* The software is implemented in units which also are tested.
4. *Integration and system testing* The units are merged together into a complete system. Further testing is required.
5. *Operation and maintenance* The system is delivered to the customer and put into operation. "Bugs" are almost always found, and therefore the system required bug-fixing and maintenance.

In each of the activities described above, one or several documents are produced. The idea is not to start on a new activity until the documents belonging to the previous activity is signed off, but in practice, this is not how it is done. Instead most of the activities overlap and all the documents feed information to the different activities. Although these documents provide a good overlook,

they are very costly to produce. Due to the cost of constantly producing, signing and reworking documents, the waterfall model should not be applied to a project where one is uncertain about the requirements[2].

Component-based Software Engineering

As the name suggest, this model is used when there are several already existing components which can be reused and put together to form a system. Normally, reuse is an important part of any software process, but in this case it is the fundamental activity. The first and the last activities, requirements specification and system validation, are the same as in any process model, but the intermediate activities differ from other models.

During the component analysis, one searches for components that correspond to the requirements. When the right components are found, they might differ a bit from the original requirement specification, thus leading to a modification of the requirements. Before the system is developed and the components are integrated, the architecture of the system must be established. This is done in consideration to the found components. When all of the components are fully integrated, it is time for validation[2].

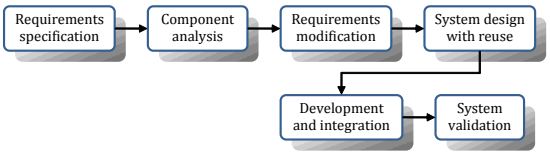


Figure 3: Component-based software engineering.

Evolutionary Development

This model is recommended for small and medium-sized systems that shall be developed rapidly[2]. The idea is that specification, implementation and validation should work as concurrent threads, all supplying input to the other activities. Customer input is considered to be very important, if not even necessary for the development and outcome of the project.

At the beginning of a project, an initial implementation of the software is produced and released to the customer. This way, the customer can make improvements and further describe details, which might not have been clear to the developers. Gradually the system will improve, and when the final release comes, the customer should be fully satisfied with the system.

Evolutionary development is divided into two major types:

1. *Exploratory development* The development start with the requirements that are best understood. To further evolve the system, new features are proposed by the customer. This idea relies on the customer's willingness to spend much time contributing to the project.
2. *Throwaway prototyping* The development starts with the requirements that are poorly understood, because there is no need to prototype requirements that you already understand.

2.2.1. Our Development Process

In this project, evolutionary development was chosen as a keystone for our

development process. The decision was mainly based on the size and the duration of the project. We had a limited amount of time, namely 4.5 months, and during this time we wanted a rapid development. Component-based software engineering was easily dismissed since there were very few, or none, components to work with. The waterfall model was dismissed, since it focuses a lot on writing documents, which is a very time-consuming activity.

When it came to deciding whether to go with throwaway prototyping or exploratory development, the latter was chosen. This decision was based on the following facts. The greatest problem was not to understand the basic requirements, but how much time it would take implementing the many possible features. Given the circumstances, the best strategy was first to implement the game foundation, and subsequently implement one feature after another, hence using exploratory development.

As indicated earlier, a process is seldom based on one single model, and models are not considered to be mutually exclusive.

Since evolutionary development is a process paradigm that provides little detail on how to work, it would be wise to choose a little more specific model. This model would have to agree with the ideas that evolutionary development represents. There are two major process models which would be suitable; *XP* (eXtreme Programming) and *agile development* methods[2]. How these two models are connected to evolutionary development, and which level of detail they hold, is represented by Figure 4.

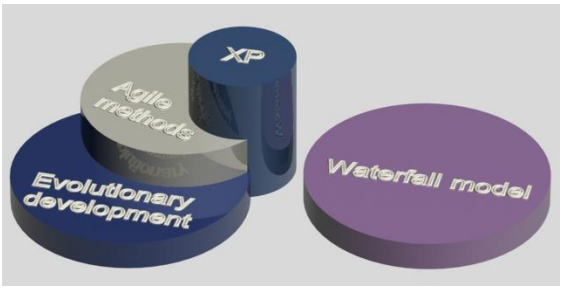


Figure 4: Visual description of how different process models are related to each other.

For example, agile methods agree with evolutionary development on some points, but other ideas and concepts may be added, hence the agile circle is not totally encased by the evolutionary circle. The heights of the circles indicate what level of detail the models hold. XP is more detailed than agile, which is more detailed than evolutionary.

Since we, in this project, wanted to test something other than XP, it was decided that agile methods would be suitable.

Agile Methods

Agile is a common name for a number of different software process models. Although the different models share the same idea, that the best way of developing software is throughout incremental development and delivery, they propose different ways to attain this. However, they have one more thing in common, namely a set of principles, which are described below[2]:

- *Customer involvement*

The customer should be closely involved in the development process and contribute by providing new system requirements and evaluate each of the iterations. In this project the developers along with the supervisor acted as customers.

- *Incremental delivery*

  The customer specifies what requirements are to be included in what increment.

- *People, not process*

  It is important that the members of the development team are free to develop their own way of working, instead of blindly following what the process specifies.

- *Embrace change*

  The system should be designed in such a way, that it is easy to implement changes or new features.

- *Maintain simplicity*

  Both development process and software should be kept simple[2].

Milestones

As indicated earlier, incremental delivery is a keystone in agile development, which also makes it an important part in this project. Each increment is here called a *milestone*, as a way of indicating how far on the road to a complete program we have come. Since evolutionary development stresses the importance of having short increments, it was decided that each milestone would last approximately two weeks. In order to get a good overview and make reasonable time estimations we, acting as developers, had two meetings per week. On these meetings, we checked the project's status and saw to it that the milestones would be delivered on time. In the beginning of each milestone, we acted as customers, thus providing new features to be added. What features to add, was based on an evaluation of the most recent milestone. For more information about what tasks that were assigned to what milestones, see appendix A.

In addition to the milestones, a rough time estimation of the major parts of the game was made, this is found in appendix B.

2.3. Game Engine

2.3.1. Game Framework

To save time in our development process, we choose to use Microsoft XNA framework, when developing our game.

Microsoft XNA

Microsoft XNA framework provides different tools to help game developers get started more quickly[4]. Such tools, as the XNA Framework Content Pipeline, help developers by reducing the time you need to add 2D and 3D content to your game[5]. In our game, we used Microsoft XNA Game Studio 3.0, which contains a complete class library for handling game content and events[6]. We took advantage of many of these classes; Audio to implement sound effects and background music to the game, Graphics to implement 3D graphics to the game, Input to

communicate with the player, and Net to implement network support to the game.

CodePlex

CodePlex[7] is a community to host open source projects; most of them made in .NET, and especially C#. The website is hosted by Microsoft, and was started to encourage open source development. CodePlex provides different tools to support the user in his or her development; such as wiki pages, discussion forum, RSS support and statistics (over downloads and visitors). It also provides the project with a version control system, which has support for different software, including TortoiseSVN[8], and Visual Studio Team Explorer, which we used in our project. A version control system was absolutely necessary for our project group to manage different versions of the project files, especially when many developers work with the project concurrently.

2.3.2. Physics Engine

Background

The part that handles the physical behaviour of the game world is called a physics engine and it is made up of three main parts; collision detection, a dynamics simulation and updating the system[9]. The dynamic simulation part solves the forces affecting the simulated object.

It is desirable that the game objects behave in a realistic way. The objects should be able to detect when they collide with other objects. When they hit another object, they should resolve the collision in a realistic way. Their speed should be affected by gravity. Each game object is represented in the physics engine by a simplified mesh, called a collision skin. The collision skins are often rough estimations of the geometry of the game object in order to make the simulation faster. The chassis of our car objects are for example represented by two boxes (Figure 5), one large for the bottom of the model and a smaller for the top.

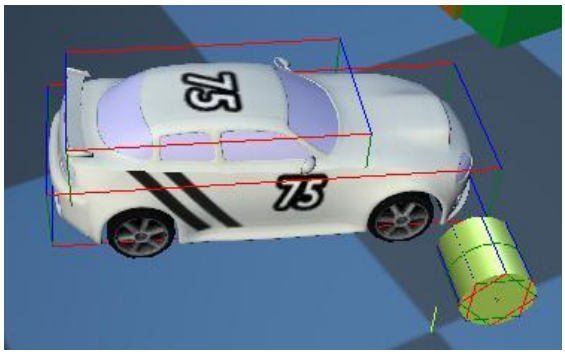


Figure 5: The car is represented by two boxes in the physics engine.

Rigid body physics is the part of physics that deals with the motion of rigid bodies. Rigid body simulations are fast and easy to calculate relative to other methods of simulating physics. This makes it very useful in game development where performance plays an important role when trying to create a realistic playing experience.

Rigid body physics can simulate movement and rotation in three dimensions and allows a body to have properties like mass, friction, torque, velocity etc. In a rigid body physics simulation, all objects are non-deformable. This means that an object can never be squeezed or bent regardless of what forces are applied to them.

JigLibX

We chose to use an open source physics engine, called JigLibX, which uses rigid body physics to simulate the game world[10]. The main reason why we choose to use an already existing physics engine, as opposed to creating our own, was to get a good simulation of physics for relatively little work. JigLibX also has a car class that was considered very useful in our game.

Collision Primitives

Collision testing is the most demanding part of the physics engine, taking up 90% of the CPU time[11]. In order to improve the performance of the collision test, collision primitives are used.

The shape of the objects being tested for collision, greatly affect the speed of the collision test[12]. By using simplified shapes, collision testing can be done more effectively.

In JigLibX, each game object has a collision skin that is built up by one or more collision primitives. The basic shapes are boxes, spheres, capsules (a cylinder with round ends, like a pill), planes and rays. There are also more advanced shapes such as height maps and triangle meshes. If one wants parts of an object to move separately, like the links in a chain, one can use joints to connect the different collision primitives (Figure 6).

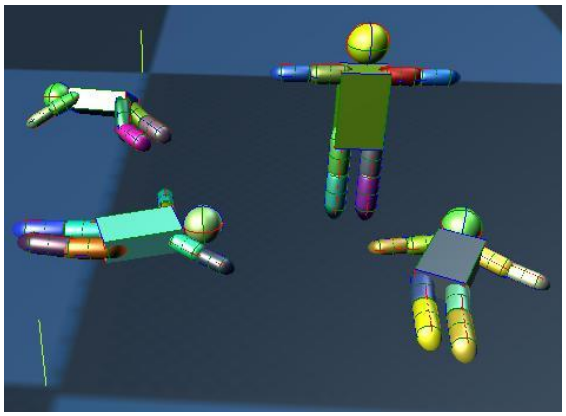


Figure 6: A sphere, box and capsules connected by joints to simulate a human body in JigLibX.

Flow of Control

In order to set up physics simulation in JigLibX, one needs to create a physics system instance that controls the movement of the bodies. Then, it is possible to add a body for each of the game objects that typically has a collision skin with one or more collision primitives attached to it.

When the physics system is setup, one can “advance time” and let the physics system simulate what the world will look like after a small amount of time has passed. In JigLibX, this is called integrating the physics system.

After the integration, one can extract the transformation for each body and apply it to the models of the game objects.

Limitations

Rounding Errors

Generally, games prefer fast simulations over accurate simulations. This is because an accurate simulation is slow and causes low frame rates, which in turn create a poor playing experience.

However, the speed does come at a cost in the form of rounding errors that can cause unrealistic behaviour in the game. When two objects are close, for example a box lying on a floor, the rounding errors can cause the position of the box to be slightly inside the floor. This causes a collision in the physics engine and the box may start to shake uncontrollably or even explode[13].

Some physics engines, like the 3D world of Second Life, solves this problem and also saves processing power by not update resting objects. The physics engine can freeze objects that have not been exposed to any forces within the last two second. The object remains frozen until it is affected by a collision with an active object. This may, however, create other undesired behaviour like freezing a ball that is thrown up in the air when it slows down and start to fall down.

Primitive Collision Skins

In order to speed up the simulation, the physics engine approximate the shape of more complex objects with primitive collision skins, referred to as the collision geometry, like boxes and spheres. If you take a tea cup that is represented by a complex mesh, it could be represented by a cylinder in the physics engine (Figure 7). This makes it impossible to fire a bullet through the handle of the tea cup, and since the cylinder typically is larger that the cup, it would detect collisions that actually should not occur. When the cup is dropped on the floor, it would not roll around on the floor like a cup but as a cylinder.

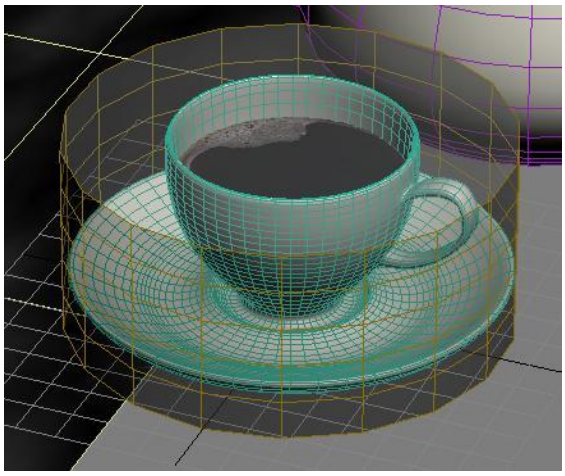


Figure 7: The cup is represented by cylinder in the physics engine.

Low Frame Rates

Low frame rates causes the physics engine to take larger time steps for each step of the simulation. Objects that are moving do not move smoothly, but appear to teleport from one point to the next each frame. At a high enough speed, a bullet could teleport past its target and miss it, if the target is thin enough to fit in the distance that the bullet is teleported between each frame (Figure 8). Even though the frame rate is improved, there is always a chance a collision is not detected[12].

In Second Life, this problem is resolved by adding a long invisible shaft to the bullet. The shaft trails behind the bullet, so that as the bullet teleports forward, the shaft is long enough to cover the gap between successive teleports of the bullet and collide with any object that might fit between the calculated frames[14].

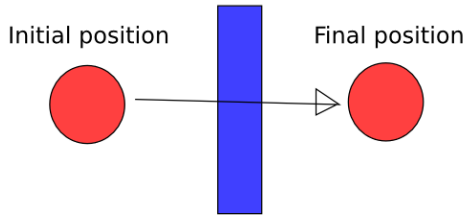


Figure 8: When the distance the object is moved between two frames is greater than the width of a wall, the object can teleport through it.

2.3.4. Game Assets

2.3.4.1. Creation

Models

Almost all of the models in the game have been created by us. Instead of hunting the web for free models that would suit our needs, we decided to create the models ourselves. This meant we had a very good flexibility when deciding what kind of objects we wanted to incorporate in the game.

The standard Content Pipeline importers and content processors available in XNA Game Studio, support various common art-asset file formats[15]. The model formats supported by default are .x and .fbx[15]. Support for these two formats was a key factor when deciding what modelling tool to use, since implementing support for a different type of format would increase our work load.

Modelling Tools

There are a lot of modelling programs on the market, such as 3ds Max[16], Blender[17], Maya[18] and Zbrush[19], to mention a few. Prices for these tools differ a lot. Some are free to use, and others are rather expensive. The programs that seemed to suit our needs best, and hence the programs that were chosen for our modelling purposes, was Blender and 3D Studio MAX. This since they both support exporting to the fbx format[20], [16]. There are also a lot of information and Internet tutorials[21] available, which also played an important role in our decision.

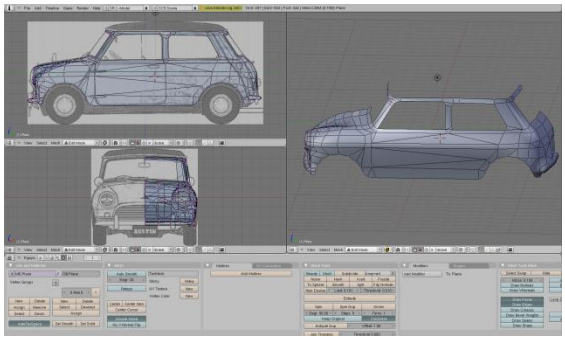


Figure 9: Creating a car model in Blender, using reference images to do one half of the car. Then copy and mirror it, and "sew" the two parts together.

Texturing

Applying textures to simple flat objects, such as a big wall or floor, is fairly simple. One can select a texture, and simply let it repeat itself as many times as needed, to cover the whole surface[22].

Texturing a more complex model, for example a car, is somewhat more complicated, but can be accomplished by the use of projector functions, or mesh unwrapping algorithms[22]. In Blender, one can mark "seems" in the model and then use the built in unwrapping algorithm, which maps each of the models vertices to specific texture coordinates (Figure 10).

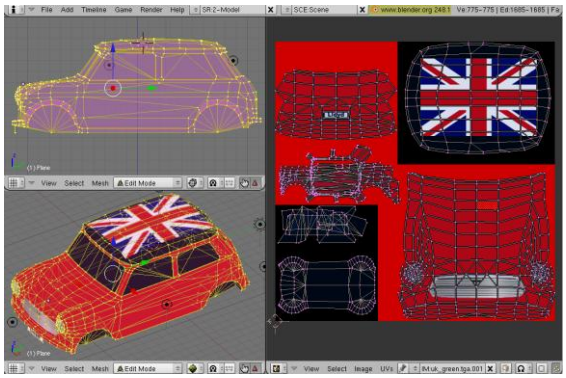


Figure 10: The car model cut to smaller sections and mapped to a texture image.

The normal map (section 3.3.2) is mapped onto the model the same way as the diffuse texture; hence the normal map can be created directly from the diffuse texture. In our game, the normal maps were created using a tool called CrazyBump[23], which lets the user create a normal map from a diffuse texture.

2.3.4.2. Result and Discussion

By choosing to create the game models ourselves we had the advantage of not being dependent on finding good models in the right format that were also free to use. It also made it easier to do changes or doing additions to the models, since we had all the necessary files and the knowledge about the modelling tool.

On the other hand we had to spend more time on creating models and learning how to use Blender.

Our textures, or part of the textures, are mostly obtained from various free resources, such as CG Textures[24], and then modified to suit our needs. The normal map and specular map were created from the diffuse texture with the program CrazyBump. A downside using this program for the creation of the normal map, instead of using a very detailed (high polygon count) model, is that the result will depend on how good the program recognizes the "shape" of the object in the diffuse texture. The best method would be to create a very high resolution model and then extract the normals directly from this model, as opposed to letting CrazyBump guess the shape of the surface. However, creating high resolution models is a very time consuming task, and would take a lot of time for us to fully master. Although CrazyBump may not produce an exactly accurate result, it is accurate enough, and the normal maps created enhance the realism in the game a great deal, hence it was used for creating all of the normal maps for the game.

2.3.4.3. Loading

The XNA framework uses an asset loader, called the Content Pipeline, which has been extended in our game to properly load models and textures.

This asset loader is run when the game is built in Visual Studio. It loads the model and texture files, converts them into a format usable by the XNA framework, and then outputs them so that they can be loaded again efficiently at run-time[25].

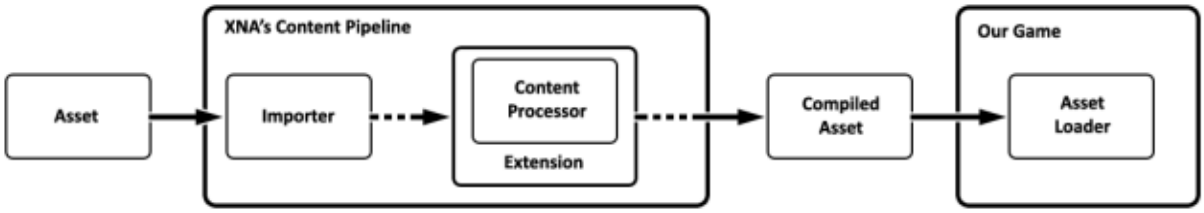


Figure 11: An asset is converted by XNA's Conent Pipeline to an intermediate file format, which in turn is efficiently loaded at run-time.

In the basic asset loader, all models are associated with the basic effect provided by the XNA framework. Since we needed our models to use our own custom effects, an extension of the basic asset loader was necessary[25].

When a model is loaded, its textures are loaded as well. The file format we have used for exporting models is not fully supported by the software we have used for modelling, and the same applies for XNA's basic asset loader of this file format[17, 25]. Because of this, our extension of the basic asset loader also includes the loading and association of additional textures to a model. In particular, a specular and a normal map are loaded in addition to the regular diffuse texture.

When a texture of any kind is used, it may be the case that the same computations will be done for all of its texels. As an optimization, these computations could be done at build-time. That way, when a texel is looked up, those computations do not have to be carried out at run-time. Similarly, we interrupt the loading of the diffuse and specular textures to do some of the normalization of the Bidirectional Reflection Distribution Function (BRDF) mentioned later (section 2.2). We also interrupt the loading of the normal maps, although this is done to transform the normal held at each texel. This is necessary, because the file format that we use to store our normal maps may only hold positive values, while the normal maps needs to contain both positive and negative numbers if it should be able to represent normal perturbations in all directions in 3D-space[26].

2.3.5. GUI, Menu and HUD

Background

Almost every game available has some sort of methods to communicate information to the player, for example score, game map, and most commonly some kind of menu system. Regarding head-up-display (HUD) information, such as score and game maps, usually this is just rendered as sprites to the screen as a last step in the rendering pipeline.

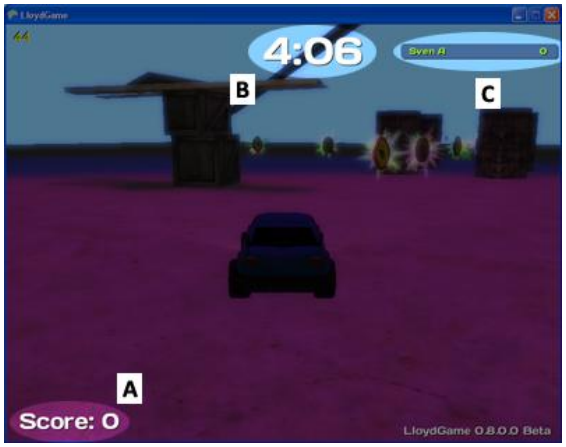


Figure 12: A typical view from our game, with GUI components such as score (A), round timer (B) and player list (C) highlighted.

Results

Menu

When we started implementing the multi-player part of our engine, we needed some way to present the player with a useful user interface. We looked at a couple of different menu systems available in tutorial form from the community website for XNA[27].

We also tried implementing these tutorials directly in our existing game, but found that this would need many low level changes to our engine, so this was scrapped quite early. Instead, we created our own very basic menu system, keeping some of the ideas from the previous implementation.

Our menu system built on something we called GameScreens, where each screen took care of input, responding to input, menu animations/transitions, and drawing itself.

Each GameScreen can have zero or more GameScreenComponents, the most basic type of GUI widget in our case. The menu hierarchy is built in such way that each GameScreen can hold another GameScreen, which is called a sub screen. For example, when the user interacts with a menu widget that should open another screen, a new GameScreen is initiated as sub screen to the active GameScreen. When a new sub screen is created, the initiator/owner of this sub screen stops its own rendering loop, and instead calls the sub screens render method. To go back to a previous screen, the active screen is just deleted. When this is done, the owner of the newly deleted screen continues its original render loop.

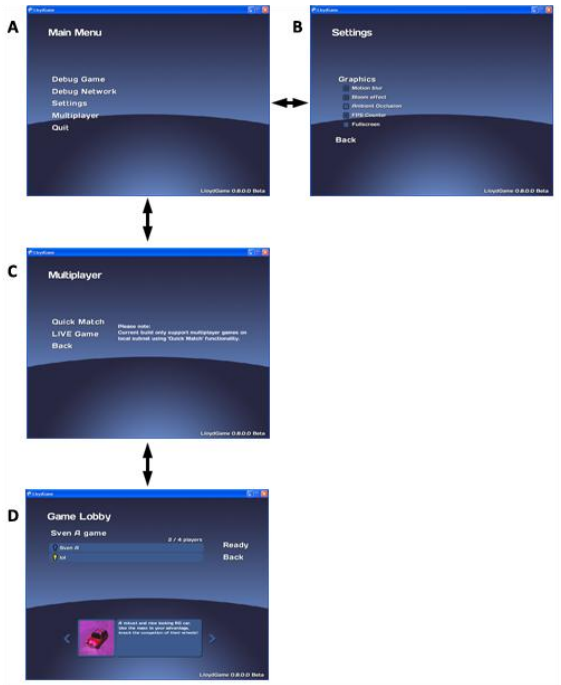


Figure 13: Our final menu system, the arrows show the flow of navigation. (A) Main Menu screen, (B) Settings screen, (C) Multiplayer selection screen and (D) Lobby and car selection screen.

HUD

As soon as we had added collectible coins/points to the game, we needed a way to inform the player his current score. This, along with most of the HUD, was rendered text to the screen. To render the text, we used a feature in XNA Game Studio called SpriteFonts. When compiling the game, XNA loads the specific TrueType described in the each SpriteFont-file, and from this creates a textures with all the needed characters[28].

For other GUI components, such as backgrounds and icons, we load them as external textures and simply render them as sprites.

Special case: ScreenFlashes

We found that when a player picked up either a coin or a power-up, it often was not sufficient to just play a sound

to inform what had just happened. To solve this, we added a component we named *ScreenFlashes*, which could display a texture on the screen for a limited amount of time, with a couple of different animation sequences. These were then used to display a picture with the name of the power-up or coin value that was picked up, with an animation sequence that zoomed and faded out the image. ScreenFlashes were also used to display the countdown sequence when the game started.

Figure 14: A ScreenFlash being rendered as a result of a remote player picking up the "Reverse" power-up.

Discussion

Thanks to our menu system, implementation turned out to be very simple. It was relatively easy to add/remove screens, and their behaviour late in the game development. One of the last things we added was the Settings-screen, which needed a new type of check box widget for each available setting. Adding the screen itself, as well as the new widget type, was a straight forward implementation. We conclude that our implementation was very well suited for the development process we used, which allows major changes to the game and code late in the development.

3. Graphics

In accordance with the evolutionary software development model, we wanted to postpone as much work related to graphics as possible, until we had come so far in the development that it was time to put focus on graphics. The XNA framework provided us with the major part of the basic functionality to render our game[29]. This allowed us to render our game with minimal allocation of developer time before we got started on the graphics of the game. Although, to give our game its own look, our usage of this basic functionality later had to be exchanged for more advanced alternatives where we had more control over how the game was being rendered.

3.1. Lights and Rendering

To capture the very complex behaviour of light in the real world so that a computer may simulate it, relatively simple models are needed. One extensively used idea is to break up light into three components; ambient, diffuse and specular. The ambient component is the colour of the light that has bounced on the surroundings numerous times before reaching the destination point whose colour is currently being computed. The diffuse component is the colour of the light that contributes to a surface's base colour. Finally, the specular component is the colour of the reflection of light on a surface[33].

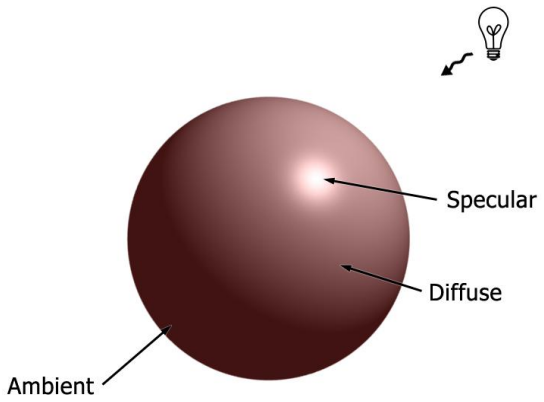


Figure 15: The different lightning components contributing to the final image.

Since it makes little sense to have a light source with different colours for different material properties, only one colour is normally used for all of light source's components. However, the material of a surface is modelled in part by these three components as well. The diffuse component of a material is lit by the diffuse component of a light. Similar reasoning holds for the specular and ambient components[33].

Another essential part of modelling the behaviour of real light is how it is affected when it comes in contact with a surface. One can not assume that the light bounces off with the same angle as the incident angle. More care needs to be taken when dealing with computer graphics. Of course each individual photon cannot possibly be traced through the entire scene in real-time with the computational power of a modern computer, and actually no light is traced in any way by a regular rasterizing renderer[22]. Instead, when a fragment of a surface is being rendered, all of the incident light is assumed to come directly from the light source (except for the ambient light), and the amount of outgoing

light in all directions from the fragment is determined by a BRDF. The BRDF is supposed to capture the surface's way of scattering the incident light, since this one point that is being rendered actually represents a small area that may not be completely flat, which would be a requirement for the light to simply bounce off the surface with the same angle as the incident angle[30].

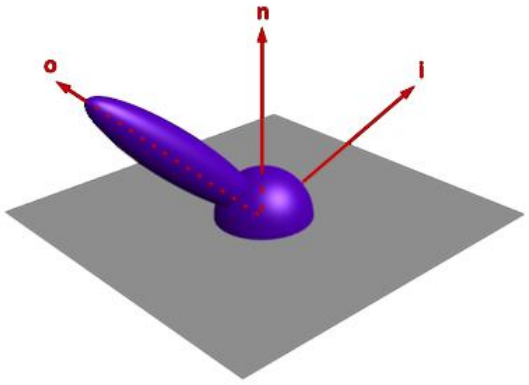


Figure 16: This BRDF lobe shows how incident light can scatter non-uniformly in all directions.

The computations involved in this model of real light behaviour are done in a separate small program called a shader, and it is run on the graphics hardware. To allow several shaders to reside in one file, called an effect file, shaders are grouped in passes, which in turn are grouped in techniques[31]. The XNA framework provides an object called an effect, which aggregates an effect file and the basic operations one would like to perform on it, such as assignment of parameters and activation of individual passes for upcoming rendering[32].

The first step towards using our own effects was to implement our own light sources that we could place in the game world. In the rendering process, we also needed to properly setup the effects used by the model. With this in place, we needed to create the actual effect files used by our models.

The game has only one effect file, which is used by all models. The corresponding effect object is used in a similar manner for all models. This means that all models in the game are rendered as if the materials they were made of were all the same. Only by giving the effect a different set of parameters is it possible to make two surfaces look as if they were made of two distinct materials. For giving a surface its own look, the textures handed to the effect is the most important of these parameters.

The effect file consists of a technique containing three passes. The first is the ambient pass, and the other two are the point light pass and spotlight pass. The latter two are essentially the same, differing only in the type of light source used. All of the passes use Phong shading, which means that the colour is computed per pixel[33]. An alternative to this, giving reasonable visual results for most cases, would be to compute the colour per vertex. A triangle's three vertices would then have its own colour, and then these colours would be interpolated over the triangle's surface. This is called Gouraud shading[34].

3.1.1. Ambient Pass

The ambient pass renders the scene with only the ambient light

contribution. After it has been run, the depth buffer holds the final depth at each pixel. This allows the following passes to save a lot of computations by discarding all fragments but the one that is actually seen in the final image[35].

The ambient pass also uses a second and third rendering target. These targets record the final depth and the surface normal of the rendered fragments. After the ambient pass has been run, the results from the rendering targets are stored as textures, for later use by the post-processing effects.

3.1.2. Light Pass

For any model, the light pass is run for each light source that lits the model. Each light source's contribution is added onto the previous light sources' contribution or the ambient light's contribution if it is the first light being used. Depending on the light source used, either the point light pass, or the spot light pass, is used.

The light passes use a normalized version of the Blinn-Phong BRDF. This particular BRDF is the one used by the OpenGL and Direct3D fixed function pipelines, though unnormalized in these cases[36, 37]. It was designed specifically to produce credible visual results, as opposed to many other BRDF's, which has been designed to model reality[38]. Because of this, the original Blinn-Phong BRDF does not preserve energy. This could become evident when adjusting the parameters to make a surface look more shiny. The specular light would be focused on a smaller portion of the surface, but it would not become any brighter. The normalization simply adjusts the BRDF to preserve energy, so that light becomes brighter when focused[22].

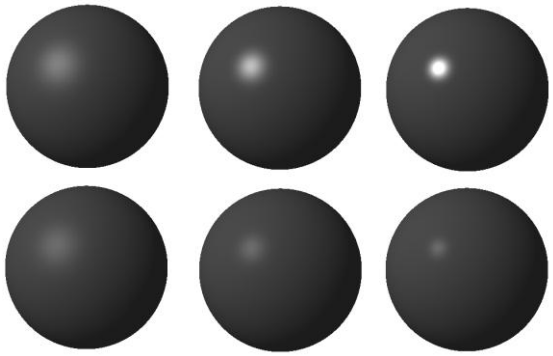


Figure 17: The upper row shows a ball being rendered with different shininess, using the normalized Blinn-Phong BRDF. The lower row shows the same ball being rendered with the same different shininess', using the unnormalized Blinn-Phong BRDF.

The diffuse and specular components are computed according to this BRDF. Then, the normalization takes place, and finally the result is multiplied by the lights intensity. A specular map is also used to determine how much the specularity a surface has at each point.

The light intensity is computed differently depending on the light source used. A point light is omni-directional, meaning that only the distance to the light is needed. When the distance to the light is within a falloff start, full intensity is used. When it is beyond a falloff end, the light is completely blacked out. If the distance is between the falloff start and falloff end, an interpolation determines how much intensity the light has. Spotlights use the same kind of falloff, but it also has a direction and a cut-off angle. This cut-off is computed in much the same way as the falloff, only that it is the angle between the spotlights direction

and the fragments position relative to the spotlights position that is used instead of the distance. Finally, the computed light intensity is multiplied by the shadowing factor.

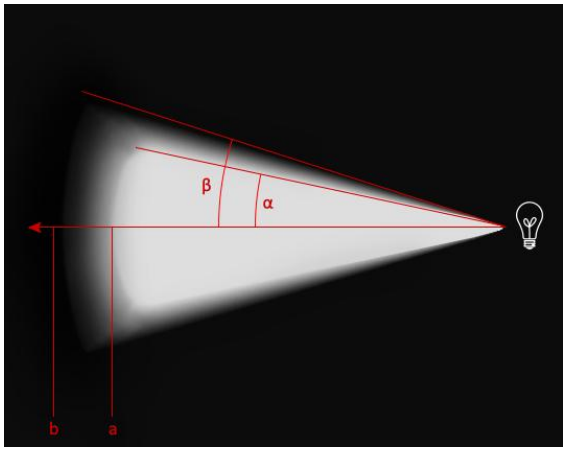


Figure 18: Full light intensity is used up to point a, it then falls off until point b is reached, where the light intensity becomes nonexistent. Likewise, full light intensity is used up to angle α, it then falls off until the angle reaches β, where the light intensity becomes nonexistent.

3.2. Shadows

Shadows are highly important for a good sense of realism. If objects that occlude the light would not cast shadows onto other objects in a scene, an observer is likely to quickly pick up on that something is wrong[39]. Variance Shadow Mapping[40] was selected for computing the shadows in our game.

3.2.1 Shadow Mapping

Shadow Mapping is a technique commonly used in games to cast shadows. The basic idea is to record how far the light from a specific source reaches, and then compare this to the length between the light source and the actual fragment being rendered. More specifically, one starts with rendering the scene from the view of the light source, but for each pixel, a depth value is stored instead of a colour. The texture produced is called a depth map, or more commonly, shadow map. This means that for any point in the scene, a depth has been recorded in the direction from the light source approximately to that point, and this depth represents how far the light reaches towards the point before something stops it. For any point that is closer or just as close to the light source as the recorded depth, the point must be in light. If on the other hand, the point is further away from the light source than the recorded depth, this must mean that some other object is occluding the light, and this point must be in shadow[41].

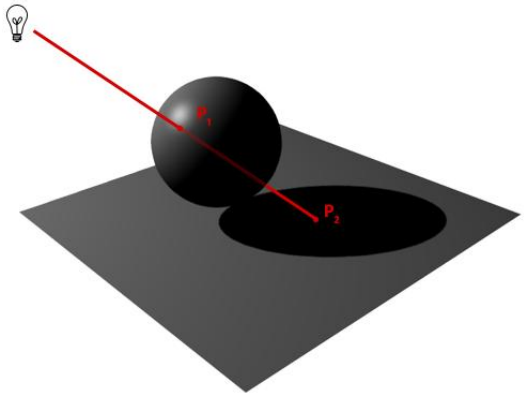


Figure 19: Point P_2 is being rendered, and the depth recorded in the direction from the light source to P_2 is looked up. The depth in this case is the distance from the light source to the point P_1, meaning that the light stops there and doesn't reach P_2.

Since the depth map only has a finite set of texels, several close by points which may have different actual distances to a light source will use the same depth value from the depth map. This may cause self-shadowing artifacts on lit surfaces. To prevent this, a small bias is added to the depths of the depth map. This lets more points be closer to the light than the recorded

depth in the points' approximate direction, and therefore putting the point in light[42].

An arguably more significant drawback of regular shadow mapping is that a point is either considered in shadow, or not. This produces what is called hard shadows, which looks less realistic than the alternative, called soft shadows. For soft shadows, there is a middle ground, such as 10% in shadow, which may be the case for a point in the outer edge of a shadows penumbra[39].

3.2.2. Variance Shadow Mapping

Variance Shadow Mapping is an improvement on the regular Shadow Mapping technique; it was designed specifically to yield soft shadows. The ideas behind it are founded in probability theory. In particular, it calculates the probability that a point should be in shadow, and based on this, a certain amount of shadow is cast on that point. Two things that differs variance shadow mapping from regular shadow mapping is that the variance shadow map does not need any bias, and it may be blurred. In fact, blurring is required in order to get the soft shadows[40].

A nice feature of the soft shadows produced by Variance Shadow Mapping is that the spread of the shadows penumbra is relative to the distance between the receiver of a shadow and the light source. That is, if a pillar is used as an occluder for a light source placed right behind it, the penumbras spread would be quite small at the root of the pillar, but the further away from the light source you look, the penumbra would become larger, and eventually the entire shadow would fade away[40].

Figure 20: This image shows how the penumbra becomes wider at a greater distance from the light source, positioned right behind the stack of boxes.

Although, Variance Shadow Mapping has its drawbacks as well. Perhaps the most prevalent of these would be light leakage. Light leakage occurs when an object casts a shadow within the region of a larger shadow. The result is that the edges of the smaller shadow lights up and become visible, even though they should be occluded by the larger shadow[40].

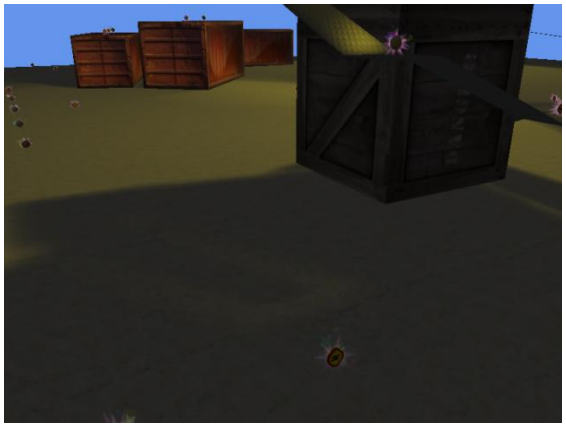


Figure 21: This image shows light leakage in our game.

In our implementation of variance shadow mapping, each light source has its own shadow map that is

recomputed whenever necessary. Creating the shadow map can be costly, although measures can be taken to keep these costs down. One way to do this is by not recreating parts of the shadow map that is certain not to have changed. For instance, a game consists of a lot of objects that does not move. If the light source has not moved either, these static objects contribution to the shadow map could be reused. For that reason, our implementation has an internal separate shadow map for static objects that is only recreated when necessary. Dynamic objects on the other hand could move from frame to frame. These needs to be taken into consideration in each frame to create the new shadow map.

3.3. Bump Mapping

The bump mapping technique was introduced by Blinn in 1978[43]. It is used to describe small details on a surface that would be too complex to render efficiently if it was represented by polygons. This is done by using a texture which contains pixel level information on how the normal differ from the surface's normal. Knowing how the normal differs, one can then account for it in the lightning calculation. This will darken or highlight small areas on the surface, giving the illusion that the surface has bumps in it.

3.3.1. Techniques

These methods are commonly referred to as bump mapping techniques: Blinn's original methods, Normal mapping and Parallax mapping.

One of the original methods developed by Blinn is to use a height field texture[22]. This means that each texel in the texture represents a height. By taking the difference between adjacent texels the slopes can be determined[44].

3.3.2. Normal Mapping

For modern graphics cards, it is preferred to use normal mapping[22]. The normal map texture stores the direction of the normal in the RGB-values of the texels. Normal mapping is preferred, since storing three components versus storing a height map, or two offsets as in Blinn's original method, is no longer considered too memory consuming[22]. These methods all produce identical results, but using normal mapping will reduce the number of computations per pixel needed when calculating the shading[22].

3.3.3. Parallax Mapping

Parallax mapping[45, 46] is a more refined method than normal mapping and Blinn's methods. The idea is that the position of bumps moves relative to each other as the view changes, and when you move around and look from different direction the bumps should occlude each other.

3.3.4. Result and Discussion

The technique chosen for our game was normal mapping. This since we were not short of memory, but more in need of fast and efficient rendering. There were also a lot of information and examples to be found that demonstrated this technique. Implementing this method for our game was fairly straight forward. The lightning calculations for a point light and a directional light were already implemented, but they only used the

normal for the surface as a whole. For the implementation of normal mapping we needed to use the information about how the normals were orientated at each pixel, obtained by sampling our normal map texture. The final orientation of the normals were then calculated in the vertex shader and sent as an input to the lighting fragment shader.

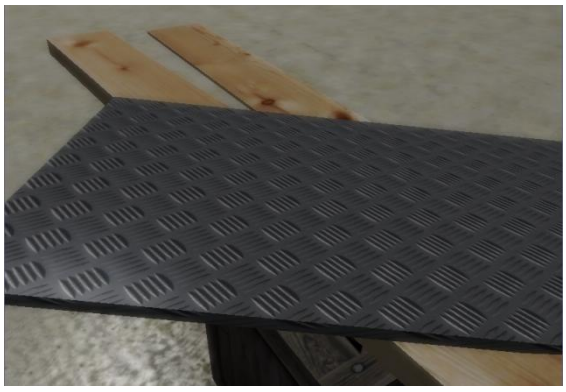


Figure 22: A scene from the game without normal mapping.

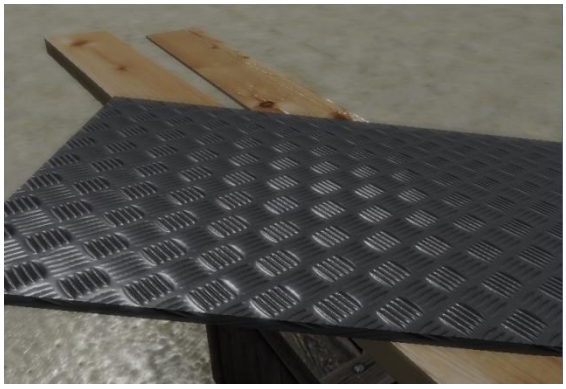


Figure 23: The same scene with normal mapping.

An unforeseen problem occurred when trying to implement this technique. It was a result of using Blender and the fbx-format for the models in the game. When exporting from Blender we could link only one texture for each model, which meant that we had to find another way to get all types of textures loaded; colour, normal and specular textures (section 2.3.4.2).

3.4. Post-processing Effects

Post-processing effects are effects that are achieved by processing the image after it has been rendered, hence after referred to as the original image. This is done by first rendering the entire scene to textures, three of them in our case, instead of the screen. The three textures contain information about colour, depth and surface normals.

When the scene has been rendered to the textures, one post-processing effect is run at a time. For each effect, the appropriate textures and other parameters are assigned to it, and then a quadrilateral covering the full image is drawn. In this way, the textures are essentially being stretched over the screen, so that each texel matches up with each of the screens pixels. This allows the fragment shader to process the rendered image at each texel, using its colours and perhaps depth and surface normals.

3.4.1. Motion Blur

Motion Blur is meant to simulate speed. The faster an object travels, the more we want to blur it. This effect could be implemented in a variety of ways. We chose to implement it in screen space as a post-processing effect because this is an easy way to integrate motion blur into an already existing rendering process. Had we chosen to implement it in another way, we may have been forced to modify a lot of our code[47].

In addition to the colour texture, the motion blur effect will also need the

depth texture. The inverse view matrix, along with the depth and the view direction at each pixel, is used to compute the view space position of the rendered fragment, that is, where in front of the camera the rendered fragment is located. The previous frame's view-projection matrix is then used to compute where this position was projected onto the screen in the last frame. The velocity of the pixel is then computed with the screen space position of the fragment, both in the current and in the previous frame. The actual blurring is then carried out by sampling the colour texture along this velocity vector, starting in the current screen space position of the fragment[47].

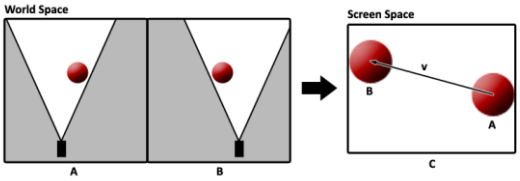


Figure 24: (A) represents the first frame, the cameras orientation makes the ball project to the right half of the screen. (B) represents the second frame, where the camera has moved, making the ball project to the left half of the screen. (C) shows how the ball has moved in screen space, and how a velocity vector is computed by this.

Since we want our car to look crisp, we use a map to mask it out in order to avoid applying blurring to the texels containing the car. The map is held in the alpha channel of the colour texture, and it consists of simply zero if the texel may be used in blurring, or one if it may not.

Figure 25: This is a comparison of a frame in our game with and without motion blur. The right half of this image uses motion blur, while the left half does not.

3.4.2. Screen-Space Ambient Occlusion

Ambient occlusion is an effect that is hard to account for by the regular shading models that are used for real-time rendering. The ambient light is not supposed to be as strong in cramped areas such as corners or cracks as it is on wide open surfaces. The effect called ambient occlusion strives to darken these places, where the ambient light would have a hard time reaching it[48].

Taking ambient occlusion into account in screen space with a post-processing effect is a relatively simple way of implementing it. It also has an acceptable efficiency, making it a good candidate for real-time rendering. Lastly, it gives more than adequate visual results for gaming purposes if implemented properly[49].

As any post-processing effect, the processing is done per pixel of the original image. At each pixel, the view space position of the rendered fragment is computed, much like in the motion blur effect. Next, a number of

samples in 3D-space located in a sphere with its centre at this view space position are considered. Each sample is projected onto the screen and the depth at the projected position is looked up in the depth texture and compared to the samples actual depth. These comparisons of the samples' depths around the rendered fragment's view space position is used to determine how much ambient light the fragment should be exposed to. If a lot of the samples' actual depths are larger than the looked up depths, it is likely that the rendered fragment is located in a corner or something similar. On the other hand, if a lot of the samples' actual depths are smaller than the sampled depths, it is likely that the rendered fragment is located on an edge[49].

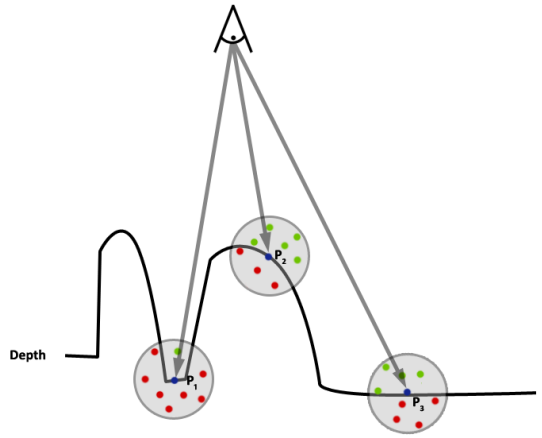


Figure 26: The figure shows a 2D-representation of three points that has been rendered, and their corresponding samples. Red samples represent occluded samples. Green samples represent exposed samples.

Our ambient occlusion effect is achieved in three passes. In the first pass, we determine how much ambient light each pixel is supposed to be exposed to, as described above. To get good results, it is important to not only have a sample pattern with a good distribution of the samples in the sphere, but also to have this sample pattern different for each pixel[49]. Although we only use one sample pattern, we rotate this pattern for each pixel by reflecting each sample around a normal that is taken arbitrarily from a separate texture holding a large number of normals. This results in different sample patterns for each pixel in a non-predictable way.

Since the samples are distributed over a sphere, half of the samples are expected to be located behind the scene geometry on a plane surface. These samples could only be used to determine how much more exposed to light the area is, compared to a flat surface. As we are only interested in darkening the image in cramped areas, and not make it brighter in exposed areas, these samples could just as well have been omitted. Hence, it would be optimal to only sample in the hemisphere above the surface[49]. We do this by looking at the angle between the surface normal, which is looked up from the surface normal texture, and the sample's position relative to the spheres centre. If this angle is larger than 90 degrees, the sample position is reflected so that it comes up on the hemisphere on the right side of the surface.

The texture containing occlusion factors from the first pass contains a lot of noise. This noise occurs because only a few samples in the sphere are taken. A weighting of the samples that are determined to be occluded, based on the samples distance to the sphere centre, helps to reduce noise, although

further measures needs to be taken[49]. In the second pass, a smart blurring of the occlusion factors is carried out. The blurring is supposed to smooth out the aforementioned noise, although this blurring cannot naively blur like all adjacent texels are the same, since that may cause occluded areas to bleed into very exposed areas. These transitions from exposed to occluded must still be sharp in the blurred texture. The smart blurring is carried out by comparing the normal at the texel to be blurred with the normals at the surrounding texels to blur with. If these normals are not very similar, odds are that they are from two distinct surfaces and their occlusion factors should not be blurred.

In the third and final pass, each texel of the colour texture is multiplied by the corresponding texel of the texture holding the occlusion factors. A texel with a high amount of occlusion has a smaller value, why the color at that texel will become darkened.

Figure 27: This is a comparison of a frame in our game with and without ambient occlusion. The right half of this image uses ambient occlusion, while the left half does not.

3.4.3. Bloom

A Bloom filter is used to accentuate bright areas of the screen by making the bright areas even brighter while also letting the brighter areas leak into the darker areas. This is supposed to improve realism by mimicking the human eyes' way of naturally accentuating brighter areas. The eye cannot do this on its own; since a computer screen can not display colours with a high enough light intensity[50].

We achieve this effect in three passes. In the first pass, a bright pass filter is applied to the original image. This will effectively make the dark areas even darker while leaving the brighter areas as they were. This bright passed image has only half the resolution of the original image, both horizontally and vertically. Some performance is gained by allowing it to have a smaller resolution than the original image, since this result in less texels to process and store. In effect, this will also result in some free blurring of the bright passed image[50].

In the second pass, a Gaussian blur is applied to the bright passed image. This is where the bright areas start to leak into the darker areas.

In the third and final pass, the blurred bright passed image is added to the original image. The bright passed image is stretched over the original image, so that each texel of the bright passed image covers four texels in the original image. That is how we get some blurring for free. While adding a bright passed texel with a texel from the original image, some additional

computations are done to adjust saturation and intensity depending on the parameters given to the effect.

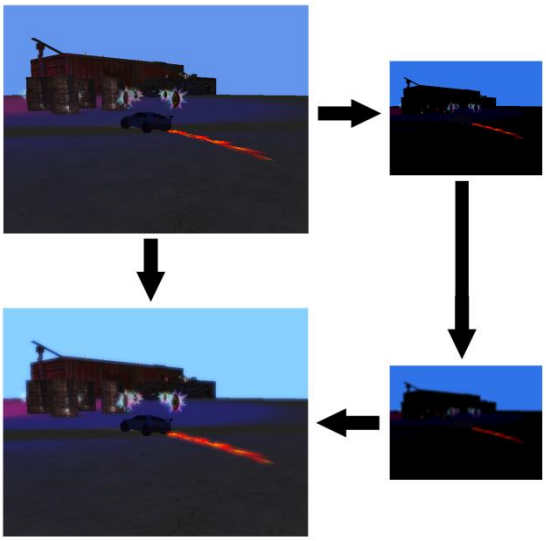


Figure 28: The rendered image is bright passed and down sampled, blurred, and then added onto the original image to produce the bloom filtered image.

Figure 29: This is a comparison of a frame in our game with and without a bloom filter. The left half of this image uses a bloom filter, while the right half does not.

3.5. Particle Systems

Particle systems are used to represent a variety of different objects, groups of objects, or general effects or phenomena that have no fixed form, such as water, clouds, thunder and fire. What they all should have in common in their physical representation is that they should consist of a large amount of individual particles, each of which may move around in the world with or without any regard to the other particles' placement or movement. Each particle may also be drawn with or without any regard to the other particles in the system. The basic operations that a particle system ought to handle is the spawning of particles, moving them around in the world, drawing them, and if necessary declare them dead or respawn them[51].

Memory bandwidth may be an issue, for instance when updating the particles of a particle system. By storing the particles in an array, they can be accessed sequentially to maximize cache hits. It may also be a good idea to do all the work for each particle in one iteration through the array. This means that as the particles are being updated, they must also be setup for fast rendering. A particle normally only has one certain position, but it may represent something larger like a small cloud. Setting up a particle for rendering then means that a shape has to be put up in the world, and the shape's vertices must be computed and stored for efficient rendering. The shapes of the particles are normally flat and screen aligned. When all the alive particles has been setup for rendering, they may all be rendered in only one rendering call to the graphics hardware, instead of the alternative to iterate through all particles again and issue a separate rendering call for each particle[51].

The simplest kinds of particle systems are those whose particles stay completely unaffected by their

surrounding environment, including the other particles in the system. That is, each particle moves around in the world and gets drawn with no respect to any other particle. This is how our particle systems work. Each particle has a position, velocity, energy, size and a few system specific attributes. When a particle is updated, its position is updated with respect to its velocity. The velocity is recomputed for each frame according to the systems defined movement for all of its particles. This allows the particle systems to have all of its particles moving individually, yet very similar to each other. The particles energy may also be decremented, to shorten its life span. The size is used only to scale the shape of the particle when it is setup for rendering.

Our particle systems may also be drawn with a technique called Soft Particles. With regular ”hard” particles, the shapes of the particles are simply drawn where they appear and whatever colour it has is used for a pixel if the particle is in front of any other geometry at that pixel. A shape may be only partially visible however, with part of the shape cutting into already drawn geometry. This could cause artifacts for some types of particle systems, since visible edges of individual particles may not be desired. With soft particles, the colour at the pixels drawn for individual particles may be faded out if it is in front of, and close to other geometry. This is achieved by handing a texture containing the depth of the drawn scene to the particle system’s draw method. For each fragment drawn by the particle system, its screen space position is calculated and the depth is looked up at the pixel where the fragment is to be drawn. This depth is then compared to the fragment’s depth, and a level of fading may be applied[52].

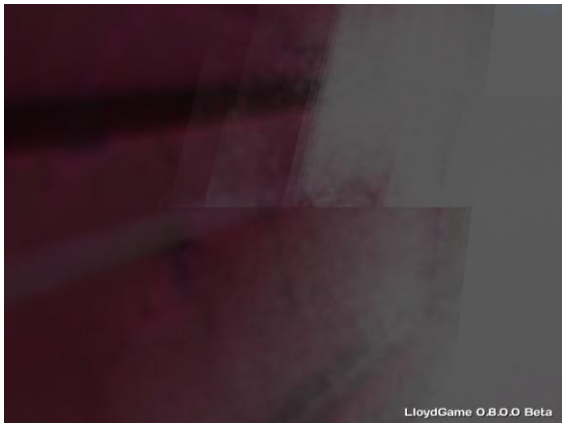


Figure 30: This is a comparison of a particle system in our game with and without soft particle rendering. The bottom half of this image uses soft particles, while the top half does not.

3.5.1. Sparks

Whenever the chassis of the car in our game hits something, several sparks are emitted with a velocity proportional to the speed of the car. A random vector is also added to the sparks velocity to get a nice spread of the particles. The particles movement is very simple, since they need only have a gravitational pull added onto the velocity for each frame. However, setting up the shape of a spark is somewhat trickier, since it would not look very convincing to just put up a quadrilateral of some sort where the particle is positioned. Instead, several previous positions needs to be stored, and a quadrilateral must be stretched from the current position to one of the previous positions. To make sure that the quadrilateral becomes fully visible, it has to be set up correctly. This is

done by disregarding the depths of the view space positions while using the positions to compute the screen aligned direction of a particle. With this direction, the vertices of a proper screen aligned quadrilateral can be computed.

Figure 31: This image shows the spark particle system in our game.

3.5.2. Sprinkles and Glow

Sprinkles, as we call it here, is a non-physical effect to bring some additional attention towards the power-ups in our game. In particular, brightly coloured stars are emitted around the power-ups. The stars then rise in a spiral while spinning and pulsating in size, until they finally fade away.

Glow is very much like sprinkles, only that it stays put where the power-up is located. As with the sprinkles, it spins and pulsates in size, but it never fades away.

Figure 32: This image shows the sprinkles and glow particle systems in our game.

3.5.3. Smoke

While we do not have anything in our game that would realistically cause any smoke, smoke can be a nice feature all on its own. Our smoke particle system simply spawns all of its particles in the centre of the system with a velocity randomly chosen over a distribution such that a flattened hemisphere of smoke appears around the systems centre. The particles' velocity is never altered, though their colours' alpha channel is gradually decreased when the particle starts to reach the hemisphere's outer bound. When the particle has fully faded away, it is respawned in the centre again.

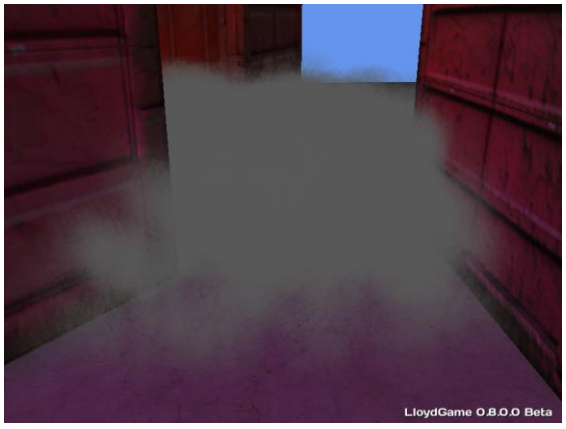


Figure 33: This image shows the smoke particle system in our game.

3.6. Result

The XNA framework made it easy for us to render our game at an early phase of the development. Because of the simple usage of the XNA framework, it was very straightforward to modify the rendering process later on in the development when it was time to focus on the graphics.

Since we knew even from the beginning that we wanted to have post-processing effects, we expected that we would have to use other render targets than the screen. Despite this, we decided to keep things simple by not immersing ourselves with these details until it was needed, to keep compliance with the evolutionary software development model. This change of render target also turned out to be very straight forward, as the XNA framework supplied us with most of the functionality needed for such a change.

Shadow Mapping was also easily integrated into the already existing rendering process, although point lights require several shadow maps to be created to record the depth in all directions, as opposed to spotlights which has a defined direction. This turned out to be very costly, and since we did not want to spend too much time on optimizations, point lights became deprecated. If we would have had a more extensive plan to begin with, as the evolutionary software development model specifically told us not to have, all the time we spent on implementing point lights could have been put to better use.

In large, the evolutionary software development model posed little difficulties when implementing the graphical part of our game. If anything, not thinking ahead too much allowed us to quickly go from one thing to another. This let us see what worked and what did not, often in an early phase, so that we could adapt our plans according to the circumstances and spend time where it was needed.

3.7. Discussion

Since we did not plan much from the beginning, and time was a limited resource, we spent time where it was needed in the present without worrying too much about the future, yet trying to proceed to the next task as fast as possible. This also meant that whenever we had to choose from several alternative methods, we mostly picked an alternative and stuck with it.

We consider most of the choices we made to have been good ones, although one thing we might have done different is to use another shadowing technique. Variance Shadow Mapping turned out to be quite costly, and we consider the soft shadows to be a bit too soft. Often in real life, shadows have an almost hard look. It also caused a lot of shadows to even fade away, long before they would have done so in a real life scene.

If time would have allowed us, there are a few features we would have liked to implement. One feature would be more, distinct shaders, allowing different surfaces to be rendered in different ways. This, we think would have given the game a much more interesting look. For instance, we would have liked to see reflections in the cars, which we could have gotten by rendering the cars with a shader supporting environment maps. Another feature would be more optimization, something which we did not look into very much, and as a result, the FPS was undesirably low. More details on the performance of our game can be found in Section 6.

4. Network

4.1. Background

Our original game design would need at least two people playing the game simultaneously, since we decided to not include any single player capability. Generally, there are two possible solutions that are considered when designing multiplayer. The first solution is a single-system multiplayer functionality, and the second is a networking multiplayer functionality.

Single-system multiplayer uses only one machine for simulation, where the players control the game using one or more input devices. In a Networking-multiplayer game, each player runs a version of the simulation on separate machines, while important game specific information is sent via a network topology.

One network topology for game networking is *Client-Server*, which has one server machine, and one or more client machines.

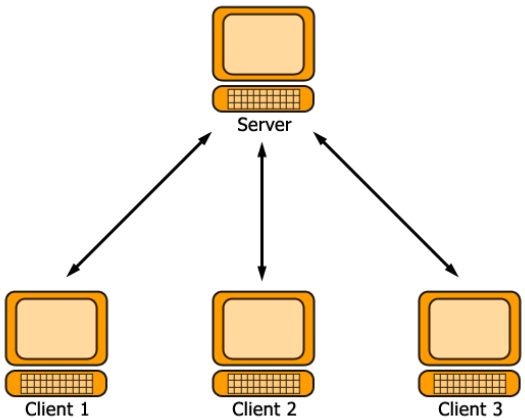


Figure 34: A typical Client-Server architecture, the arrows show the line of communication.

In a Client-Server approach, each client sends its game-action/-data to a designated server, which in return propagates the data to the rest of the clients. Depending on the implementation, servers can act either as a dedicated server, and just propagate the data, or as a combined server and game simulation. Another possible network topology for game networking is a *Peer-to-Peer* structure. In contrast to Client-Server, Peer-to-Peer lacks a dedicated server; instead every client is considered both client and server.

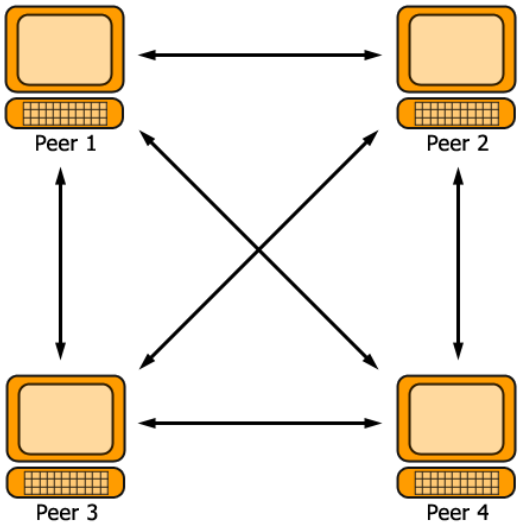


Figure 35: A Peer-to-Peer network, the arrows show the line of communication.

Each peer is connected to the other peers, instead of only being connected to the server, as in a Client-Server topology, and the data is sent to each peer. This means that a Peer-to-Peer structure is less vulnerable, and less likely, to break down on connection errors/problems, while if the server in a Client-Server structure looses connection, no client would be able to continue communicating[53].

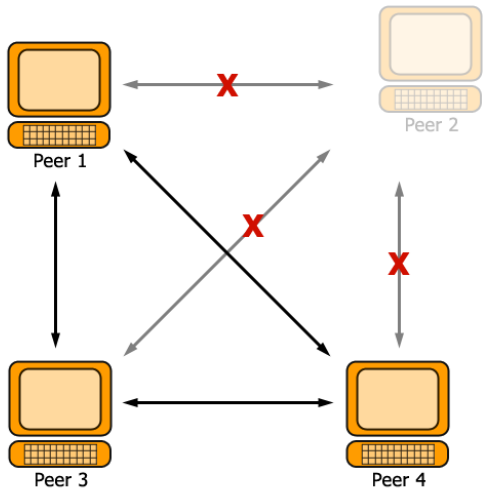


Figure 36: A Peer-to-Peer network with a disconnected peer (Peer 2). The communication will however continue between Peer 1, Peer 3 and Peer 4.

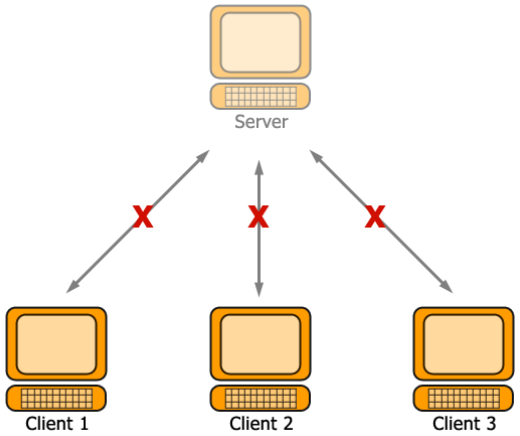


Figure 37: A Client-Server network. Here the server is removed, and no further communication is possible.

Implementing either of these, or any other topology, can either be done from scratch via the network interface provided by the language, or via an external network library. Some network libraries may even come equipped with different topology support, like Peer-to-Peer in the Lidgren-networking-library[54].

4.2. Results

Early in the development, we established the need for multiplayer support in the game, but very little research into different alternatives was made. Midway into development, when the most ground work on the engine was done, we started looking into two attractive options, Lidgren-nework-library and the standard XNA networking interface.

XNA Game Studio 3.0 provided a network interface that could be used on both Microsoft Windows and Microsoft Xbox360 machines with none to minor code changes. While this was a very tempting alternative, it also had its share of limitations. A couple of things to consider when networking via XNA Game Studio 3.0:

- **LIVE accounts**

  Windows Live is a collection of services provided and developed by Microsoft, which include instant messaging, photo and movie sharing[55] for example. To access these services, a Windows Live account is needed. Likewise, using the built-in network methods in XNA also require the usage of Windows Live accounts, even for local networking. This means that as soon as we would want to do anything over the network, the user/developer would need to be logged into a Windows Live account. In a sense, this is a positive feature, since we would not need to handle player name/profile creation, as this is automatically handled via Live accounts. But for small games that do not need any

account management, and in a debugging view point, it might result being unnecessary.

- **No Internet connected games on Microsoft Windows**

There was one difference in the network capabilities via the XNA framework on Windows to the Xbox360 version. On Xbox360, there was both the possibility to initiate an Internet or system link (a game only available on the local subnet) game, while on Windows only the later was available. To be able to do networking via the Internet on Windows machines, usage of the underlying System.Net-classes would be needed. This would however in return not work on Xbox360 machines[56].

- **Voice Chat**

A feature we discovered in the final days of development was that Windows Live-accounts support Voice-chat by default. This was a pleasant surprise, as it worked without any modifications to our code. It is reasonable to assume that Voice-chat is more interesting to those players with a large geographical distance between each other. Considering this, it might not be of much use on a local subnet game. And as described earlier, it is not possible to connect games over the Internet on Windows with XNA, so Voice-Chat in XNA games on Windows might not make much sense[57].

An alternative network library compatible with C#/XNA is *Lidgren networking library*. This is contrary to the built-in interface that XNA provides, in such way that it does not require any part of XNA to work. Another difference is that it is not possible to utilize networking on Xbox360 machines via Lidgren networking library. The Lidgren library also lacks account management, in contrast to XNA usage of Windows Live accounts; it does however provide methods to send serialized typical XNA data.

To make a decision regarding which library we would use in our game engine, we surveyed how we wanted to synchronize the physics simulations and player data between the clients. The only data that would really need to be synchronized was the players' cars, while letting each client have its own physics simulation. This could of course result in different physics representation on each individual simulation, but it is reasonable to assume the difference would be minimal and would not really impact the game play. We, however, wanted one simulation to act as a primary one, in case we wanted to synchronize the simulation data in the future. This primary simulation could either be chosen in some predetermined way in a Peer-to-Peer topology, or for example just act as the server in a Client-Server topology. In our game, we simply went with the later choice.

During the development phase, we had access to two Windows machines, connected to the same subnet network. This meant that while developing, the

network tests would never take place over the Internet, only via the local subnet. Knowing this, we choose to use the XNA networking interface.

Our first network implementation sent the following data:

- **Spawn point data**

  Spawn point is 3d information specifying where the players are able to start when the game is initiated. These points are saved in the level file. When the game is started, each client loads the map data, but does not parse the spawn point data. The server, on the other hand, reads the spawn point data, and then sends one spawn point for each client, to every connected client.

- **Player data**

  When the game has started, for each game update, the players send the position, orientation (rotation) and velocity of their car to the server. When the server receives this information from a client, it starts by updating its local re-presentation of the client's car, and then forwards the information to all the other clients. And likewise, when the clients receive player data from the server, they update their local representation of the player cars with this information.

One problem with multiplayer games on connections with high latency is that it can result in jittering and jumpy/freezing behaviour on data shared over the network. In a racing game, the car could appear to jump from different positions between frames, because the position data of the car is taking to long to travel on the network. This however, is solved by interpolating and taking qualified guesses of the position at each frame, by taking into account the positions from previous frames.

In our game, since each player runs its own simulation, and we also know velocity beside position and orientation, we could automatically get a simple interpolation by letting the physics engine take care of it for us. We do this by just letting each remote player be represented in the physics engine by a car, and updating its position and velocity. This means, if data for a specific player has not been received in a long time, the car's position is just simulated in the physics engine depending on previously received position and velocity data.

This implementation worked out smoothly and very straight forward, and from what we could tell, the interpolation via the physics engine worked surprisingly well.

In the last steps of the development, we needed to add some more game play data to be sent over the network.

Among these were:

- **Car selection**

  Each player should be able to pick the type of car they wanted to control. Such selection was added to the lobby screen, before each game round. When a new car is selected, the new car type is sent to the server,

which saves the car selection to the corresponding player. When the game begins, the server then sends out the correct car type together with each spawn point.

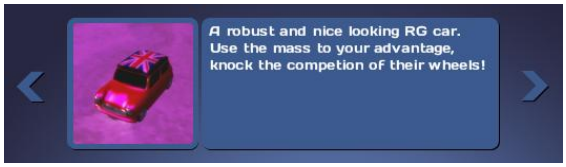


Figure 38: The car selection interface.

- **Game round timer**

Until late in the development, we did not have any specific type of game round. More precisely, the game did not have an end. To solve this, we added a timer that when reaching zero, ended the game and returned the gamer to the game lobby screen. We accomplished this by letting each client/server have a timer each, but that was only started when the server sent a special message. The timers on the client side filled only a graphical purpose, while the timer on the server was the only one that could end the game. This was done in such way to avoid sending to much unnecessary information over the network to just synchronize the clock.

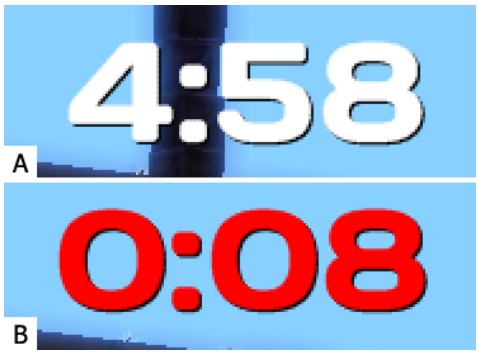


Figure 39: (A) shows the round timer which has just started. The round timer turns from white to red when the time left is below 10 seconds, as shown in (B).

- **Round start count down**

We needed a way to show the player that the game had started. Earlier, the player jumped directly from the lobby screen into the game. This was solved by letting the server send four special messages to all the clients, as soon as the game world was loaded. The messages contained a "count down" number, starting at 3, down to 0, with a second interval. For each "count down" message received on the clients, a sprite was rendered for the number and a sound clip was played. For the final number (zero), the sprite rendered was simply the text "GO!".

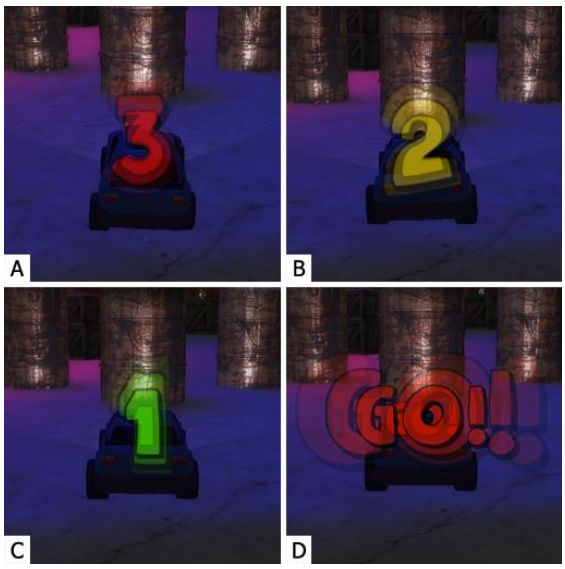


Figure 40: (A), (B), (C): The count down sequence as shown to the player. (D): When the count down reaches zero, the text "GO!" is shown instead.

4.3. Discussion

By using the built-in networking capabilities in XNA Game Studio 3.0, we were forced to limit ourselves to local subnet games. In our case this was not a problem, but in a larger game development, with the need of Internet multiplayer, this would have to be taken into consideration when deciding network library.

However, if we had changed our game design late in the development, in a way that would need games to be able to connect over the Internet, this would result in major changes to our code. This is because the multiplayer component in our game is heavily affected by the network library. For instance, the lobby-screen relies much on specific features provided by XNA. If we would change network library, we would be forced to implement these features ourselves, or altogether drop the lobby functionality in our game. This does not go well with the process we have been using, that stresses the importance of embracing change. In retrospect we should have implemented a more flexible solution, to accommodate this design principle.

5. Sound

The graphical aspect of the game definitely was one our top priorities, but even though we focused much on blowing the players' visual mind, music and sound effects in the game were also features we wanted to include. We felt that visual effects lost some of its context without corresponding audio effects, and that sound therefore was necessary in the game to offer the player a complete experience.

5.1. Sound Effects

In order to find sound effects that matched the graphical dittos, we had to search through several different sound banks. The largest sound bank, which provided us with sounds like the electronic engine, was Digiffects[58], a sound bank provided by many companies in different countries, including Swedish "Ljudproduktion AB". We also took great use of many public sound banks on the Internet: FindSounds[59], and PacDV[60].

Even though we were satisfied with many of the sound effects we found from different sound banks, we still had problems finding good sound effects for collisions. By recording our own sound effects, we felt like we were adding a more personal touch to the game, besides the fact that we needed collision sound effects.

One of the lecture rooms on the 5th floor in the EDIT house (Chalmers University, Gothenburg), became our studio when we recorded our own sound effects. The relatively compact size of this lecture room, in combination with its great acoustic qualities, made it a convenient choice to use as a recording studio.

We used a variety of different tools for our collision recording session, including crayons, a blackboard duster, a cleaning sponge, a computer mouse, and a mouse pad. The cleaning sponge and the mouse pad made excellent tools to use when we needed a more dull sound. The sounds were produced by simply dropping the object on a

desk, banging them into each other, or smashing them softly into the blackboard repeatedly.

For our collision sessions, we used a basic computer microphone[61], connected to a computer, and recorded with Adobe Audition 2.0. To increase the quality of the sound effects, we used a restoration effect called Noise Reduction. With Noise Reduction, we captured a noise profile from the quite sessions in between the sound effects to analyze the background noise in the environment. This noise profile was then used to filter the whole session from background noise, leaving only the desirable sound left, more like what a studio recording would have resulted in. Each session was filtered based upon its own background noise, which is always changing between the sessions. We also changed the amplitude of the sessions to match each other, and then splinted up each session into different sound effects, making our own sound bank.

5.2. Background Music

Just as sound effects, different types of music can also be found on the Internet. For example, PacDV, provided not only sound effects, but a variety of different instrumental music tracks as well. Even though some of the tracks were possible to have in our game, we felt that we wanted more famous background music, which could possibly give the user a nostalgic feeling. We choose to use background music from an old game called Turtles, which was released on Super Nintendo Entertainment System.

6. Results

Due to the fact that we chose to base our development process on the evolutionary development model, there were not very many requirements on the game when the development started. As the project evolved, more and more features were added, in order to reach the *game play requirement*. When it comes to the result of the project, we distinguish between features and requirements, mainly because they can be viewed as *medium priority* requirements, respectively *high priority* requirements.

6.1. Requirements

We managed to meet the major part of all our high priority requirements, such as multiplayer functionality and a fun game play.

One of the most significant low priority requirements was to develop a death rally game, where you could shoot and destroy the opponents. This requirement was not met, since we change our priorities.

6.2. Features

Here follows a list of features and their respective results:

1. *Power-ups*

   All desired power-ups, such as punch, reverse keys, slow, nitro, double points, and low friction, were implemented.

2. *Models*

   All major models, such as containers, cars, boxes, barrels, and thread plates, were made and implemented, although, it

would be desirable that all models were uniform, and that the level of polygon detail would remain the same. It would also be desirable that more miscellaneous objects had been created.

3. *Textures*

   The models' textures were satisfactory, although they were not always particularly uniform.

4. *Sound*

   The sound effects that was added satisfactory, was car engine sound, power-up sounds, and coin sounds.

5. *Graphical effects*

   The most important graphical effects that we wanted to use in the game were added with an appealing visual result.

6. *Optimization*

   Not very much time was spent on this part, which resulted in a low FPS. On our test computers, the FPS ranges from between 35 and 40.

   The specifications for our test computers were:

   AMD Athlon(tm) 64 X2 Dual Core 5000+ 2.6 GHz
   2 GB RAM, PC2-5300 DDR2
   nVidia GeForce 8800 GS

   The game was run with a resolution of 800 x 600, and all post-processing effects enabled.

6.3. Time Estimations

In a software development project, one thing that is considered to be very difficult is time estimations. The time estimations in project Lloyd was very good, and we managed to produce a playable game within the given time limit.

6.4. The Development Process

In order to evaluate the development process used in this project, each of the agile principles was evaluated.

Customer involvement The developers, along with their supervisor, have acted as customers, providing feedback on releases, and what features that should be added to what milestone.

Incremental delivery The project was divided into milestones with releases once every two weeks.

People, not process The process was not allowed to control the way of working.

Embrace change The system was initially developed to support future changes, but as it evolved, the structure became harder to maintain.

Maintain simplicity In order to maintain simplicity for the development process, focus lied on these five principles. In regard to the software, it was difficult to maintain simplicity as the project progressed.

7. Discussion

The process we used is well fitted to projects that need fast specified results. We believe that our working method enables us to achieve more features in our game than if we used a more pre-planned method. With our method, developers can get started directly with game specific features, and do not need to think about structure and writing general code.

The lack of early planning can also be a downside for this method. Solutions that seem valid early on can turn out to be unsuitable for the project. Many times, this could have been avoided by more rigorous planning.

For larger projects, we think that planning and having a good structure, become even more important. For example, if we wanted to make some major changes to our game, we would probably have to discard a lot of previous work, because our lack of good structure and generally written code. This is something we think could have been avoided by choosing another development method that focuses more on early planning, and structure throughout the project.

Choosing C# and Microsoft XNA as a framework turned out to be a good choice for this type of project. The functionality provided by XNA offered a good ground for us to start develop the game as fast as possible. Using another language, like C++, which was also in the discussions when choosing development language, would have required us to write much of the fundamental functionality ourselves. Using an already complete framework saved us time, and allowed us to start building the game directly, rather than building the foundation of the game. All C# coding was done in Microsoft Visual Studio 2008, which we consider to be an excellent developing environment, allowing us to debug the game properly. Using Microsoft Visual Studio 2008 also allowed us to use CodePlex, which provided us with a version control system in Microsoft Team Explorer.

The game concept changed several times during the project. In our opinion, the final product became a very action filled game, which was very fun to play as well. This was greatly influenced by our development method, which allowed us to change many of the concepts during the development. When we found a working concept that was fun, we centered the game idea around it. In the beginning, the game concept did not include collecting power-ups, but as we came up with the idea of a punching power-up, we found that it made the game fun and action packed. As more ideas for different power-ups emerged, we could implement these as well in the game. Soon, we had a racing car game with different kinds of power-ups, making it the suitable mix for an action car game.

We believe that it is difficult to establish all of the game concepts in an early stage. It is preferable to let the good ideas grow, and discard the bad ones, as the project develops.

8. Abbreviations

GUI – Graphical User Interface

HUD – Head-Up Display

BRDF – Bidirectional Reflection Distribution Function.

FPS – Frames Per Second.

Pixel – Picture Element.

RSS – Really Simple Syndication

Texel – Texture Element.

Appendix A

Here follows a list of milestones and what tasks that were assigned to which milestone.

Milestone 1

Implement fundamental classes; GameModel, GameObject etc.
Make it possible to load models, for now, placeholder models will do. The development of models will be ongoing throughout the project.

Milestone 2

Create a class for handling the camera. Input should be taken from both mouse and keyboard.

Milestone 3

Find a physics engine and integrate it into the game.

Milestone 4

Game logic, create and add new models.
Implement a car and make integrate it properly into the physics engine.

Milestone 5

Add effects, shadow maps.
Continue working o models.

Milestone 6

Make progress on models.
Multiplayer functionality, evaluate and implement.

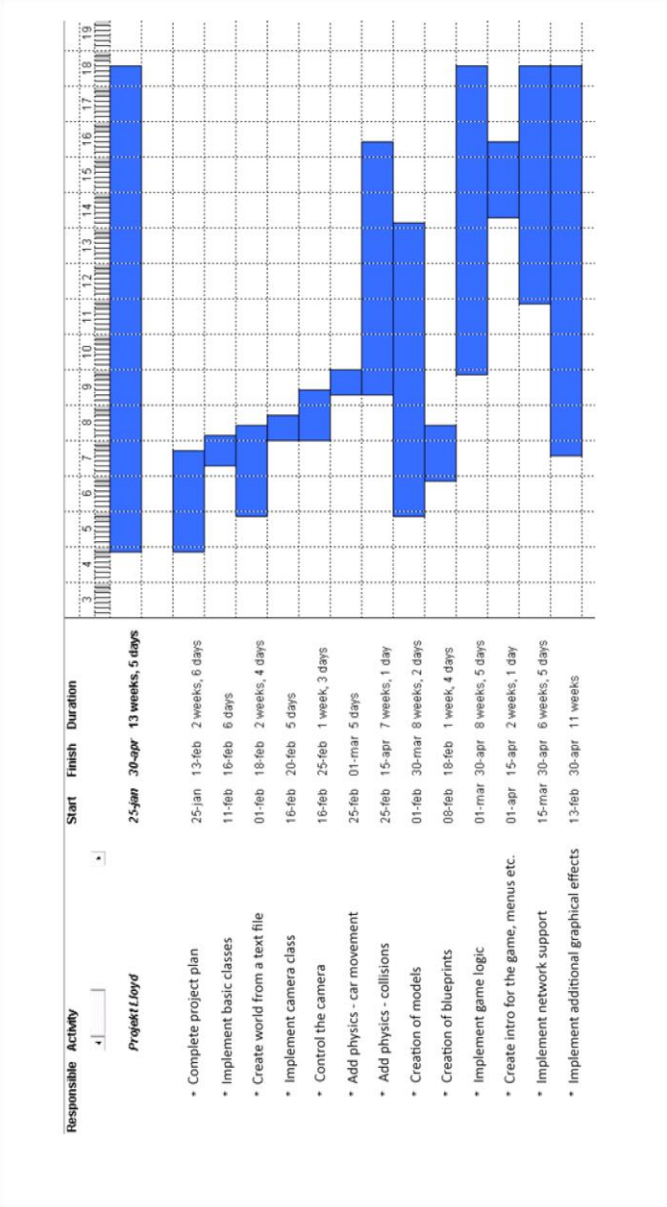
Milestone 7

- Multiplayer (via local subnet) with up to 4 players (not much of a game play yet)
- Menu system with lobby etc
- Redesigned level (demonstration purpose)
- Score
- Add power-ups
- More models, e.g. container, can, walls + floor, wooden box
- Add specular textures
- Shadows
- A simple level editor
- Show FPS

Milestone 8

- Add more power-ups
- Add more coins
- Modify the oil barrels' collision skin
- Add more models, e.g. shelves, pallet, pencil, can, table, lamp
- Make power-ups and coins respawn
- Extend the racing track
- Synchronize the power-ups (network) so that they may affect other players
- Normal maps
- Motion blur

Appendix B



Responsible	Activity	Start	Finish	Duration
	Projekt Lloyd	*25-jan*	*30-apr*	*13 weeks, 5 days*
	* Complete project plan	25-jan	13-feb	2 weeks, 6 days
	* Implement basic classes	11-feb	16-feb	6 days
	* Create world from a text file	01-feb	18-feb	2 weeks, 4 days
	* Implement camera class	16-feb	20-feb	5 days
	* Control the camera	16-feb	25-feb	1 week, 3 days
	* Add physics - car movement	25-feb	01-mar	5 days
	* Add physics - collisions	25-feb	15-apr	7 weeks, 1 day
	* Creation of models	01-feb	30-mar	8 weeks, 2 days
	* Creation of blueprints	08-feb	18-feb	1 week, 4 days
	* Implement game logic	01-mar	30-apr	8 weeks, 5 days
	* Create intro for the game, menus etc.	01-apr	15-apr	2 weeks, 1 day
	* Implement network support	15-mar	30-apr	6 weeks, 5 days
	* Implement additional graphical effects	13-feb	30-apr	11 weeks

References

[1] Remedy Entertainment. http://www.remedygames.com/games/death_rally.html (2009-05-10).

[2] Sommerville, Ian. *Software Engineering*, 8th edition; Pearson Education Limited: Harlow, 2007; pp 44, 65-70.

[3] Royce, W.W. Managing the Development of Large Software Systems. *Proceedings*. 1970; pp 1-9.

[4] Microsoft: Next Generation of Games Starts With XNA. https://www.microsoft.com/presspass/press/2004/mar04/03-24xnalaunchpr.mspx (2009-05-18).

[5] XNA Game Studio. http://msdn.microsoft.com/sv-se/directx/aa937794(en-us).aspx (2009-05-18).

[6] XNA Framework Class Library. http://msdn.microsoft.com/sv-se/library/bb203940(en-us).aspx (2009-05-18).

[7] CodePlex – Open Source Project Community. http://www.codeplex.com (2009-05-18).

[8] CodePlex Information and Discussion. http://codeplex.codeplex.com/Wiki/View.aspx?title=Using%20TortoiseSVN%20with%20CodePlex&referringTitle=CodePlex%20FAQ (2009-05-18).

[9] Eberly, D. H. *Game Physics*; Morgan Kaufmann Publishers: San Francisco, 2004.

[10] JigLibX Wiki. http://jiglibx.wikidot.com/ (2009-05-09).

[11] Hansson, H. *Craft Physics Interface*; Linköping: LiTH-ISY-EX--07/3887—SE, 2007. (http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-8497)

[12] Van den Bergen, G. *Collision detection in interactive 3D environments*; Morgan Kaufmann Publishers: San Francisco, 2004.

[13] Google Video, Limitations of 3D Physics Engines: Unintended Explosion. http://video.google.com/videoplay?docid=-1279309767827721998 (2009-05-09).

[14] Wiki, Second Life. http://wiki.secondlife.com/wiki/Physics_engine (2009-05-15).

[15] MSDN: XNA Developer Center, Standard Importers and Processors. http://msdn.microsoft.com/en-us/library/bb447762.aspx (2009-05-10).

[16] Autodesk 3ds Max. http://www.autodesk.com/3dsmax (2009-05-10).

[17] Blender. http://www.blender.org (2009-05-10).

[18] Autodesk Maya. http://www.autodesk.com/maya (2009-05-10).

[19] Pixologic. http://www.pixologic.com (2009-05-10).

[20] Blender Features. http://www.blender.org/features-gallery/features/ (2009-05-10).

[21] Blender 3D: Noob to Pro. http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro (2009-05-10).

[22] Akenine-Möller, T.; Haines, E.; Hoffman N. *Real-Time Rendering*, Third Edition; A K Peters: Wellesley, 2008; pp. 150-151, 154, 187, 257.

[23] CrazyBump. http://www.crazybump.com/ (2009-05-15).

[24] CG Textures. http://www.cgtextures.com/ (2009-05-15).

[25] MSDN, XNA Developer Center, Content Pipeline. http://msdn.microsoft.com/en-us/library/bb203887.aspx (2009-05-19).

[26] Truevision TGA File Format Specification. http://www.dca.fee.unicamp.br/~martino/disciplinas/ea978/tgaffs.pdf (2009-05-19).

[27] MSDN, XNA Developer Center, XNA Creators Club Online - game state management. http://creators.xna.com/en-US/samples/gamestatemanagement (2009-05-19).

[28] MSDN, XNA Developer Center, SpriteFont Class. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.spritefont.aspx (2009-05-19).

[29] MSDN, XNA Developer Center, Microsoft.Xna.Framework.Graphics Namespace. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.aspx (2009-05-19).

[30] Koenderink, J. J., van Doorn, A. J., and Stavridi, M. *Bidirectional Reflection Distribution Function Expressed in terms of surface scattering mode*, Proceedings of the 4th European Conference on Computer Vision, 1996, vol. 2, pp. 28-39.

[31] *O'Rorke*, J. *GPU Gems: Integrating Shaders into Applications,* R.F., Ed.; Addison-Wesley Professional, location: Boston, MA, 2004.

[32] MSDN, XNA Developer Center, Effect Class. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.effect.aspx (2009-05-19).

[33] Phong, B.T. Illumination for Computer Generated Pictures, *Communications of the ACM*, 1975, *vol. 18, no. 6*, pp. 311-317.

[34] Gouraud, H., Continuous Shading of Curved Surfaces, *IEEE Transactions on Computers*, 1971, *vol. C-20*, pp. 623-629.

[35] Catmull E., *Computer Display of Curved Surfaces,* Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures, Los Angeles, CA, May 1975, pp. 11-17.

[36] McReynolds T.; Blythe D. *Advanced Graphics Programming Using OpenGL*; The Morgan Kaufmann Series in Computer Graphics; Morgan Kaufmann: San Francisco, 2005.

[37] Fosner R. *Real-Time Shader Programming*; The Morgan Kaufmann Series in Computer Graphics; Morgan Kaufmann: San Francisco, 2002.

[38] Blinn, J.F. *Models of Light Reflection for Computer Synthesized Pictures*. *ACM Computer Graphics (*SIGGRAPH '77 Proceedings), July 1977. pp. 192-198.

[39] Wanger, L. The effect of shadow quality on the perception of spatial relationships in computer generated imagery, *Computer Graphics (1992 Symposium on Interactive 3D Graphics)*, 1992, *vol. 25, no. 2,* pp. 39-42.

[40] Donnelly, W., and Lauritzen, A., *Variance Shadow Maps,* Proceedings Symposium on Interactive 3D Graphics, 2006, pp. 161-165.

[41] Williams, L., *Casting Curved Shadows on Curved Surfaces,* Computer Graphics (SIGGRAPH Proceedings), August 1978, pp. 270-274.

[42] Schüler, C. *Eliminating Surface Acne with Gradient Shadow Mapping*, Engel W., Ed.; *ShaderX4*, Charles River Media: Boston, USA, 2005, pp. 289-297.

[43] Blinn, J.; In *Simulation of wrinkled surfaces*, Computer Graphics (SIGGRAPH '78 Proceedings), August 1978, pp. 286-292.

[44] Schlag, J. *Fast embossing effects on raster image data*; P. S. Heckbert., Ed.; Graphics Gems Series; Academic Press: San Diego, 1994; pp. 433-437.

[45] Kaneko, T.; Takahei, T.; Inami, M.; Kawakami, N.; Yanagida, Y.; Maeda, T.; Tachi, S. In *Detailed Shape Representation with Parallax Mapping*, Proceedings of ICAT, Dec. 2001; Tokyo, 2001; pp. 205-208.

[46] Welsh, T. *Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*, Infiscape Corp., 2004.

[47] Rosado, G., *GPU Gems 3, Motion Blur as a Post-Processing Effect, H.N., Ed.;* Addison-Wesley Professional, location: Boston, MA, 2008.

[48] Langer M.S.; Bulthoff H.H. Depth discrimination from shading under diffuse lighting. *Perception*. 2000, 29, 0301-0066.

[49] Mittring, M., *Finding Next Gen-CryEngine 2*, SIGGRAPH 2007 Advanced Real-Time Rendering in 3D Graphics and Games course notes, 2007, http://ati.amd.com/developer/gdc/2007/Mittring-Finding_NextGen_CryEngine2(Siggraph07).pdf.

[50] Anirudh, S. S., *High Dynamic Range Rendering*, GameDev.net. http://www.gamedev.net/reference/articles/article2108.asp (2009-05-19).

[51] van der Burg, J. *Building an Advanced Particle System*, http://www.gamasutra.com/features/20000623/vanderburg_pfv.htm (2009-08-06).

[52] Lorcach, T. *Soft Particles*, http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf (2009-05-19).

[53] Kurose, J. F.; Ross, K. W. *Computer Networking: A top down approach*, 4th ed.; Addison Wesley: Boston, 2008.

[54] PeerToPeer - lidgren-network - The Peer to peer class - Google Code. http://code.google.com/p/lidgren-network/wiki/PeerToPeer (2009-05-19).

[55] Essentials - Windows Live. http://download.live.com/ (2009-05-19).

[56] MSDN, XNA Developer Center, XNA Frequently Asked Questions. http://msdn.microsoft.com/sv-se/xna/aa937793(en-us).aspx (2009-05-19).

[57] MSDN, XNA Developer Center, Voice Support. http://msdn.microsoft.com/en-us/library/bb976068.aspx (2009-05-19).

[58] Ljudproduktion – Digiffects. http://www.ljudproduktion.se/digiffects.html (2009-05-18).

[59] FindSounds. http://www.findsounds.com/ (2009-05-18).

[60] Free Sound Effects. http://www.pacdv.com/sounds/index.html (2009-05-18).

[61] Clas Ohlson Internetbutik. http://www.clasohlson.se/Product/Product.aspx?id=60642904 (2009-05-18).