MAIN PROJECT:

TITTEL

AIR FEAR

AUTHORS:

Bjørn Willy Stokkenes
Espen Åmodt
Sverre Strandenes

Date:

05.23.2002

SAMMENDRAG AV HOVEDPROSJEKT

Tittel:	AirFear	Nr. :	
	AirFear – DirectX 3D Multiplayer Game	Dato :	23.05.2002
Deltaker(e):	Bjørn Willy Stokkenes		
	Espen Åmodt		
	Sverre Strandenes		
Veileder(e):	Svein Gautestad		
Oppdragsgiver:	Innerloop		
Kontaktperson:	Henning Rokkling		
Stikkord	DirectX, Game-Programming, Multiplayer, 3D Sound.		

Antall sider:	94	Antall bilag: 5	Tilgjengelighet (åpen/konfidensiell): Åpen

Kort beskrivelse av hovedprosjektet:

AirFear prosjektet omhandler spill programmering mot Microsofts DirectX API. Fokus er satt på spill-motor programmeing med utvidelses-muligheter. Med spill-motor, menes et rammeverk som driver grafikk, lyd, input, nettverk og andre spill-relaterte grunnstener.

Preface

Bjørn Willy Stokkenes, Espen Åmodt and Sverre Strandenes started in November 2002 the work with the AirFear project. This is the final graduation project, before they become computer engineers at Høgskolen I Gjøvik (Gjøvik College, located in Norway). Since the three of us are interested in the work behind computer games, as well as playing the games themselves, we decided to try to make a game of our own. After a consultation with the 3D game company Innerloop in Oslo (which also became our employer), our mind was set. This turned out to be an interesting and challenging assignment. The final product of our work is published in this document. We would like to say thanks to the following people:

Mikael Kalms
- for math assistance and general game-engine advices.

Gustav Gran
- for letting us use his office for 3 months.

Arne Magnus Bakke
- for his code-template used while making a part of the collision detection class.

Tor Slind
- for advice on how to use Maple during testing of things like regulation- techniques.

Ivar Farup
- for the graph delimiting algorithm used in collision-detection (players against players), and pointing us in the right direction concerning calculation of billboard orientations.

Gloom
- for his music contribution.

Kravitz
- for his cool render-state tip.

This project is the beginning of an object oriented game-engine, providing intermediate to advanced functionality and design for later expansion. This engine is going to compete in the game-development competition at Assembly 2002, held in Helsinki (Finland) in August.
Parts of this projects code will not be open to the public, but most of the techniques used, is explained in this document.

Gjøvik / Norway, 23rd of May 2002

Bjørn Willy Stokkenes **Espen Åmodt** **Sverre Strandenes**

Contents

MAIN PART

Chapter 1

1 Introduction

1.1 Organization of this report

After consulting our supervisor Svein Gautestad, it was determined that we should follow the standard template for documenting graduation projects at HIG. However, since this project is partly a research project, the group decided that some discrepancy had to be done concerning the standard template which is designed to handle stable projects with more certainty in the targeted applications. The report is therefore written in a slightly modified template which fit our project better. We feel it is worth mentioning that some readers might find some of the technical terms difficult to understand. Likewise some of the mathematics is beyond the scope of engineering mathematics. In this case, see the section where some of the terminology is explained.

1.2 Connection between chapters

Chapter 1:
This chapter contains introduction, task-definitions, limitations, end-users, terminology and the developers' background.

Chapter 2:
Specification of demands, what we are about to solve. Here we write some words about the limitations and functionality of the upcoming system.

Chapter 3:
This chapter describes what this game-engine is capable of doing.

Chapter 4:
The main features of the system are described. This chapter shows how the group has decided solve the demands specified in chapter 2. The architecture of the system is showed. This is the chapter where the reader should get an idea of how the different modules in the system are connected to each other.

Chapter 5:
This chapter contains information mostly for the programmers. Some of the source is showed, and the most important algorithms are explained. As said, this chapter is meant for people with intermediate to advanced programming skills.

Chapter 6:
This chapter shows how the tested the system under the development process. We also show how we have handled the quality assurance. A person with no programming skills should easily be able to read this chapter.

Chapter 7:
In this chapter we discuss the results versus the initial plans.

Chapter 8:
This chapter deals with conclusion and what the group have learned during this short period.

Chapter 9:
List of literature used.

Chapter 10:
Appendixes.

1.3 Terminology

3D sound:
A 3D sound is a sound that in addition to sound data has information about its alignment and position in a 3D world.

3ds-format:
This is 3D-studios scene-file format from the earlier versions of the package. The format contains mesh, texture and material information. Used together with the Conv3ds utility to convert 3d-objects into the x-file format that can be handled by DirectX graphics.

3d-world:
When speaking in graphics-terms a 3D world is a set of vertices, lines and faces arranged in a coordinate-system along the X,Y and Z axises.

API:
Application Programmers Interface

Billboard:
In much the same way as point sprites, billboards contain a 2dimensional texture that always faces the camera. In our implementation some extra functionality beyond the scope of point sprites was needed to get the desired effect from the explosion.

Camera:
A camera is a phrase, used to describe the position of the viewer in a 3d environment.

Collision Detection:
A test whether one object collides with another object.

Cube-Volume:
A data structure containing information about which faces reside in a particular cube-shaped area of the 3D world. Used in reducing computation and drawing overhead in the areas of collision-detection and rendering respectively.

DirectX:
This is Microsoft's multimedia/game API.

Face:
A face is the area of a plane constrained by a set of edges that are the connections between a set of vertices. Normally for game-purposes, filled triangles consisting of 3 vertices connected by 3 edges are used. Often these triangles have a texture to give extra details to the surface.

Joystick:
This is a control device designed for games. It consists of a control-stick (like in a helicopter) and a range of buttons which one assigns different game-actions to.

Linear IntERPolation(LERP):
This is the mathematical process of moving in steps from one point to another point in a straight line.

Mesh:
A mesh is a grouping of primitives such as vertices, edges or faces. Often referred to as a 3d-object. These describe the game's 3D graphics. Examples of meshes used in this game, are the player's craft, the levels, the explosions and the bullets.

Multiplayer:
This is a term used to describe many players playing against each other over a network connection.

Point Sprite:
This is a primitive containing texture, scale and position information. Used for rendering 2dimensional, textured points in the game. These are used for displaying the fragments of an explosion in the game. Point sprites are also referred to as particles.

Quaternion:
A 4 dimensional complex number with one real component and 3 imaginary axes. In 3d graphics terms normally used for high precision rotation-algorithms and in spherical line interpolation of camera-angles.

SDK:
Software Development Kit

Spherical Line IntERPolation (SLERP):
As opposed to a normal linear interpolation, this interpolates across an arc aligned with a spherical surface rather than in a straight line.

Texture:
In our case a texture is a flat bitmap image that is assigned to a surface of a mesh's faces to give an impression of more detail without the computational expenses of adding more 3D geometry such as vertices and faces.

Vertex (plural vertices):
A vertex is a point used as the most primitive building block for describing the shape of a mesh. A vertex has no volume only position information.

X-File:
This is a file format developed by Microsoft. The file-format contains mesh, texture and material-information for a 3D object.

1.4 Limitations and problem domain

Due to our current knowledge about how to program against the DirectX interface and game-programming it selves, there is no good way of determining the problems that could arise, nor the limitations that must be set. We can only make some assumptions.

We want the program to have sufficient functionality to make the user able to control a camera through a 3d scene. The scene will be built by 3d cubes and/or simple primitives. We hope to use texture mapped objects. By texture mapping, we mean that we could have the surfaces of the objects in the 3d scene mapped with a 2d bitmap-drawing. We want textures in the scene, because this will improve the view of the game. When this is said, there must be a clear understanding to the reader, that the game is not intended to be filled by nice graphics. Instead we will do our best to make the engine ("the code behind the scenes") be able to control and render smoothly moving graphics. The modelling itself is not of high priority. Also, the purpose of the project is not mainly to get a product that can be sold or made profit of. The idea is to get to know how it would be to work in a team, and do some hard core design and programming.

A project focused on creating a game or game-engine, could never be accepted as a serious project, but this doesn't mean game-programming is easy. Keep in mind that this is a school-project, where the sole purposes are to learn how to work in a team, and get more training in programming and other subtleties that a project might contain. Therefore, for us, this project means nothing more than serious learning.

Note: After about one month of programming and testing, we had already reached our goals of this project. Therefore we continued going further into the DirectX documentation, and reading more books concerning game-programming. We thought it was worth mentioning this, so the reader won't be confused when reading about things like etc, Multiplayer, and so on.

1.5 Target group

As this project is more a research project than a project meant for the consumer market, there is not really a targeted group. We hope that people who are interested in real-time graphics and/or game-programming will find the information in this report useful.

1.6 Purpose of this project

The purpose of this project is to make the members of the project-group get some idea on how to work on larger projects. We also hope to learn more about how to work in a team, along with getting better programming skills. As we were the ones who contacted Innerloop (our employer) and not the other way around, we have some room to play our own bosses. The good thing about this is that we are free to make choices of our own. The disadvantages are that we get less guidance concerning programming, and setting the limitations of the project.

1.7 The developers background

All three of us are currently wrapping up three years of higher education studying for an engineer degree in computer science. Bjørn Willy Stokkenes has a past starting out as a Commodore 64 Motorola 6502/6505 coder in 1986 and later joined the Amiga demo-scene and programmed in assembly on hardware controlled by the MC680x0 series of processors. Notable productions being the official Gathering99 Invitation intro used to promote this happening to the Amiga user-base. He has also programmed a great deal of other processors like PIC-controllers where size and speed optimizing is essential. As a preparation prior to this project, he chose to take a course in 3d-animation and modelling and distributed operating systems.

Espen Åmodt is still active on the Amiga demo-scene, releasing his first productions in 1995 programmed in MC680x0 assembly language. Since then he has together with his group released several top ranked productions such as the currently highest ranked 4096-bytes intro in the world. At some time he held the top position on the official "Eurochart" in the coder category. From his experience in the scene he has also acquired graphical coding skills as well as basic capabilities in 3d-modelling, pixel-drawing and design. After having developed a 3d-engine in software on the Amiga platform, the next natural move was to move to hardware accelerated graphics-coding on PC.

All of us were complete beginners at DirectX programming upon project initiation, and had little experience programming graphics and sound for the windows platform. Bjørn Willy had a brief knowledge of the features of DirectX in the areas of sound, graphics and input handling.

1.8 Chosen way to work

The man which first brought up the idea of making a game, using DirectX, was Bjørn Willy Stokkenes. This, along with his knowledge on the capabilities of the DirectX, made the decision easy in the choice of a project leader. When this is said, we must add that very little leadership was needed. When there were new decisions to make, we were all sitting down, making sure everyone agreed and understood the new line of actions to take, in solving the current problem.

The starting phase of this project was particularly problematic due to the members of the group's lack of knowledge in game programming. Research, planning, design and experimenting had to be done in plenary, so that every team-member could understand the basic principles. Two of the members Bjørn Willy Stokkenes and Espen Åmodt, had already a great deal of experience in the demo scene. But it was clear that this is certainly not enough to embark on a fully fledged game-project. To program a game takes a lot more than just showing fancy graphics and playing sound at the same time. When you have to take care of the user demands (input), and also hook this up with network capabilities, along with large scale scenes etc, the most unthinkable problems may arise.

The first month, time was used to read books and DirectX documentation. Finally we got a room, where we could work together (1st of February 2002). Also during the first month (January), it was decided that the best way to work, was that each member should investigate in detail a specific problem at a time. This meant that one should specialize in etc. sound, while another should work primarily on graphics and so on. This doesn't mean that not all members were involved in all parts of the code, but that each of us had his own main area. The idea was to build the game-engine, module by module, continually testing all modules, and later sew these modules together. Then make them communicate with each other. Some people call it the bottom-up method. Think of the bottom-up method as a pyramid, where the top of the pyramid, is the main loop, and at the bottom, there are several small modules that are being coupled together on the way up. This method has proven to be a successful way of developing games before. There is also a top-down approach to develop a program (game), but this method requires a vast knowledge of all problems which may occur.

There were also other techniques for systems development used. Since we had the possibilities to make our own alternative turns during the project, and we were constantly getting new ideas on how to make the game-engine meet new demands from the public.

Due to the advanced techniques and algorithms, used in a great deal of the code, parallel-development was not a good choice when it came to efficiency (both at code time, and runtime). Therefore, large parts of the project was designed and coded in pair.

1.9 Development model

Knowing that developing this game would lead to a lot of experimenting with the DirectX API calls and encountering unknown problems due to inexperience in game-programming, there was no way we could decide on a complete and definitive design for the application early on. A top-down approach would not work since the functionality of each module and what communication was needed between the modules was unknown. The consequence of such an approach would be that the low-level classes in the hierarchy would suffer badly from mistakes made in the higher levels of the class-hierarchy.

Considering the project's size and the given time-limit of one semester a "bottom-up approach" was chosen as development model.

The planning:
In the initial planning the "User stories" of each module were laid out describing in broad strokes the name and known functionality of each module.

Implementation:
Basic low-level functionality was created for one class at a time with appropriate tests written to analyze the output of each method. Great care was taken to ensure that methods and variables had logical names that would make the code readable without riddling it with heaps of unnecessary comments.

System metaphor:
Naming conventions used conform loosely to the DirectX Common File Framework's style of prefixing to maintain a uniform look and feel of all sources. Each method has a header description that is parseable by the development-environment that provides the programmer with comments while browsing the methods in code-completion mode.

Pair programming:
All code was written by two programmers sitting at one machine. All code was reviewed as it was written. Pair programming means two people working side-by-side on the same problem on the same computer. Typically, the person using the keyboard and mouse will be focused primarily on coding the current method, while the other coder will focus more on strategic issues, like how the method fits into the system, and is it needed at all.

Collective Code Ownership:
No parts of the code, was at any time inaccessible by the other team members. Constant updates were made to the current code-base shared on a local network between the developers.

While developing the algorithms used in the project a 'Do the simplest thing that could possibly work' (DTSTTCPW) line of thinking was used. As long as our development machines had enough resources for the task, no attempt at optimizing was made. At a later stage, the modules proving to be bottlenecks were re-factored with optimizations in mind.

Not all of XP's practices were put into use at all times. For achieving the goals/dreams of this project, time didn't always allow a non-dynamic structure of development. Some less critical modules didn't require rigorous testing, or couldn't be tested thoroughly until another module was done.

Chapter 2

2 Specification of demands

This is where we describe the project's initial functional and operational requirements.

2.1 Background

When the main-project options were presented we already had an own idea for a project which built on our interest for game-technology. There's a wide range of games on the market. The last scream was to run a game through the internet web-browser using a package such as Flash, Director or as a Java-Applet.

The disadvantages of these games are that they are under heavy performance-constraints because of processor overhead and lack of hardware acceleration. The advantage being that they run on multiple platforms. Our interest was to develop a product that took advantage of superior hardware in order to get high performance 3D graphics and sound.

After investigating the different tools/interfaces that had the possibilities of fulfilling our requirements, the choice fell on DirectX for the Windows platform. This is currently the leading game development and multimedia platform.

After discussions with Tom Røise, we knew that there would be a great advantage in having a game-development company as our employer. After a roundtrip on the internet, we discovered Innerloop's homepage. We decided to contact them and they were willing to become our employers. Tuesday the 13 of November 2001, we had a meeting in Oslo and talked to Stein Pedersen at Innerloop. He gave away some tips on how we should proceed and gave us a few sites on the internet that could be used for reference. We also got some help to decide on realistic goals for the project.

2.2 Operating environment

The AirFear game-engine should be designed to need a Microsoft Windows platform containing DirectX version 8.1 or newer. It should be designed to give higher frame rates if the computer that runs the game has a faster CPU, but a faster CPU should not mean that some players got better flying capabilities. By flying capabilities, we mean things like rotation velocity or translation velocity in the 3d environment. It should be designed to run in a window. There should be no restrictions to the screen resolution used as default at the desktop, but it requires that the depth of colours is at least 16 bits per pixel (65K colours).

2.2.1 What is DirectX [6]

DirectX was designed by Microsoft as a set of low-level APIs for making complex multimedia applications that is compatible with a wide array of hardware while maintaining speed.
It serves as a shortcut to access input devices, graphics, network and sound hardware which are the parts of the computer that multimedia applications and especially games, require the most performance from. It's a standard development platform for all windows-based PCs.

In common Microsoft fashion, the first releases were ridden with bugs and it's widely accepted that anything before release 6.0 of the API, is close to unusable.

With version 8.0 things have taken a different turn. DirectX has come a long way, and is now the standard of the game-business. As long as the hardware manufacturers assure that their hardware is DirectX compatible, programmers and engineers are more free to focus on their applications functionality and interface than on "banging the metal" directly.

A common mistake by the uninitiated is to think that DirectX provides complete solutions. It's only to be regarded as a toolkit for the multimedia-programmer, and not some kind of complete "Game Maker" package. All semantics of the application is solely up to the individual programmer and little or no structure is forced upon the application just because it uses DirectX.

A huge advantage is the fact that DirectX is under constant development. Critical performance related routines are optimized and can serve to improve the developer's products over time as long as they use the DirectX API. A slow game-engine made today, might become a cutting edge high performance engine with the next release of the API.

As with all APIs there is the threat of incomplete documentation and routines not acting the way that the documentation states. Generally, the APIs are black boxes which you should do some testing on before deciding on what features to use. This is mostly a problem with fresh new features.

The main components of DirectX:

- DirectX Graphics
- DirectX Audio
- DirectInput
- DirectPlay

- DirectShow
- DirectSetup

DirectX Graphics:

An application programming interface (API) that you can use for all graphics programming. Herein lies the functionality for programming hardware accelerated 3D graphics and also software 2D graphics. The component includes the Direct3DX utility library that assists in many graphics programming tasks.

DirectX Audio:

This is an API for playing and scheduling various types of sounds. Dynamic soundtracks, static sounds, 3d sounds with hardware acceleration, MIDI files, downloadable sounds (DLS). This API also has features for capturing sounds from external sources and features for manipulation of sound in real-time. Possibilities for extension by specialized applications such as reading, processing or parsing of custom sound formats are also present.

DirectInput:

Includes support for a wide variety of input devices such as joysticks, mice, the keyboard and also devices with force-feedback capabilities. DirectInput communicates directly with hardware drivers, rather than relying on, and waiting for Windows System-Call messages.

DirectPlay:

Provides support functions for network oriented programming. Message-queues, congestion control and the likes are handled.

DirectShow

This is a set of functions that provides for high-quality capture and playback of multimedia streams.

DirectSetup

This is a simple API that provides one-call installation of the DirectX components.

2.3 Users of the system

The average user of this product will be your normal person whose computer skills are average. The user must be capable of starting a program inside windows, and needs to be familiar with the traditional graphical user interface abstraction. This product is not intended for people suffering from epilepsy which could be accidentally triggered by the visual effects generated by this program.

2.4 Functionality

We are planning to do a simple airplane/spaceship game where we could fly through a 3d-landscape consisting of simple models such as cubes and/or other primitives. The player should be able to control his craft using the joystick. We want to explore the possibilities that the newest version of DirectX could provide such as the new capabilities of the Nvidia GeForce3 card.

2.5 Aspects concerning life cycle

It's impossible to make a fully functional and errorless system during a one semester graduation project. If the lifecycle of this project is to exceed beyond our development process, we have to make sure that it is easily expandable. When the clock starts ticking "time-out", we will only make sure that the program can be run and shown. The process of continuing on this game-engine will hopefully go on. Therefore we must be careful by designing the structure of the code for further expansion. We must keep in mind, things like AI, Multiplayer, Multiple-objects-control and other subtleties concerning game-development has to be taken into account.

Chapter 3

3 Analysis

In this chapter we make the analysis of what we have fulfilled in these three to four months.

3.1 What is this game-engine capable of doing

As mentioned in "LIMITATION AND PROBLEM DOMAIN", the game-engine did become a great deal more sophisticated than expected by the AirFear-team in the first place.

3.1.1 Graphics

Modules, and algorithms, dealing with graphics:

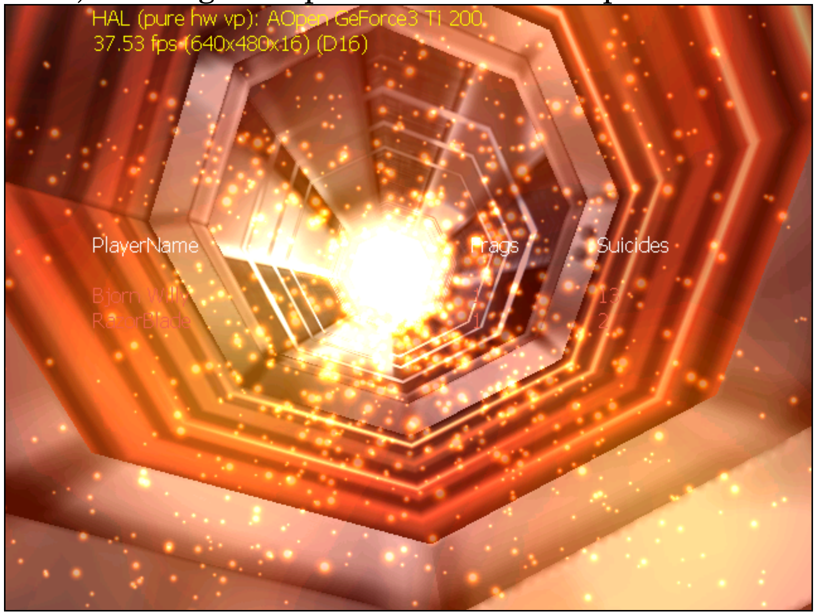
- Loading of 3d-models
- Delimiting
- Explosion generation
- Transparency

Loading of 3d-models:
First of all, the engine was expanded to cope with 3ds files, created using 3d Studio Max (version 3.0 or 4.0). These files had to be converted to .x-files in order to be compatible with the graphics-modules we made.

Delimiting:
Graphics was further expanded with algorithms, taking care of splitting geometry into cube-volumes. These cube-volumes are used along with delimiting algorithms, making sure that only the graphics which need to be drawn are drawn. The result of using such techniques, are possibilities for more detailed scenes whilst still maintaining a high frame-rate.

Explosion generation:
For a game to be nice looking, it needs something that "kicks". Therefore, near the end of this semester, we decided to expand the engine with explosion effects. The explosion consists of 2 parts. A billboard and a particle-generation function. This could be used to make space-crafts and similar objects explode. The particles are actually point-sprites, while the billboard is a 2d-polygon (consisting of 4 vertexes) which is always facing the camera.

An in-game shot, showing an explosion of the local space-craft:


HAL (pure hw vp): AOpen GeForce3 Ti 200
37.53 fps (640x480x16) (D16)
PlayerName
Frags
Suicides

Transparency:
Some of the objects in the scene are transparent.

3.1.2 Multiplayer

The engine currently supports up to 8 players, playing simultaneously.

3.1.3 Collision Detection

The engine supports a vast range algorithms dealing with collision-detection.

These are some vital collision related functions:

- Player against player
- Player against geometry
- Bullet against player
- Bullet against geometry
- Conflict solving
- Delimiting

Player against player:
Detect if a collision revolts between two or more players.

Player against geometry:
Detect if a players "vehicle" has crashed with the geometry.

Bullet against player:
Detects if a bullet (shot by etc: player1) has hit another player's (for example: player5) "vehicle".

Bullet against geometry:
Detect if a bullet has hit the geometry.

Conflict solving:
Suppose a bullet is traveling at great speed. There might insurrect cases where the bullet travels through both a "player-vehicle", and a wall at one single screen-update. The engine has been carefully designed to handle these types of cases. This means, that etc; if a bullet hits a "player-vehicle", before it hit a wall (in one single update), the bullet will be disabled, and actions will be taken, assuring that only the "player-vehicle" will be hit, and not the wall.

Delimiting:
As mentioned under "Graphics", the geometry is split into cube-volumes, to make sure that only the graphics that need to be rendered, is rotated, translated and drawn on the screen. This is also done concerning the collision-detection. The difference is that other delimiting algorithms are used, to get a more optimized performance for the collision-detection-types "Player against geometry" and "Bullet against geometry". Also smaller cube-volumes are used.

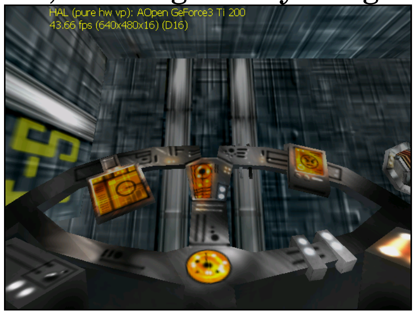
3.1.4 Response

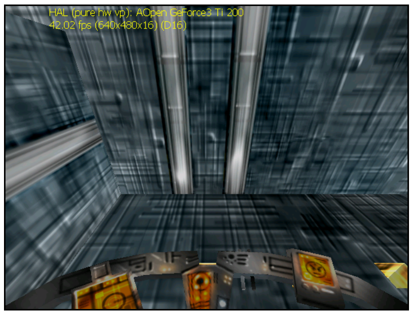
Response is one of the last features implemented. Response is among the advanced sections in this project. So what is response? Response is an extension to the collision-detection module. Instead of only detecting if a polygon has been hit, and return the distance to it, our response-code continues testing against the geometry calculating a new bounce-position. The code consists of mathematic expressions for calculating the correct angles and velocities, and will continue calling itself recursively, until a valid position in the scene has been found for the object in consideration. In our test-game, we use the response-part to bounce the space-crafts off the walls. If the craft has high enough speed, and its hit-angle is close to the normal of the surface, the ship will explode.

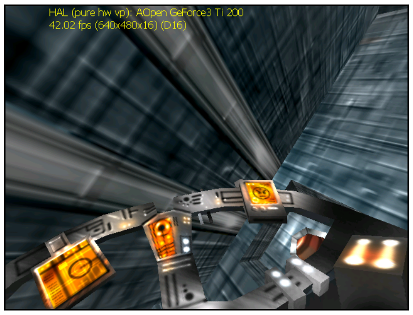
3.1.5 G-Forces

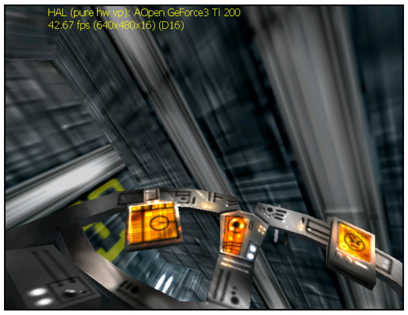
To simulate g-forces, the game-engine has algorithms taking care of spherical (SLERP) [1] and straight (LERP) interpolation between two positions in 3d space. The test-game has the inside of the space-crafts as a separate object. This object has the effects of g-forces on the acceleration components.

In-game shots, showing clearly the g-forces:


HAL (pure hw vp): AOpen GeForce3 Ti 200
43.66 fps (640x480x16) (D16)


HAL (pure hw vp): AOpen GeForce3 Ti 200
42.02 fps (640x480x16) (D16)


HAL (pure hw vp): AOpen GeForce3 Ti 200
42.02 fps (640x480x16) (D16)


HAL (pure hw vp): AOpen GeForce3 Ti 200
42.67 fps (640x480x16) (D16)

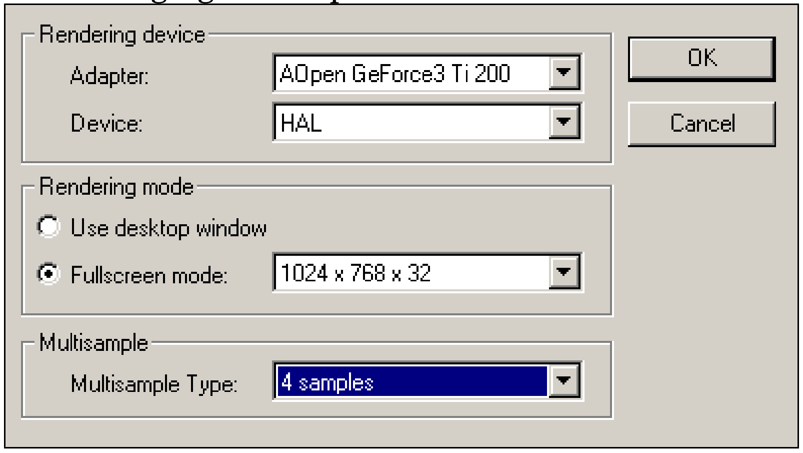
3.1.6 Z-Buffer correction

If you have played some 3d games, you might have discovered objects, either near to the camera or far away from the camera that is not drawn correctly, or switching on and off randomly. This could be the effect of a low detailed z-buffer, where the code-writers have done nothing to prevent z-buffer overlaps. This game-engine uses logarithmic z-buffer, which has higher precision the closer to the camera the rendered graphics are drawn. This makes sure there are no z-buffer errors close to the camera. Still, there was the problem of objects far away from the camera turning on and off when intersecting neighbor objects. This problem is solved by rendering the different portions of the scene with different z-buffers.

3.1.7 In-game screen-resolution change

Remember when Unreal Tournament first arrived at the market. One of the AirFear-team members watched an interview with one of the game-engine-writers. He said that they could now for the first time present a game which could change screen-resolution while in-game. This is a feature that requires careful design of the graphics-modules in the game-engine, making sure that every device-dependant object that's uploaded to the hardware could be easily deleted prior to resolution change and restored afterwards. Operations such as resizing the window also needs special handling when using dynamic vertex-buffers. Our game-engine has this capability, and will do this at run-time (game-play). The user of the system may also change the levels of multisampling. Multisampling is used for anti-aliasing in the game.

In-game GUI for changing screen-parameters:



Rendering device

Adapter: AOpen GeForce3 Ti 200

Device: HAL

Rendering mode

○ Use desktop window

● Fullscreen mode: 1024 x 768 x 32

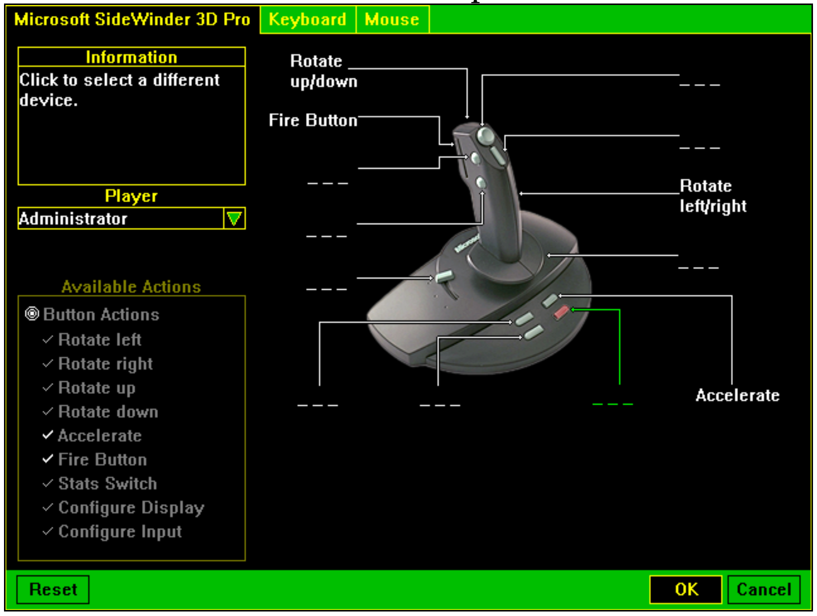
Multisample

Multisample Type: 4 samples

3.1.8 Vast range controller support

The engine supports every game-controller compatible with DirectX on the market. The user may choose his own controller, as joystick, mouse or keyboard. Even racing wheel (not very suitable) is possible. In addition, the user may also define her preferred keys or buttons.

In-game screen-shot of the controller's setup-screen:



Microsoft SideWinder 3D Pro | Keyboard | Mouse

Information

Click to select a different device.

Player

Administrator

Available Actions

◉ Button Actions
- ✓ Rotate left
- ✓ Rotate right
- ✓ Rotate up
- ✓ Rotate down
- ✓ Accelerate
- ✓ Fire Button
- ✓ Stats Switch
- ✓ Configure Display
- ✓ Configure Input

3.1.9 Multiple large-scale scenes

The game-engine reads the entire scene by using several configuration files, which is partly written by the front end, and partly written by the scene-creator. By combining this with a module that reads and interpret this information, the engine has support for multiple scenes. Each scene could be made by using a highly structured file-hierarchy.

3.1.10 Lightening

The engine supports simple directional lights. These are hardware accelerated if your graphics card supports it. There is room for adding other different types of light like omni-lights.

3.1.11 Sound

The sound modules are designed with many real world factors in mind.

Here are the main features:

- Stereo Sound
- 3D Sound
- Doppler effects
- Sound garbling
- MP3 Player

Stereo Sound:
To be able to play some music during some sequences of a game, stereo sound is important. This is implemented in the sound-module, where one might specify “Stereo Sound” as parameter when initiating the preferred sound-file.

3D Sound:
One of the nicest features in the game-engine, are the capability to play 3d sound. The sound could be placed anywhere in a scene, specifying volume, speed, distance etc.

Doppler effects:
As an amendment to the sound-module, the game-engine supports Doppler effects. This is a heavy processor demanding task that is currently disabled until the hardware in the soundcards is capable of meeting these kinds of demands. The Doppler effects are made by adjusting the pitch of the sounds according to the listener’s position and velocity. The velocity and decay of the sound-sources are also part of the calculation.

Sound garbling:
To make a nifty finalization on the sound-modules, the sounds are garbled according to their respective positions in the scene. This means that a sound that is close, and is not moving is crispy and clear, while a sound that is far

away in the scene is garbled. The effect also motivates the orientation of the listener.

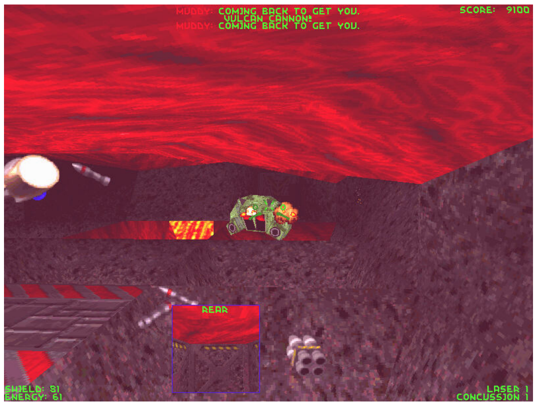
MP3 Player:
The game-engine also supports playing of .mp3 streams. This could be used as background music, front-end-music, end-of-match etc.

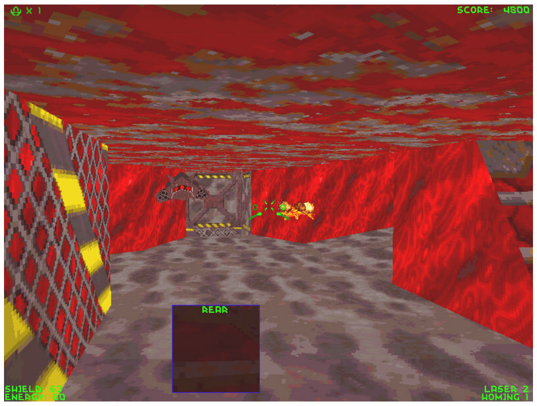
3.2 Similar projects/games

There are many games on the marked. Among games using similar game-engines are the games; Descent and Descent2 (sold worldwide). These games make the user being able to fly through tunnels and halls in the earth, where orientation is a dilemma. We thought of mentioning these games, since they are very alike what our game-engine performs. The Descent games did only present stereo sound, and none of the features we present here. This is probably because 3d-sound and garbling are features that were impossible until recently. Also these games had a maximum resolution of 800x600, where the AirFear-engine supports all resolutions available on the current graphics-card. In return, the Descent2 had a lot of graphics that were nice to look at. When this is said, remember that we were 3 people designing this game within 3 months, where none of us had experience in game-programming. The Descent2-team had 2-3 years, where the team consisted of about 20 experienced game-developers, and they where only expanding an already existing game-engine (Descent), which they had made their-selves.

Screen-shot from Descent1:

Screen-shots from Descent2:

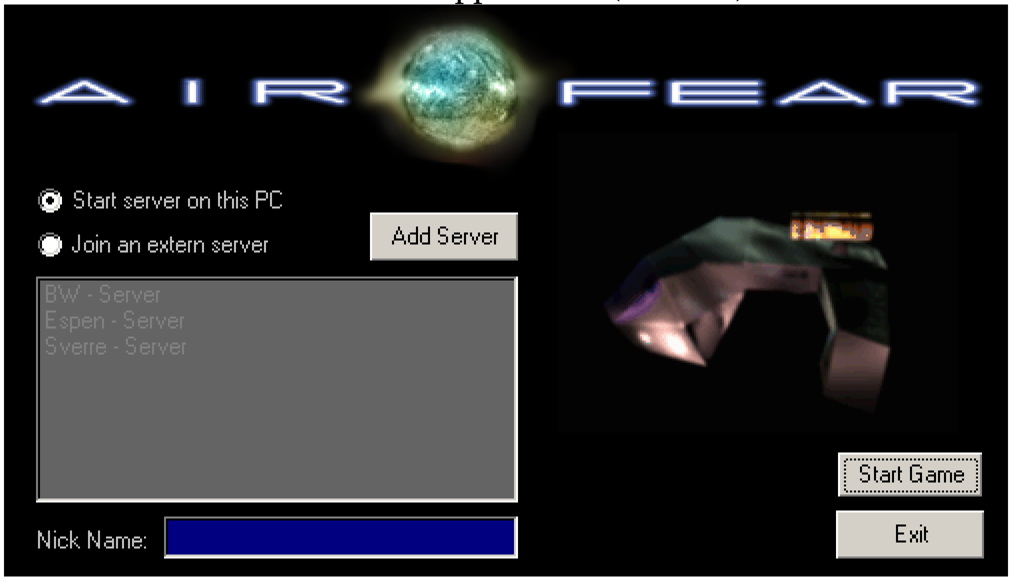

MUDDY: COMING BACK TO GET YOU.
VULCAN CANNON!
MUDDY: COMING BACK TO GET YOU.



3.3 GUI

Today's game's does not use standard GUI's. Instead the developers try to make as nice looking GUI as possible. The AirFear-team has based some parts of its application's GUI on the standard, exported by the common file framework [6] and recommended by Microsoft, while some parts have been custom made. Remember that the games that could be driven with our engine, is not meant to be in a desktop window. Although this is fully possible, the users will probably prefer full-screen modes. The front-end on the other hand, is a desktop window.

Screen-shot of the Front-end application (window):


Start server on this PC
Join an extern server
BW - Server
Espen - Server
Sverre - Server
Nick Name:

3.3.1 Screen Resolution
The game-engine supports every resolution the graphics-card is capable of displaying.

3.3.2 Functionality / user-guidance
The test-game for our game-engine is of type “instant action”. This means that one’s the user enters the game, she will find her self in the middle of a war-fare.

Pressing F2 leads to the screen-parameters setup-screen. Here the user can choose between window-mode and full-screen-mode. She may also choose level of multi-sample. Multi-sample is a heavy process which requires state of the art graphics accelerators. If the frame-rates are going down the basement, lower the level of multisampling. If the user’s PC got more than one graphics-card, she might want to choose which one to use. This is solved by a Combo-box. An option which should not be used by the in-experienced user (although it’s not critical) is the rendering device option. This is only meant for testers of the engine to measure frame-statistics. The game-engine will always search and find the best available rendering-device.

Pressing F3 leads to the controller setup. Here the user might redefine keys, or choose to use a game-controller. How to assign buttons will be obvious to the users, as there is continuous user-guidance during each configuration-screen.

There are some game-functions that should be mentioned:
- Rotate Left
- Rotate Right
- Rotate Up
- Rotate Down
- Accelerate
- Fire
- Stats Switch

Rotate Left:
Makes the users space-craft rotate left.

Rotate Right:
Makes the users space-craft rotate right.

Rotate up:
Makes the users space-craft rotate up.

Rotate down:
Makes the users space-craft rotate down.

Accelerate:
Makes the users space-craft accelerate forward.

Fire:
Fire a bullet in the direction of the user's space-craft-orientation.

Stats switch:
Turn on and off the status-information. This information contains info like "Frag-Count" and "Suicide-Count" of each player in the session.

Chapter 4

4 Design

The main features of the system are described. This chapter shows how the group has decided solve the demands specified in chapter 2. The architecture of the system is showed. This is the chapter where the reader should get an idea of how the different modules in the system are connected to each other.

4.1 System architecture

The system will be described part by part. All the modules will briefly be explained, and the hierarchy of the game-engine showed. Coders should make mind-prints especially of the module-hierarchy. The system architecture has been a major challenge, since we knew very little about game-programming. Many games have lapsed due to bad architecture. "Blade of Darkness" is a game that was developed for three years. At this time, the game-coders discovered that the game-engine was to poorly designed, and they had to start all over. This is a major problem that concerns all large projects. Therefore, before we were dived too deep into the code, all the members of the team were involving a serious brain-storming. Producing thoughts concerning all necessary modules/features that might occur in our project as well as modules/features that the project should be able to be expandable with. An example of such a feature would be Multiplayer.

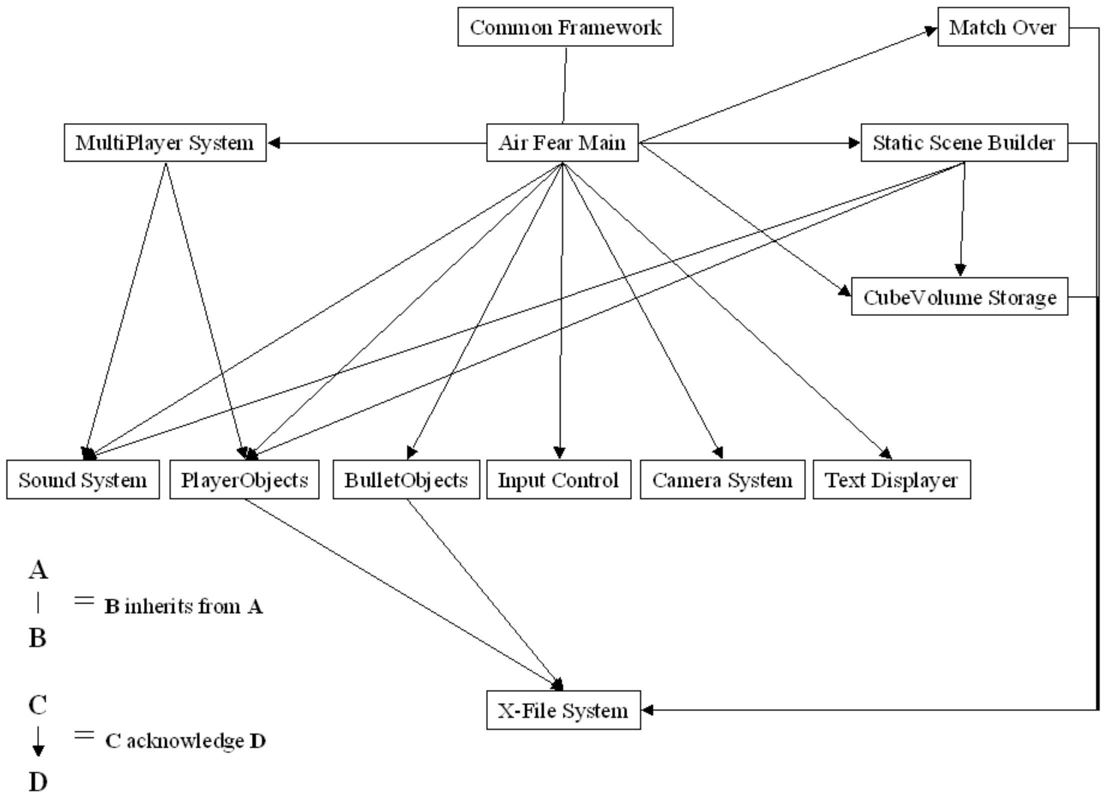
4.1.1 General

The systems architecture has been very little modified during the project. The process of expanding the game-engine, have gone more or less without jar. Most of the technology used, are state of the art. Every added feature was firmly tested before sewed together with the infrastructure.

The game-engine uses Microsoft's Common Framework. This API is recommended by Microsoft to use as a fundament of every 3d application. This framework contains empty methods that describe the various stages that a typical Direct3D application goes through. It contains methods like InitializeDeviceObjects()[3], FrameMove() and Render().

One of the things we have prioritized, are the division between the various modules. Not only are these going to be separate, but the communication between them should be as distinct and fast as possible. To make the modules communicate as fast as possible, we have decided to make a flat structure of the modules that are most active during the real-time running.

Here is a drawing, containing all modules in the game-engine:



Note: Not all modules are in this sketch.

4.1.2 Server [5]

The server is only a single module. Therefore, to make it as fast and optimized as possible, it's been designed mostly in C (not C++).
The structure of this application lays in the organization of the functions.

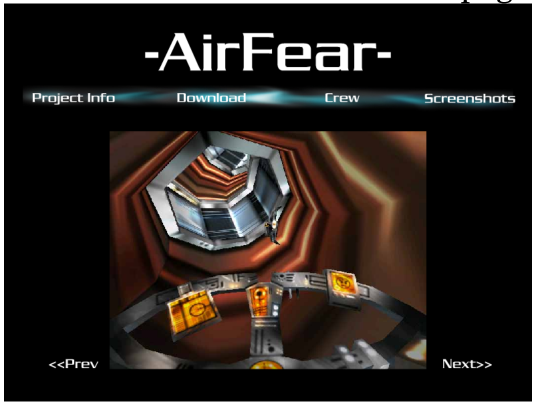
It consists of several queues, taking care of incoming and outgoing network packages. To meet the demands of high performance real-time, the server is capable of sending both UDP and TCP packages. This laid in structured functions making sure the programmer won't be confused when later expanding the functionality.

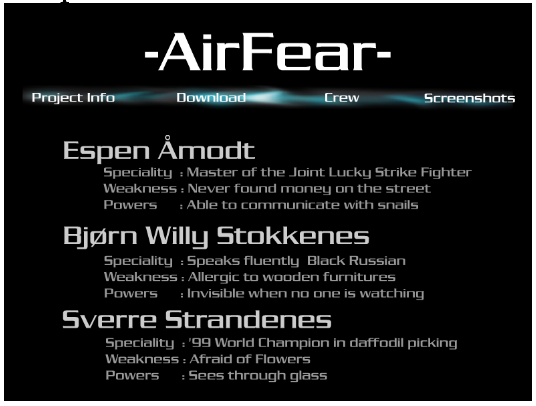
4.1.3 Web Page

We got two web-pages for the project. The first web-page uses the standard template to HIG, and one page which are more game-related. The game-related page is made in Flash. This page has a bit humor, and is designed to be nice-looking and to make the watchers want to download the game. This page shows screen-shots, crew-members, and more. From this page you may download the latest version. Note, we have not yet decided the URL, but http://airfear.stokkenes.com will probably always be intact.

Some screen-shots from the web-page intro:

Some screen-shots from the web-page main part:




-AirFear-

Project Info Download Crew Screenshots

Espen Åmodt
Speciality : Master of the Joint Lucky Strike Fighter
Weakness : Never found money on the street
Powers : Able to communicate with snails

Bjørn Willy Stokkenes
Speciality : Speaks fluently Black Russian
Weakness : Allergic to wooden furnitures
Powers : Invisible when no one is watching

Sverre Strandenes
Speciality : '99 World Champion in daffodil picking
Weakness : Afraid of Flowers
Powers : Sees through glass

4.1.4 Choosing the programming language

Upon project initiation we had our eyes on several programming languages. As DirectX was already decided on, we needed a language that would support it.

Our choices were:

- Delphi
- Visual Basic
- C/C++

Delphi:
Delphi would have been a nice choice for its ease of use, but the DirectX SDK documentation wasn't geared towards this language. We couldn't run the DirectX samples that came with the SDK from Delphi either. Choosing Delphi would have made the learning curve much steeper and only one of the team members had any extensive knowledge of the language from before. Because of this we rejected Delphi as the development platform. The reason why the learning curve is steeper for development in Delphi is that every line of code has to be ported from C++ to Pascal. The DirectX's documentation explains only how to connect to the interfaces using C, C++ and Basic. We must mention here though, that we used Delphi to make the front-end application.

Visual Basic:
Visual basic is naturally supported by the documentation since it is a Microsoft product. This is a slow language intended for beginners and geared towards ease of use rather than full control. Neither of us had much experience from this either, so learning it from scratch with a mission to become experts was a bit far fetched.

C/C++:
This became our language of choice because it was supported by the documentation, and it is a known fact that this is the language used in the game industry. It is generally a less forgiving language and harder to debug than the others, but C/C++ gives us all the control we need and the possibilities for writing fast, structured code not hampered by interpreting or unnecessary "invisible" control-code inserted between our lines.

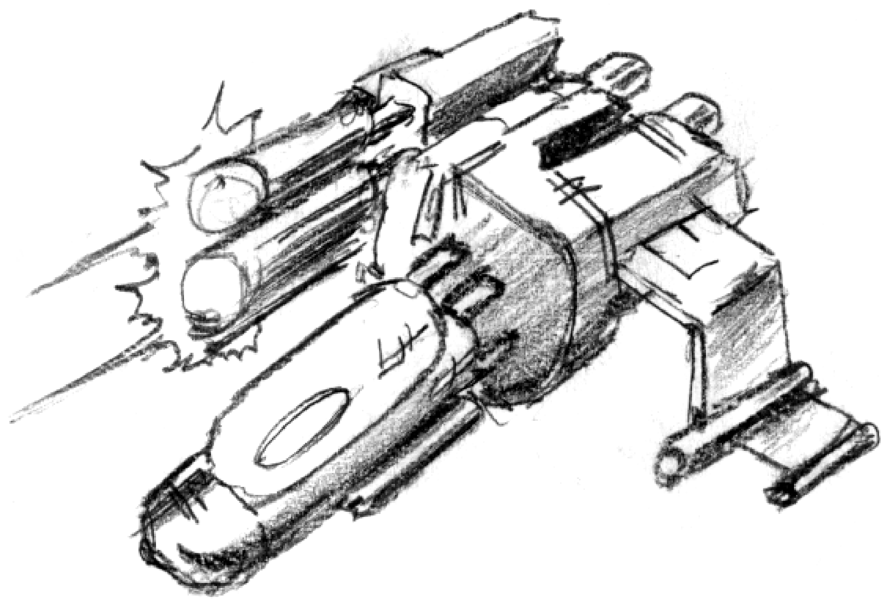
We decided on an object oriented design, so most of the game-engine's code is in C++, but the game server is written in C to make it perform at its best.

4.2 Graphic User Interface

As mentioned under chapter 3, the GUI of a game must not, and should not follow standard guidelines for making a GUI. The ideal design would be an entire new set of widgets tailored to the look of the game, but this has not been a priority. We have chosen to use the standard GUI exported by the common-file-framework, and thus we save time. To test the user-friendliness of some of the screens, we have invited people, not familiar with playing games. These people have tried to setup the server and the client. The user-interface should be intuitive to the average computer user. Linux people might like to fiddle around with our configuration-files as well. ☺

4.2.1 Design of shuttle/spacecraft

This was the first adventure into the 3d-studio object design realm. We wanted to do an object that we had enough skills to model. A few pieces of conceptual art were mocked up.



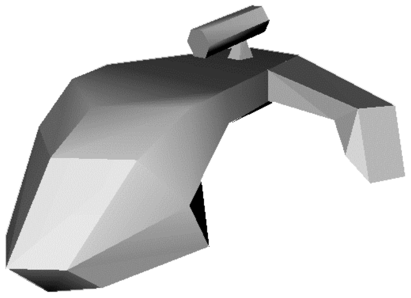
This ship was rejected much because we feared a too high polygon count and it was also complex to model. It looks a bit heavy too.

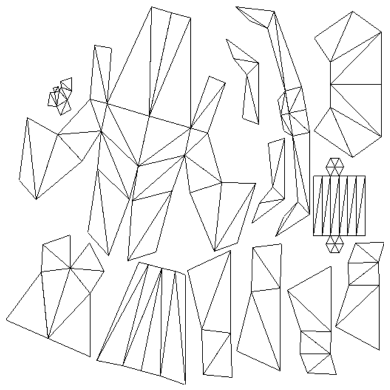
This relatively simple to model ship was chosen because it doesn't require too many polygons and its shape can be nicely approximated by a bounding-sphere (for the collision detection algorithms).

As opposed to rendering animated sequences where the polygon-count can be as high as you wish as long as you have enough memory, a game object requires a low polygon count. This was the first design consideration prior to starting the modeling of the spacecraft. Another important decision we made, was that we only allowed ourselves to use one texture per object because of graphics-card and engine-limitations. Using few textures in a scene ensures less uploading of data between system-memory and graphics-card memory. Such an upload is usually followed by a severe loss of frame-rate which had to be avoided in order to make the game playable. Our objects could not have sub-objects or hierarchies.

The spaceship was carefully modeled using fairly standard techniques in 3d-studio such as moving vertices, extruding and mirroring. The spaceship was formed out of a box primitive with a minimum amount of segments.

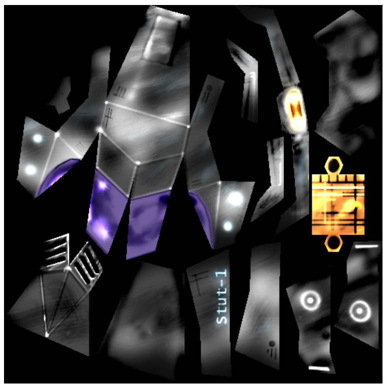
Since we only had one texture-map per object, we needed to merge all polygons into a single object, unwrap it and flatten it. This is a pain-staking process but the end result is worth it. In this way we can add detail to any part of the ship as we want to. The process of un-wrapping the object involves detaching all polygons into sections of the object that do not distort when unfolded. Then the detached vertices of the object-sections are welded together at the edges defining the axes that the polygons will be rotated about upon unfolding. When all of this is done correctly (which took us approximately a day for an object like this) we used a simple script to assist in the following unfold procedure. The following figure shows the initial object to be unwrapped to the left and then the un-folded object to the right. Take a moment to recognize the various parts of the 3d-object that conforms to the flat images to the right.



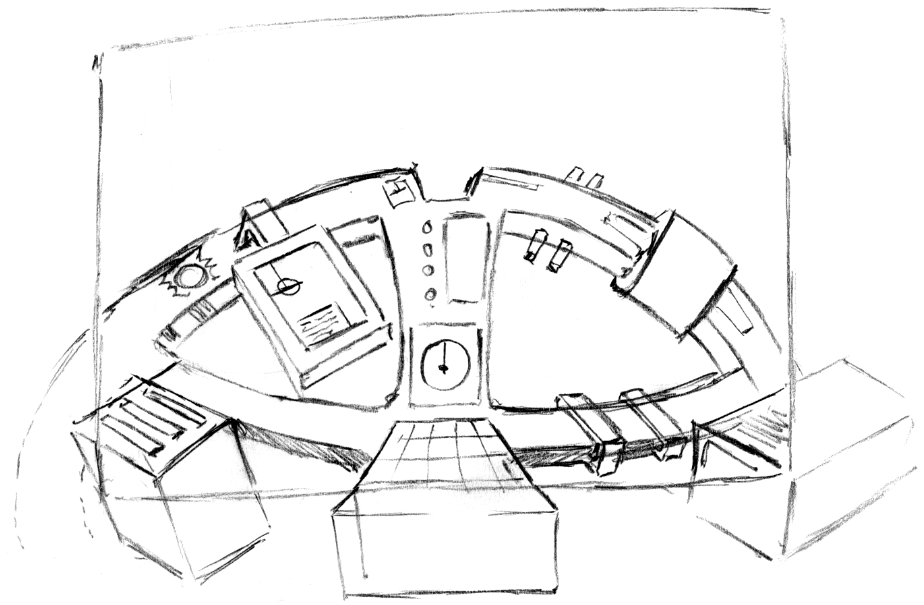


When the unfolded flat object is done, texture coordinates are assigned to each vertex. The result is an object with its texture coordinates laid out in a 2dimensional fashion whilst the object itself is 3dimensional.
After this, an image of the flat object is exported which corresponds to the dimensions of the texture-map.

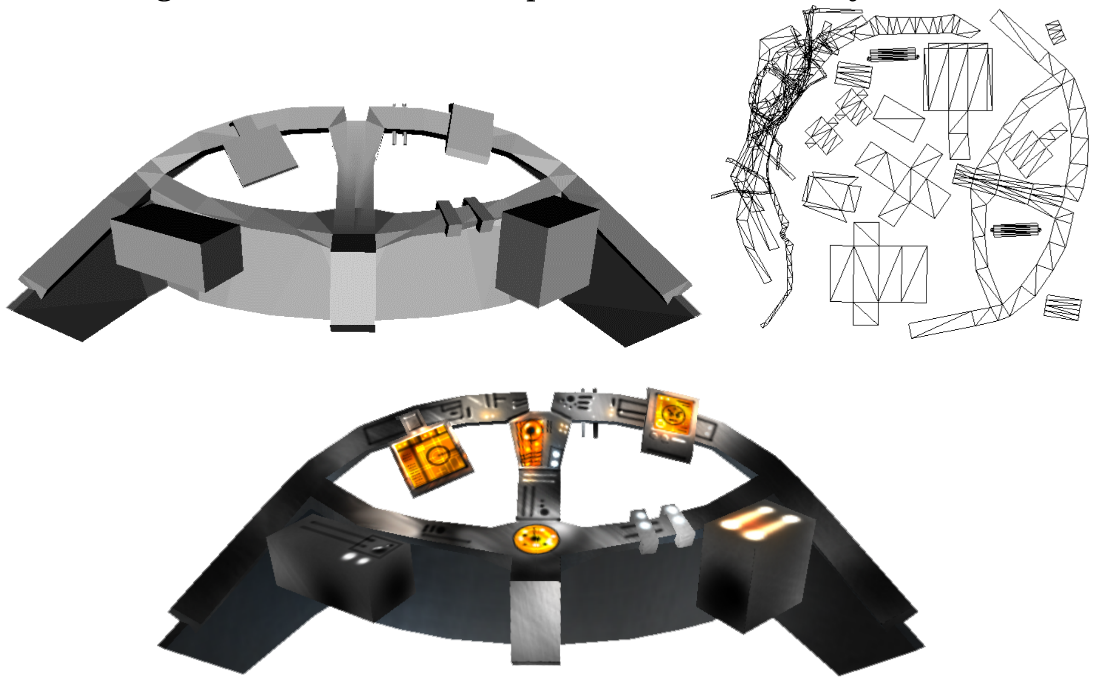
The texture is drawn straight on top of the exported image. The texture may have as high a resolution as your graphics card supports, but keeping is at small as possible saves memory and is the rule of thumb. Textures sizes should be powers of two to avoid expensive multiplication and division instructions on the GPU and replace them with fast bit-shifts instead. Our texture sizes range from 32x32 to 512x512. On some older graphics cards, texture-sizes must be quadratic, but this is usually not the case on new hardware. Below are the finished texture and the complete spaceship.



The same process was repeated on the cockpit object. First some conceptual art was drawn.

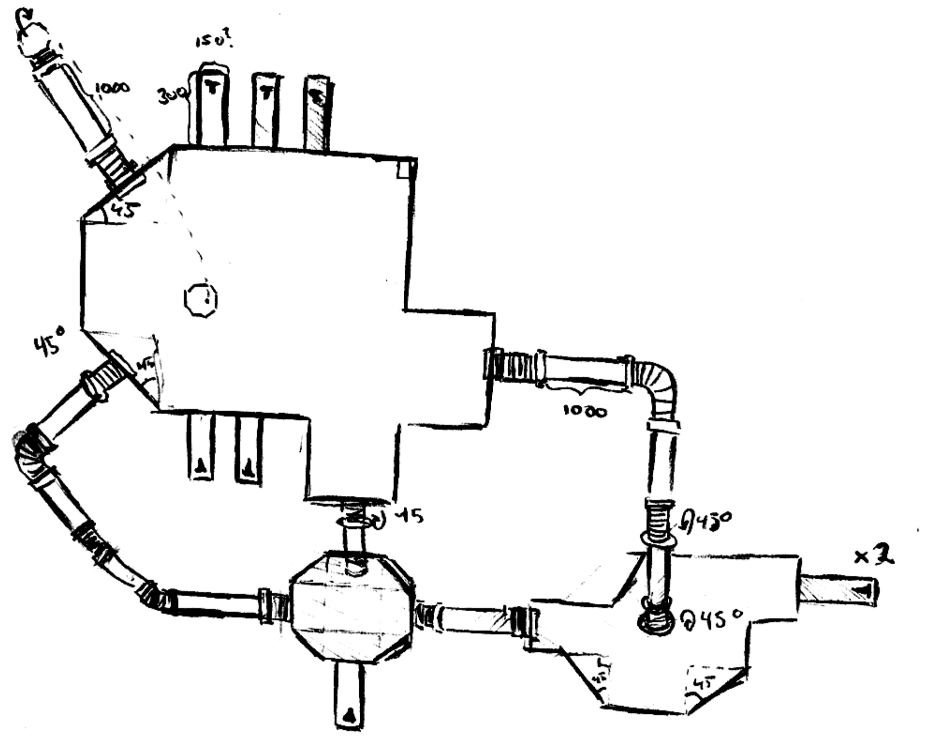


Then the object was made and the same unfolding process was conducted on this. Notice that some parts of the cockpit such as the left panel has been scaled up in the 2dimensional representation, this gives a larger space on which to draw the texture for this part of the object and allows for more detail. The figure below shows the steps and the finished object.



4.2.2 Design of level/area of play

Before we started modeling we laid out a top-view of how we wanted the scene to be. Some estimates of measures and angles were made.

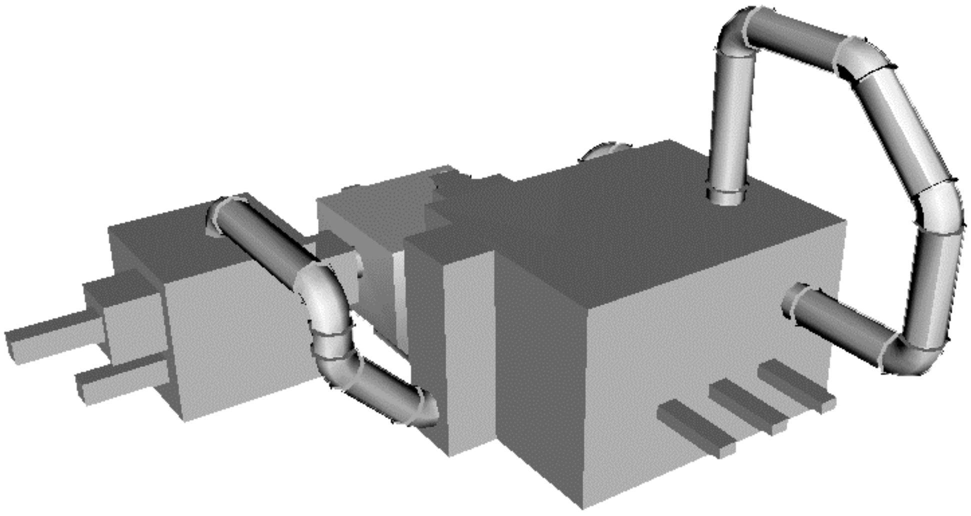


The level was made along the same guidelines of the spaceship. We wanted to have as few vertices as possible and a minimum number of textures to conserve graphics memory. The level is made up of a number of objects each unwrapped and textured in the same manner as the spaceship. A problem arose while positioning the objects. We found out that global scaling, positioning and rotation information was removed by the Conv3ds utility upon converting the 3ds format to the X-File format.

The solution to this problem was to never touch the pivot-points of the objects or move them in normal object-selection mode. To get the right positions and alignment, only the vertices could be modified as then the data would be stored in the mesh's vertex data rather than in an external transformation matrix that could not be exported by Conv3ds.

When large portions of a level had to be moved in to place, we wrote max-scripts to automate the slow process of selecting one object, converting to editable mesh, do a relative translation and finally deselect and move on to the next object.

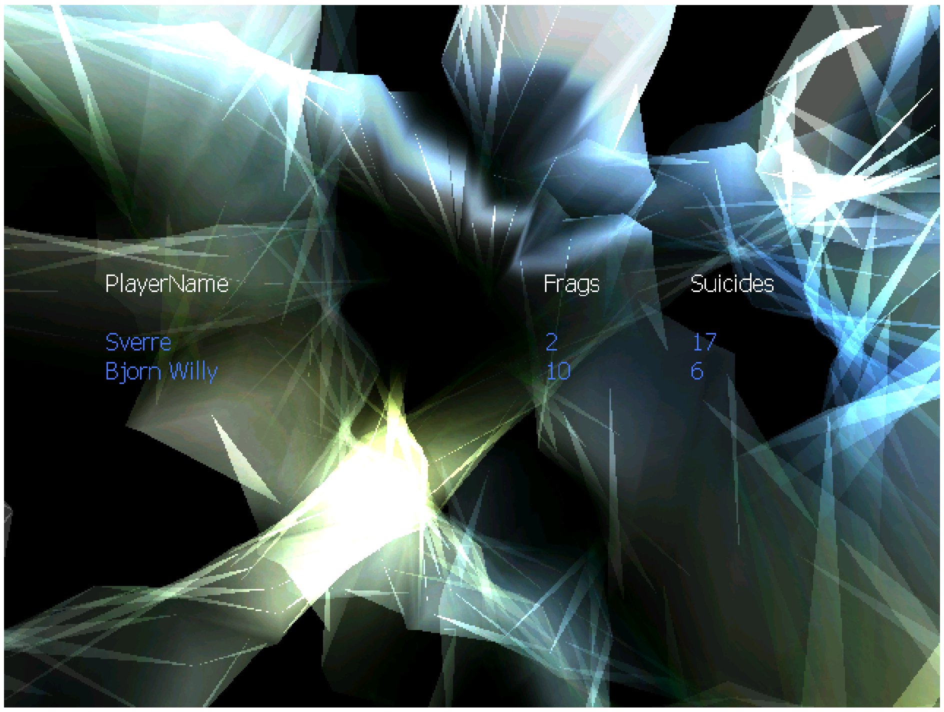
We could not use all modeling techniques while designing the level, because some of the techniques generated meshes with some unknown artifacts that didn't correspond to the standards we designed our Cube-Volume class after. Splitting a set of polygons for generating more detail in a certain area could not be used. Below is an outside view of the demonstration-level rendered without textures.



We decided to make a space station which is closed so that the freedom of the players is limited. This is to keep the battle inside a reasonable space so that frequent encounters between players are more likely.

4.2.3 Design of victory/loss sequence

As a reward to the winner and an insult to the loser, a match over part was designed. It consists of some high-detail half transparent graphics in the background coupled with a soundtrack and a sample telling you whether you lost or won. The statistics of the match is shown so that people can see how they did in the match. After a set period of about 30 seconds the match restarts. This is server-synchronized.



PlayerName	Frags	Suicides
Sverre	2	17
Bjorn Willy	10	6

Chapter 5

5 Implementation

This chapter contains information mostly for the programmers. Some of the source is showed, and the most important algorithms are explained. As said, this chapter is meant for people with intermediate to advanced programming skills.

5.1 Different tools used

The program-packages used to develop this game:

- Visual C++
- 3D Studio Max
- Adobe Photoshop
- Conv3ds
- X-File Mesh viewer
- Microsoft Word XP
- Microsoft Project 2000

5.1.1 Reasons for choosing these tools

Visual C++:
We chose to use VC++ because DirectX itself has been written in it and it contains application-wizard that sets up the DirectX Common File Framework. It also has a great editor, interface and debugger.

3D-Studio Max:
3D-Studio is widely used in the games industry to model low-polygon meshes. It was a natural choice because Bjørn Willy had taken the 3D-animation course at HIG and learned how to model in 3D-studio during that course. A utility comes with the DirectX SDK that can convert 3ds files into the X-File format used with DirectX Graphics.

Adobe Photoshop:
Photoshop was chosen because of its ease of use and great layer handling. An alternative was Gimp, the GNU Image Manipulation Program, but we didn't have much experience using the program.

Conv3ds:
A utility that comes with the DirectX SDK distribution intended for converting the 3ds-format into the X-File format. This naturally came in handy. The alternative to using conv3ds would be to write our own 3ds converter, which is beyond the scope of this project.

X-File mesh viewer:
This is another handy utility that came with the DirectX distribution. Used for previewing objects while drawing textures prior to inserting them into the final scene.

Microsoft Word XP:
This became our choice of software for writing the report. We considered LyX as an alternative, but neither of us had any time to learn it. Word has its awkwardness and a mind of its own (sometimes an evil mind), but makes up for it in ease of use and functionality.

Microsoft Project 2000:
A utility used for creating this project's Gant-Diagrams.

5.2 Principals we have followed

We have been studying as a close team, making rock-firm positions in the group. Instead of rotating the tasks among the group-members, we needed a fast and robust working method. There could be no time for practising, and we had to do it right from the beginning. Here are some of our principals when coding:

- Large variable- and function-names, which needs little or no explanation.
- Return using HRESULT, which makes us able to debug easier.
- Class-names start with C and have a capital letter at every word describing the class.
- All variables start with none-capital letter, and have small letters except for the beginning of every new sub-word in the variable name.

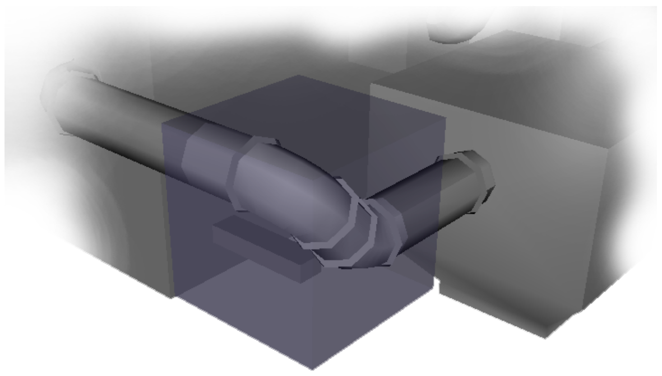
5.2.1 Scene partitioning

Although the computers of today are getting more and more powerful in the graphics-rendering capability-stakes and in the processor-power department, they are not all that powerful. When rendering a large scene, there's no point in wasting rendering-power on geometry not visible on screen. The same goes for collision-detection-testing which is a slow and heavy process. There should be a way to limit rendering to the geometry surrounding the camera, and likewise a way to limit collision-testing to the nearby surroundings of an object.

To achieve this, one has to partition the scene into smaller manageable chunks. One solution is to partition all the faces of a scene into a tree-structure, arranging the faces in nodes according to their position in relation to a plane such that they're locations can be identified on traversing the tree.

This approach is the one used by the BSP (Binary Space Partitioning), quad trees and oct-trees.

We chose to go for a method of dividing the scene into a grid of cubes. Each cube contains a list of faces that are contained in a specific area of the scene. The image below shows one such cube that contains an area of the level inside.



This method eliminates the search-time required for the methods mentioned above. Using this way we can do a simple conversion from world space to cube-space (or grid-space if you like), and index directly into an array. The downside is that this technique requires more memory when using enormous scenes. Take for example an adventure game, where the levels require being of extremely large scale, tempting to represent an entire world with multiple cities etc. In our case, we need only fighting arenas, designed to ensure frequent encounters between players. This is the reason we made the choice of fast accessing, instead of memory conserving techniques that may reduce performance. Our entire test-level that has an appropriate spatial size for about 6-8 players, uses only 5 megabytes of space which is suitable for our computer-requirements. A modern game typically uses combinations of a wide array of methods to get the best possible performance versus memory usage. These methods are not available to the public and are considered to be cutting-edge.

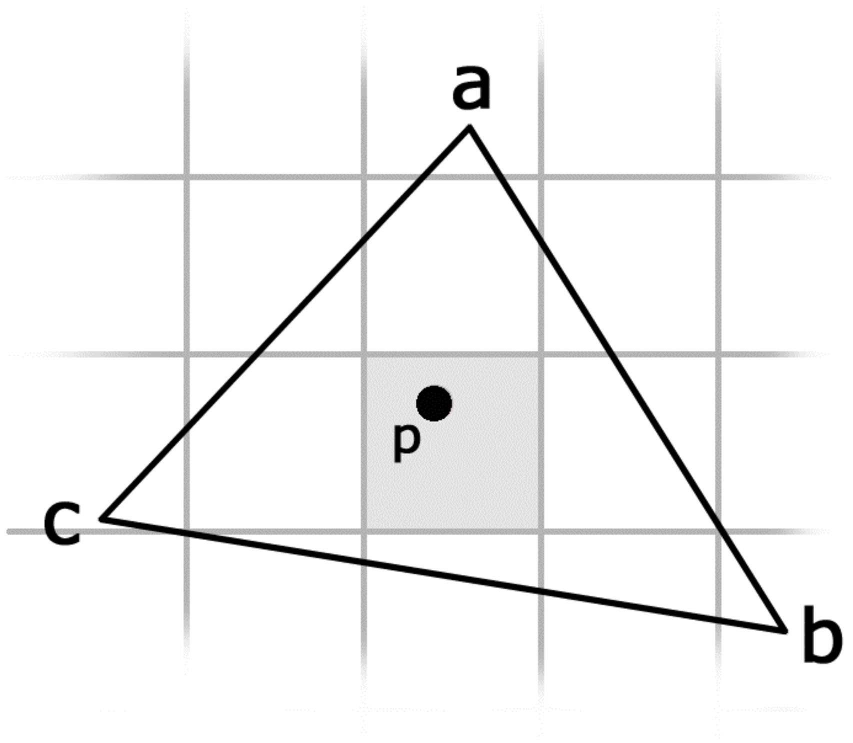
5.2.2 Determining what faces should be in which cubes:

Two techniques were used to group faces inside boxes depending on what the cubes should be used for.

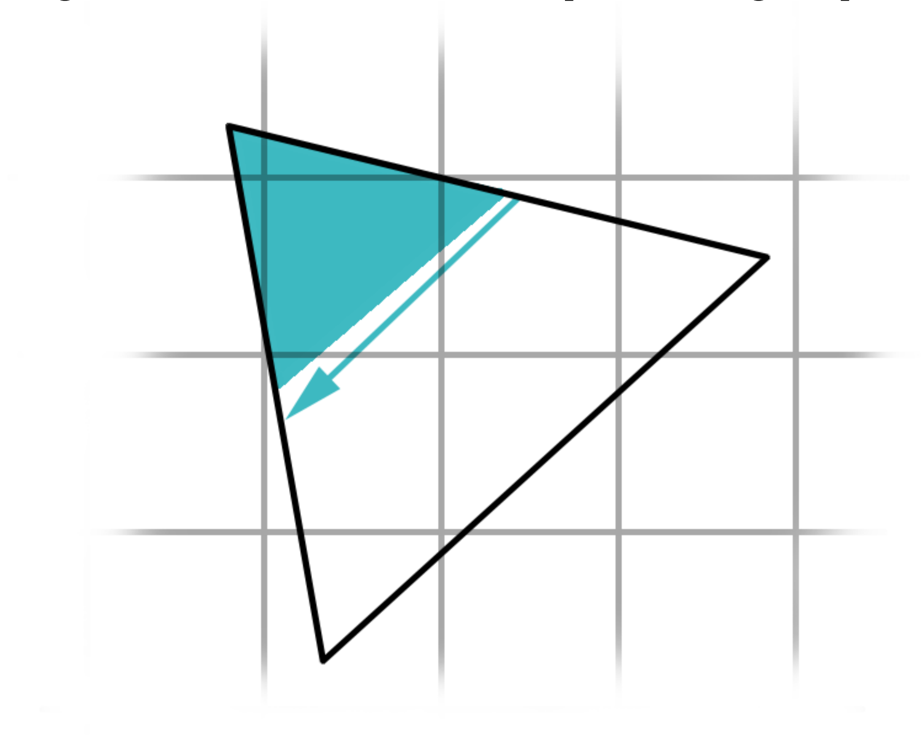
The two cases are:

- Graphics cube volumes
- Collision cube volumes

Graphics cube volumes:
In the case of the graphics-cube volumes, a face was only to exist in one cube at a time. This is because the graphics delimiting algorithms choose a larger group of cubes for rendering. If a face isn't registered in one cube, it's likely that it's registered in one of the neighbouring cubes instead which will in many cases be drawn as long as it is within the sphere-bounding volume (see 5.2.2). The faces intended for the graphics-cube-volumes were grouped according to their centre point. This is a very fast computation, and since one point can only reside in one cube, we avoid the cases of the same faces rendered more than once.



Collision cube volumes:
When testing for collision, it's important that all faces that span a cube are present. The technique of grouping faces using the face centre as reference point isn't sufficient since the face might span several cubes. In this case we chose to interpolate through the surface of each triangle in small steps to ensure that the face is recorded into all cube-volumes it spans. This is a very time-consuming process if there are many large faces in the scene. So time-consuming in fact that we still haven't interpolated at highest precision.



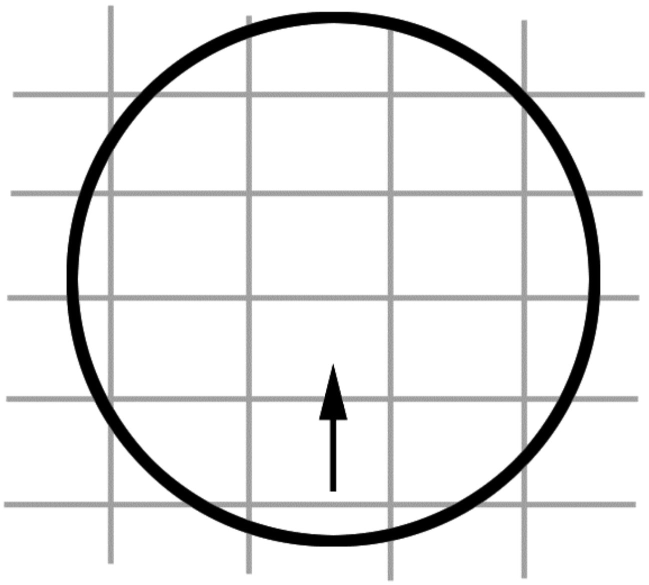
Note that the face will be recorded into all cubes that the triangle spans.

5.2.3 Delimiting the scene render calls:

For the graphics, a large cube-size was used. This reduces the amount of render-calls per frame. Rendering large amounts of faces using one call is faster than rendering the same amount of faces using multiple calls.

Selecting which cubes to render is determined by the position and orientation of the camera. There were several techniques to consider. One solution would be to approximate the camera's view cone using 4 planes to describe the limits of the view-port, and then clip each cube-volume against these planes to determine what cubes are visible. One would also need a fifth plane to limit the cube-rendering on the local z-axis of the camera. Some games use an infinite view these days, but doing so, could be hazardous to speed unless you had support for progressive meshes with a variable level of detail. This is far out of scope of this project, and could be considered a project in itself.

We chose to approximate the view-port using a bounding-sphere with a slight offset from the camera on the local z-axis. The figure below shows the cube-volume grid, the camera (arrow) and the bounding sphere.



All cubes that are inside the sphere are rendered. This proved to be an appropriate solution for this project as an infinite view wasn't needed.
The test to determine whether a cube is inside the sphere is relatively simple and has low computation costs. Note that this is only a 2d drawing, showing a circle, and not a sphere. Some of the cubes drawn would not always be visible to the user. Again, this is another performance issue. We could easily have made a stronger interpolation algorithm, making sure that only the cubes that are visible to the user was drawn. This is not done because today's GPUs are so fast, that the extra time spent waiting for the CPU is not worth it.
To hide the artefact of geometry suddenly popping up in the far distance, fog is drawn. The fog also helps to add atmosphere to the scene.

5.2.4 Collision-detection-testing:

Collision detection is the test of whether one entity hits another entity.

In this game we support a range of collision-detection tests:

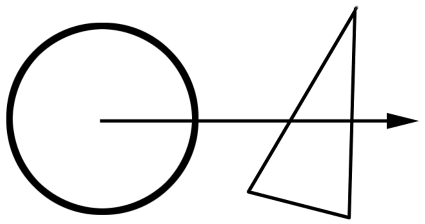
- Player crashes with geometry
- Bullet crashes with geometry
- Bullet crashes with a player
- Player crashes with another player

In a full-scale 3d animation package, collisions are to be precise and likewise the response to the collision needs to be believable and as close to natural as possible. This takes huge amounts of processor-power and can take several hours to calculate. In a game these calculations need to be simplified down to a level where the computer can manage to calculate the collision and response in real-time. A mesh may consist of 100s of faces that need to be tested against another mesh's 100s of faces. This is not yet possible to do in real-time on the average PC.

There are several approaches to simplifying the testing of collision. Most techniques involve enclosing the mesh in a mathematically defined volume such as a bounding box or a sphere. The more advanced solutions involve enclosing the mesh in a hierarchy of multiple bounding volumes in order to get a sufficiently accurate shape to test for collision.

We chose to approximate the players craft using a bounding sphere enclosing the mesh. The bullets were approximated by simple points. The geometry is treated as is, without any approximations.

Player crashes with geometry:
When taking the approximation information above into account, the test of whether a spacecraft hits a wall boils down to whether a given bounding-sphere has intersected a triangle. To further simplify the test, we scale the sphere down to a unit sphere. Now, this will only work if we scale the world surrounding the player accordingly. So now we have reduced the problem into a test whether a unit-sphere intersects with a triangle. We must also take into account the player's velocity vector. We get a line-segment that is the player's velocity-vector and the player's position as origin.



The test whether a unit sphere with a velocity vector intersects a triangle, can be broken down into the following steps:

- Calculate the plane the triangle lies on.
- Calculate the intersection point on the unit sphere where it will hit the plane.
- Calculate the point on the plane where the unit sphere collides.
- Check if this plane intersection point is inside the triangle. If not, calculate the triangle intersection point (if any).
- If we hit the triangle and the distance to the intersection is less or equal to the velocity-vector, we have a collision.

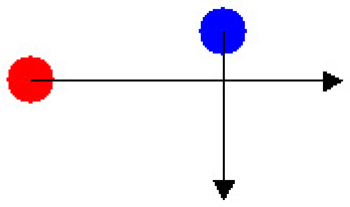
These are the basic steps, but each step involves a set of computations to handle special cases involved in the test. We want to give people the basic idea of how this works, but will not go into great detail in this part of the report.

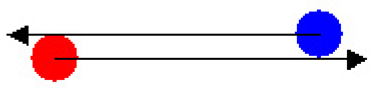
Bullet crashes with geometry:
Bullets are treated as points with a velocity vector. The testing is much like the test done for the "player crashes with geometry" case, but since we are working with a point instead of a sphere we can make this test simpler. This algorithm is faster, and needs to be so, since there are far more bullets flying around in the scene than there are players.

The test itself boils down to testing whether the velocity vector originated from the bullet's current position intersects a triangle. We shoot an infinite ray with the same direction and origin as the bullet and check if this intersects a part of the geometry. If the length to the intersected triangle is smaller than the length of the bullet's velocity vector, then the bullet will hit the geometry. If it is longer, we miss the geometry.

Bullet crashes with player:
Bullets are treated as points with a velocity vector. Players are treated as spheres with a velocity vector. We need a test to check whether a line-segment (the bullet) has intersected a sphere (the player).
We shoot a ray from the bullet and check the distance to the intersection point with the sphere (if any). If this distance is shorter than the length of the velocity-vector, the bullet has hit the player. One important thing to remember is that both the bullet and the players are dynamic moving objects. Therefore we need to subtract the player's velocity-vector from the bullet's velocity-vector prior to running the test. Having done this, we can treat it as a dynamic point against a static sphere test.

Player crashes with another player:
This boils down to a test whether one sphere intersects another. We measure the distance between the spheres. If this distance is smaller than the sum of the two spheres' radiuses, we have a collision. There is another problem to this which complicates the testing a great deal. Both spheres are dynamic objects. Take a peek at these sketches to get an idea of the problem:



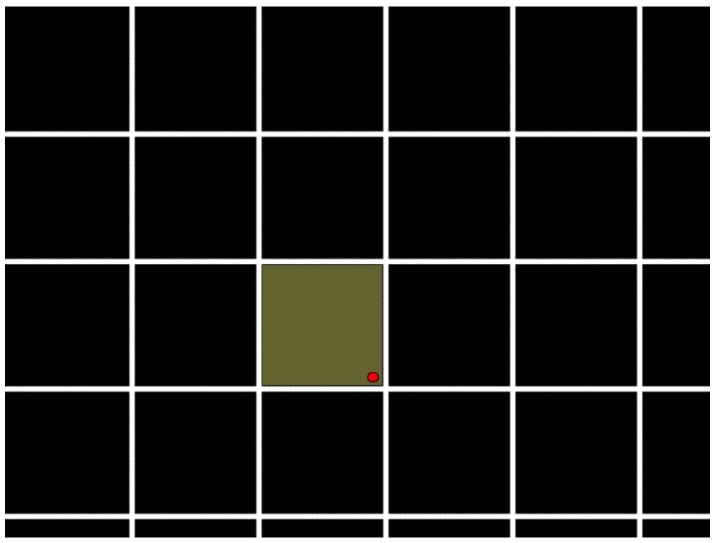


As you can see, we must consider the velocities as well as radiuses. The solution to this is to subtract one sphere's velocity-vector from the other sphere's velocity-vector prior to running the test. Having done this, we can treat it as a dynamic sphere against a static sphere test.

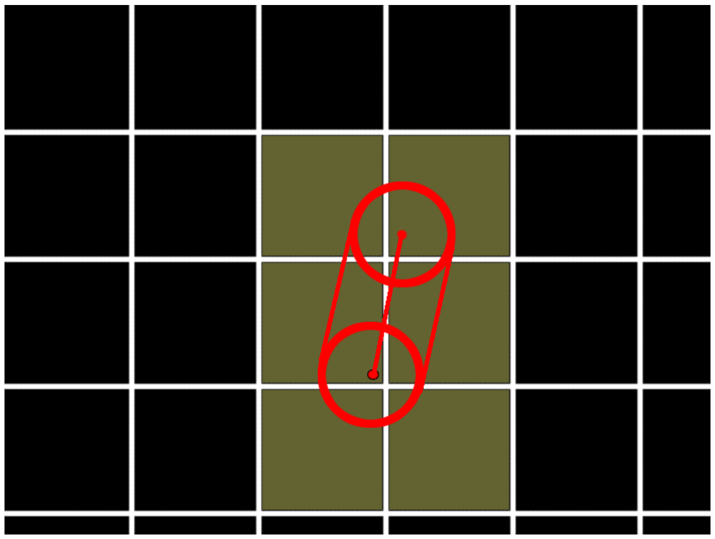
"Player crashes with another player" gives confrontation to another problem. What if the position for a player is not correct at a client? What happens if player1 detects collision with player2, and player2 don't? This may cause serious errors in the game. Therefore another algorithm comes into play. This algorithm handles the delimiting of collision against player detecting. Instead of making all players test against each other, we make sure that no duplicate testing is happening. The problem boils down to an algorithm which evenly distribute the collisions to each client in the session. When a collision has been detected, a TCP packet is being sent to the server. The servers updates the stats of the players involved, and sends a TCP packet to every client on the network with the information of that an explosion should happen on a specific player. This eliminates the duplicate testing problem as well as speeding up the collision-detection.

5.2.5 Collision-detection delimiting (this hurts)

When is comes to collision-detection concerning "players-against-geometry" and "bullets-against-geometry", there has to be some sort of function or algorithm that makes the harsh drudgery of finding the correct cubes to test within. Again, we show a 2d-grid and not 3d, since 2d is most suitable on paper.



Notice the small red dot. Think of this point, as the point where a space-craft (using bounding-sphere) resides. This makes us think, that only the marked cube is only to be tested within, but this is not the case. There are two more factors to consider; velocity and radius of the bounding-sphere. Let's see what this leads to.



Now we can see that the delimiting algorithm must be carefully designed to not miss any cube-volumes. The algorithm first gets the critical walls, where the radius of the bounding sphere is close enough to need collision.

We implemented a small function in the CubeVolumeStorage-Class to do this job:

```
BYTE CCubeVolumesStorage::FindCriticalCubeNeighbours( D3DXVECTOR3 position, FLOAT radius,
INT colCubeIndexX, INT colCubeIndexY, INT colCubeIndexZ ) {
    BYTE  criticalWalls = 0;
    DWORD tempDist;

    tempDist = ( (INT)position.x + (SceneLengthX / 2) ) - ( UnitsPerCube * colCubeIndexX
);
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_X_NEG;
    tempDist = abs( ( (INT)position.x + (SceneLengthX / 2) ) - ( UnitsPerCube *
(colCubeIndexX+1) ) );
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_X_POS;

    tempDist = ( (INT)position.y + (SceneLengthY / 2) ) - ( UnitsPerCube * colCubeIndexY
);
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_Y_NEG;
    tempDist = abs( ( (INT)position.y + (SceneLengthY / 2) ) - ( UnitsPerCube *
(colCubeIndexY+1) ) );
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_Y_POS;

    tempDist = ( (INT)position.z + (SceneLengthZ / 2) ) - ( UnitsPerCube * colCubeIndexZ
);
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_Z_NEG;
    tempDist = abs( ( (INT)position.z + (SceneLengthZ / 2) ) - ( UnitsPerCube *
(colCubeIndexZ+1) ) );
    if( tempDist <= radius ) criticalWalls |= CRITICAL_WALL_Z_POS;

    return criticalWalls;
}
```

The delimiting algorithm can use this function to determine the outer walls, and simply fill inn the empty cubes between. To make the engine run robustly, the algorithm also make sure no cubes residing out-of-bounds is being sent to the collision-detection module.

5.2.6 Multiplayer network traffic

The network traffic in a real-time game-session has to be as low and smooth as possible. To make this happen, we have made support of types of packages; TCP and UDP. TCP is sent when things are critical, like when a player hits the geometry, while UDP is sent when data is none critical, like when updating positions of the space-crafts etc.

5.2.7 Multiplayer movement prediction

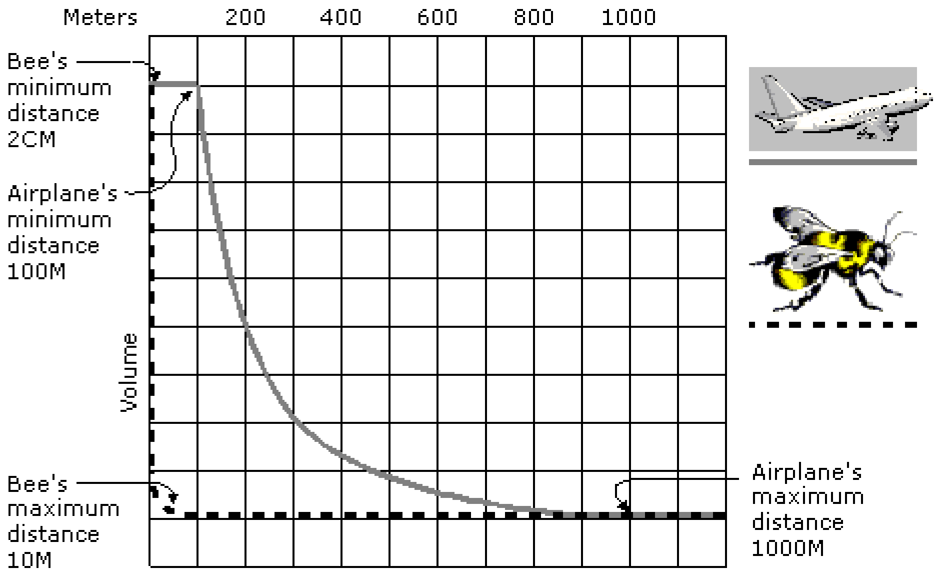
Implementation of moving objects in a network-game might seem straightforward. Obviously one needs position and alignment information for each object at all times. But this is not physically possible because of limitations in bandwidth. The frame-rate of a smooth running game depends on the processor and graphics card and can be anything between 15 fps to 500 fps. Frame updates are dynamic and the idea of sending a packet every frame is not an option at all. Instead one must send packets at regular intervals that do not generate too much traffic on the network. The AirFear game-engine sends positional-data only 5 times pr. second. To cover up for the lack of position and alignment information between the packets we chose to linearly interpolate (LERP) the player-transformations based on the input information received in the most recent transformation packet. To do this, one must ensure completely synchronized timers on each machine, as the frame-rate may vary from machine to machine. Our engine uses Windows multimedia timers that are considered to be the most precise timers on the platform. One must also provide a numbering of the packets in case they arrive in different order, which is very common while playing over the internet. These packets are sent using the UDP protocol to ensure small packets and little overhead. It is not critical if a packet gets lost or corrupted and so there is no need for resending. If an “old” packet arrives after a new “packet”, it is simply discarded.

The values interpolated are the rotational velocities, acceleration and positions. Each time a new packet arrives, the values are refreshed with the exact transform of the current player. As long as the player doesn’t change his input between two packets, we get a completely smooth movement without butchering the network.

Bullets are handled in a slightly different manner because of their completely linear movement, packets are only sent once when the player fires a bullet. After this no further correction is needed.

5.2.8 Sounds

To make sure we could easily assign sounds to the scene, we have implemented 4 classes in the sound-module. These classes are handling stereo-sound, 3d-sound, manipulation of sound-data, and playing of sound-data. As with the graphic and collision-detection (and response), we must make sure that the sounds outside a certain distance is stopped playing, so that the sound-processor isn't asphyxiated. This is done, by making the level-designer specify a sound max-distance. Outside this max-distance, the sound won't be played. When max-distance is mentioned, let's also say some words about min distance. This is the distance to where the sound should start decaying. The following illustration shows how minimum and maximum distances affect the loudness of a jet and a bee at increasing distances.



To specify information for the Doppler, the level-designer has to inform the system about the velocities for the different sound-objects in the scene. Normally, these objects would be static, and therefore the velocity is 0. Note also that these parameters are only needed for 3d-sounds.

As the camera moves around in the scene, the listener-source has to be moved accordingly. This is what we in the 3d-sound-system refer to as the listener. To make it easy to update the data from the main-loop, we made a function for each parameter-setting in the 3d-sound-system. To further speed up the updating, we use a feature in DirectX to update all the settings only, when all parameters are ready. This increases the performance in the sound-card-processor radically.

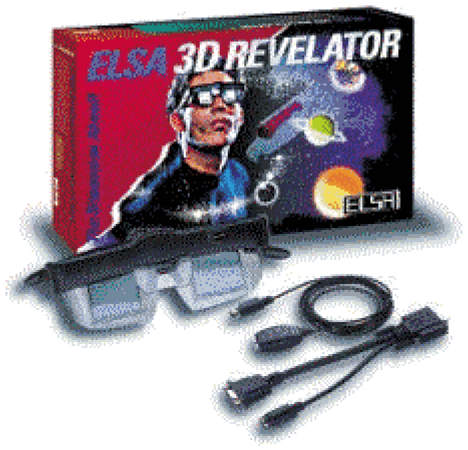
Here are two example-functions on how to update the listener:

```
HRESULT C3DSoundControlSystem::Set3DListenerVelocity( D3DVALUE XVel, D3DVALUE YVel,
D3DVALUE ZVel ) {
    HRESULT hr;

        if( NULL == g_p3DAudiopathMain )
        return ( E_INVALIDARG );

    IDirectSound3DListener* pDSListener = NULL;
    if( FAILED( hr = g_p3DAudiopathMain->GetObjectInPath(
        0,
      DMUS_PATH_PRIMARY_BUFFER,
      0,
      GUID_NULL,
      0,
      IID_IDirectSound3DListener,
      (LPVOID*) &pDSListener ) ) )
        return DXTRACE_ERR( TEXT("GetObjectInPath"), hr );

        if( FAILED( hr = pDSListener->SetVelocity(
         XVel,
         YVel,
         ZVel,
         DS3D_DEFERRED ) ) )
      return DXTRACE_ERR( TEXT("SetVelocity"), hr );

    SAFE_RELEASE( pDSListener );

    return (S_OK);
}

HRESULT C3DSoundControlSystem::Set3DListenerPosition( FLOAT fXPos, FLOAT fYPos, FLOAT
fZPos ) {
    HRESULT hr;

        if( NULL == g_p3DAudiopathMain )
        return ( E_INVALIDARG );

    IDirectSound3DListener* pDSListener = NULL;
    if( FAILED( hr = g_p3DAudiopathMain->GetObjectInPath(
         0,
       DMUS_PATH_PRIMARY_BUFFER,
       0,
       GUID_NULL,
       0,
       IID_IDirectSound3DListener,
       (LPVOID*) &pDSListener ) ) )
      return DXTRACE_ERR( TEXT("GetObjectInPath"), hr );

        if( FAILED( hr = pDSListener->SetPosition(
                fXPos,
                fYPos,
                fZPos,
                DS3D_DEFERRED ) ) )
        return DXTRACE_ERR( TEXT("SetListenerPosition"), hr );

    pDSListener->CommitDeferredSettings();
    SAFE_RELEASE( pDSListener );

    return (S_OK);
}
```

Note that we always update the position at the end of every update-sequence. That's the reason the last function contains the line "`pDSListener->CommitDeferredSettings();`". This line makes sure every none-committed parameter is being committed and calculated in the sound-hardware.

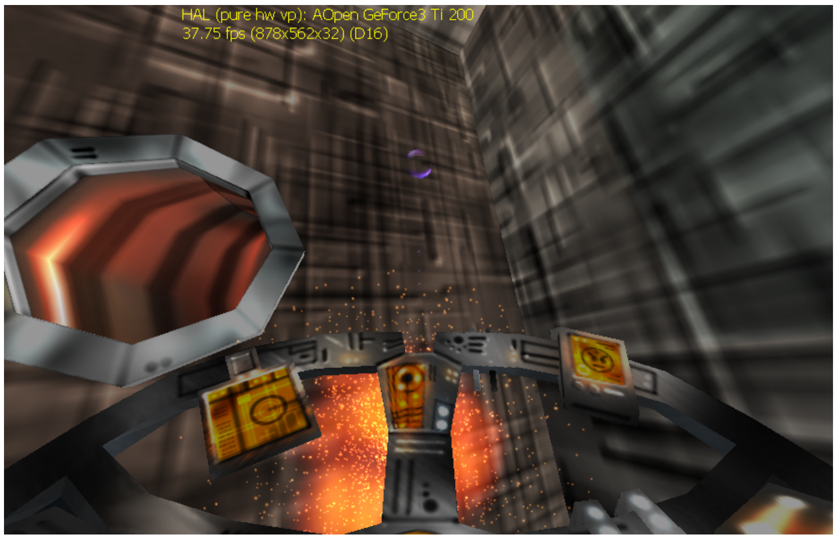
5.2.9 3D Glasses

Since this game is developed in DirectX. Any user that has 3d-glasses could use these to watch our game in 3D. AirFear is very suitable for 3d-glasses, because there are no 2d-objects present. We have tested this, and the results are astonishing. The technique is to make the glasses block first the left eye, and then the right eye. By doing this along with flipping the camera position from left to right according to the z-buffer used in the 3d applications (like AirFear), and synchronize them, we have stereo-vision. Unlike former techniques, this technique actually works.



5.2.10 Explosions

An explosion in AirFear consists of two different effects. A fire/smoke cloud and particles simulating scrap metal from the space-crafts. The particles, is done by using point-sprites. By doing this, we disable the writing to the z-buffer and make the particles semi-transparent. In addition, we make many layered particles seem glowing, by adding the colours from the different portions of the particles together. To make the particles look realistic, we make initializes every particle to a random speed, in a random direction. The most challenging about the explosion, is the flammable sky in the middle of the explosion. Here is a picture, showing clearly the flames in the explosion when a space-craft has been shot:



```
HAL (pure hw vp): AOpen GeForce3 Ti 200
37.75 fps (878x562x32) (D16)
```

Notice that these flames have been modelled in 3d-studio-max, and consists of 21 animated pictures. These pictures is being sequentially showed on a 2d-polygon which is always facing the camera. A polygon that is always facing the camera is called a bill-board. The technique for making the bill-board always face the camera is described next.

- Position the bill-board at a basis position with a basis orientation in the scene.
- Make a normalized vector from the bill-board, out of its normal-side.
- Get the current position of the camera.
- Make a vector from the bill-board basis point to the camera position.
- Normalize the new vector.
- Determine the cross product between the two vectors described.
- Find out the octant in which the camera-point resists, and make the arc-sin value give you the angle of the two known vectors (the arc-sin value is known from the length of the 3rd vector, which is the normal between the two already known vectors).
- Make a quaternion rotation, using the 3rd vector as a rotational axis and the angle already found.

To make things a bit better, we added some extra special cases. To mention one; when the bill-board is within a certain range to the camera, the bill-board stops rotating. This last case, makes the player move trough the explosion, instead of just se that the explosion turns to one side when passing it.

5.2.11 Bullets

To show bullets, we use two semitransparent spheres (with low poly-count), rotating at different velocities. Each sphere has its own texture.

Chapter 6

6 Testing and ensuring the quality

This chapter shows how the tested the system under the development process. We also show how we have handled the quality assurance. A person with no programming skills should easily be able to read this chapter.

6.1 Documentation and versions

This project was built up module by module in object oriented fashion. All the documentation of the classes are done inside the *.cpp files as standard descriptions as supported by the development environment.

Example of class description:

```
//-----------------------------------------------------------
// File: CubeVolumesStorage.cpp
//
// Desc: Class that splits static meshes over predefined
//       cubic areas.
//       Contains methods for rendering only parts of the
//       full scene and for limiting the amount of faces
//       needed for collisiontesting by returning only the
//       faces close to an object.
//-----------------------------------------------------------
```

Example of method description:

```
//-----------------------------------------------------------
// Name: CCubeVolumesStorage::ConvertPointToCubeSpace()
// Desc: Converts a point in worldspace to x, y and z
//       coordinates in the CubeVolumeArray.
// Note: -
//-----------------------------------------------------------
```

These are recognized by the development environment and pop up as information to the programmer during code-completion.

6.2 Tools

Visual C++ has been used for documentation, and function description. Backup has been done by batch scripting.

6.3 Backup

The most recent versions were backed up once per day to the team-members' machines to ensure that nothing was lost in case a hard-drive chose to go "belly up". We also had copies of important older versions containing possibly useful code for later use. Once a week, the most recent version was also backed up to an online location. This last precaution was taken in case of hazardous events such as fire.

6.4 Quality insurance of this product and testing

Here we explain how we did test our product under the development process, seen from a programmer's view-point.

6.4.1 Testing from the programmers point of view

Testing was continuous throughout the whole project. Sometimes, a run through the debugger was sufficient to ensure correct input/output of a method or module, but in cases like the collision detection, graphics rendering and the 3d-sound module, tests were written along with most new methods to make sure the functionality was correct. These are critical modules that were made separately and were tested before they were implemented in the finished product. For example, while coding the collision detection method "CheckCollision()", which handles collision between a unit sphere and a triangle, a test using one invisible triangle and a unit sphere centered at the camera origin was written. A simple response of changing the background color upon detecting a collision was used to validate the functionality of the method. The camera was controlled by primitive input functions.

Error Reporting:
Routines using DirectX and Windows functions were written using the macros provided by Microsoft to evaluate the results of functions returning a result handle (type : HRESULT). This is a standard return value for most DirectX and Windows functions. Upon an error, we would get at best a logical error message followed by the name of the routine that failed and from where it was called.

6.4.2 User-testing

When the initial test product was ready, we brought in a small band of beta-testers to join in on testing the multiplayer functionality over a limited connection. This proved to work out fine even with a high number of players. We also tested different configurations of graphics cards, sound cards, processors and operating systems. The product worked nicely as long as the graphics card had an on-chip transformation and lightning engine.
Notable graphics cards tested:

- Nvidia TNT2 Ultra
- Nvidia Geforce 1
- Nvidia Geforce 2
- Nvidia Geforce 3

Operating systems tested:

- Windows 98 Second edition
- Windows ME
- Windows 2000
- Windows XP

Processor speeds tested has ranged from 450 MHz up to 1.8 GHz. No configurations caused any critical bugs.

We were also open to suggestions on how to improve the game-play. Fancy explosions, a larger level with multiple rooms and tunnels were implemented to please the eager gamers. A much requested feature that will be implemented after we have handed in the project is the possibility to pick up multiple weapons with different abilities and firepower. This will extend the game's strategic potential greatly and then the game will be at a level where it can actually compete with many commercial products.

Chapter 7

7 Discussions of the results

In this chapter we discuss the results versus the initial plans.

7.1 Comparing; End-product vs theoretical results from chap.2

The finished product fulfils the demands set at the start of the project and adds to the configuration with a lot of extra features. In fact, the extra features are now the largest part of the application which can now be considered a fully fledged 3d-game.

The initial requirements were:

- Simple 3d world
- Texture mapping
- Camera controlled by input

The final results area complete game-engine with the following features:

- Loading system for multiple 3d levels.
- Level builder application
- Game-server application
- Front-end for configuration
- Multiplayer system with player prediction
- A range of different collision detection tests
- Explosion generation (Particle system)
- Billboards
- Cockpit where pilot is affected by g-forces
- Graphics and collision delimiting
- Simple physics simulation for the craft movement.
- Mp3 replay routines
- Stereo sound
- 3D sound
- Audio related special effects simulating the real world.
- Two original soundtracks made for the game.
- A fully featured level containing a wide array of different geometry, textures and sound-effects. Tailored for up to 8 players playing simultaneously.

The flexibility, functionality, looks and speed of the game far exceeds what we had hoped for upon project initiation. For example, we can rebuild this game into a space-racing game without spending more than a couple of weeks on it because of the object-oriented design.

There are still some hard to track bugs that occur at random intervals on some configurations, but none of them are of a nature which harm the functionality of the finished game. Most failures occur while starting up, and are solved by just restarting the application again if it doesn't continue. Another bug is the "missing-sounds" bug where one or two sounds disappear. This seems to be due to soundcard limitations. This is completely harmless. Still we will continue to track bugs until we find them, hopefully in time for the presentation.

7.1.1 Deviation from the specifications of demands

The AirFear-engine has its discrepancies. No discrepancy though, has been in the favor of the functionality of the engine. Due to the teams increasing interest in this kind of development, along with the large amount hours spent developing, studying, researching and programming, the final product has been not less than 10 times more advanced than expected at the beginning of this project. No functionality has been left out. A vast range of functionalities has been added.

7.1.2 What can be improved in the game-engine?

As time has been a large issue. We decided to keep the advanced features at high priority. Things like multiple space-crafts, multiple weapons, cool status display, taunts etc are things which takes a lot of the programmers' time-schedule. Again, to accomplish this, is not an advanced task, rather time consuming. These features will be implemented before departure to Assembly 2002 (game-developers compo), but at the present time, these are features lays in the category of "things that could be improved in the game-engine".

Some algorithms could be more optimized. To make the game run faster, there is a performance penalty at the collision-detection and response module. This are always the bottleneck of today's game-engines, therefore, some portions of the code could be further optimized by implementing parts of the "innerloops" in assembly.

The interpolation routine in the Scene-Builder could be further optimized by using larger amounts of delimiting techniques. Another method for interpolation through triangles that may work better (yet unknown), is to test for intersect-triangle in each bounding-cube.

7.2 Propositions regarding changes in which way to work

We can't say that we have encountered any problems in terms of system-design issues. Our development model has proved its worth during the development process. If the members had more experience in the game programming field prior to startup we would have known what were realistic goals. Still the motivation of the team-members has been an important prime-mover making up for lack of experience.

Chapter 8

8 Conclusions

This chapter deals with conclusion and what the group have learned during this short period.

8.1 Main conclusion

From the beginning this project has been rather plagued by uncertainty as far as complexity is concerned. How much would we be able to do?

The learning curve has been steep, and as we suspected, the most complex features were the least visible. We soon discovered that digging into record-breaking material isn't possible without years of experience in the field. Learning to walk before running is the way to go.

This game sports the fundamentals of 3d-game programming which in some cases improve on the methods written in articles by some game authors. The examples and tutorials given in our literature must be considered out of date and incomplete in places according to our opinion. We found the literature to be riddled with easy short term solutions in places. For example, the book :"Multiplayer programming" which we used as a guide(not the best guide) for creating the multiplayer module and server doesn't feature movement prediction at all and has awkward workarounds to cover up for it instead.

We have found some great articles on Gamasutra which explain in mathematical terms some of the principals we have used in this project.

We reckon that the degree of difficulty has been very high. This project has been extremely challenging for us because of the mathematics and amount of code required. This project has about 27000 lines of code including all applications and the common file framework. The 3d-graphics modelling and design has also been difficult since we were not that experienced in 3d studio and Photoshop.

After much hard work, we are very happy to conclude that this has been a very successful project from our point of view judging from the amount of information we have learned, and the development experience we have gained.

8.2 What has the group learned?

To mention all the things we have learned during this project are when it comes to this document; out of scope. We think it’s more appropriate to mention some of the entries in where we have improved our skills:

- 3d-graphics programming.
- Real-time programming
- Creating simple 3D-Studio Max Scripts
- Multithread programming.
- High performance network programming.
- Local and distributed synchronization of critical sections.
- C++ callback functions and windows message-pump handling.
- Sound programming.
- User-input programming.
- Hooking up COM interfaces to C/C++.
- Advanced 2d/3d/4d vector Mathematics.
- Interpolation techniques.
- Regulator techniques.
- High/Low polygon modeling.
- Advanced texture mapping techniques.
- Texture painting techniques.
- A vast range of 3d terms.
- Flash, web design.
- The DirectX API
- Delphi programming

8.3 Further work on this project

The work on this project continues during the summer. The goal is to come to a complete game-engine, capable of handling all the points mentioned under 7.1.2. More levels will be designed, and the ability to make the server run circular through a configured list of arenas will be implemented. Some optimization will be done in the most critical modules, and the workflow will be better structured for faster synchronization of critical sections.

Chapter 9

9 Litterature list

9.1 Books and manuals

1. R. Stuart Ferguson, Practical Algorithms for 3D Computer Graphics, A.K Peters, Ltd. 2001

2. Beck Zaratian, Microsoft Visual C++ 6.0 Programmers' guide, Microsoft Press, 1998

3. Wolfgang F. Engel & Amir Geva, Beginning Direct3d Game-programming, Prima Publishing, 2001

4. Peter Walsh, Advanced 3d Game Programming using DirectX 8.0, WordWare Publishing, 2002

5. Todd Barron, Multiplayer Game Programming, Prima Publishing, 2001

6. Microsoft, DirectX Documentation, NA, 2002

9.2 Internet resources

www.gamasutra.com. This is probably the best game-programming web-page in the world. We found many articles on this page giving us perspective to things we never had though about concerning game-programming.

www.flipcode.com. This site contains information for the advanced texture mapping techniques used in the project. As the name says, it's also a lot of information on how to program games.