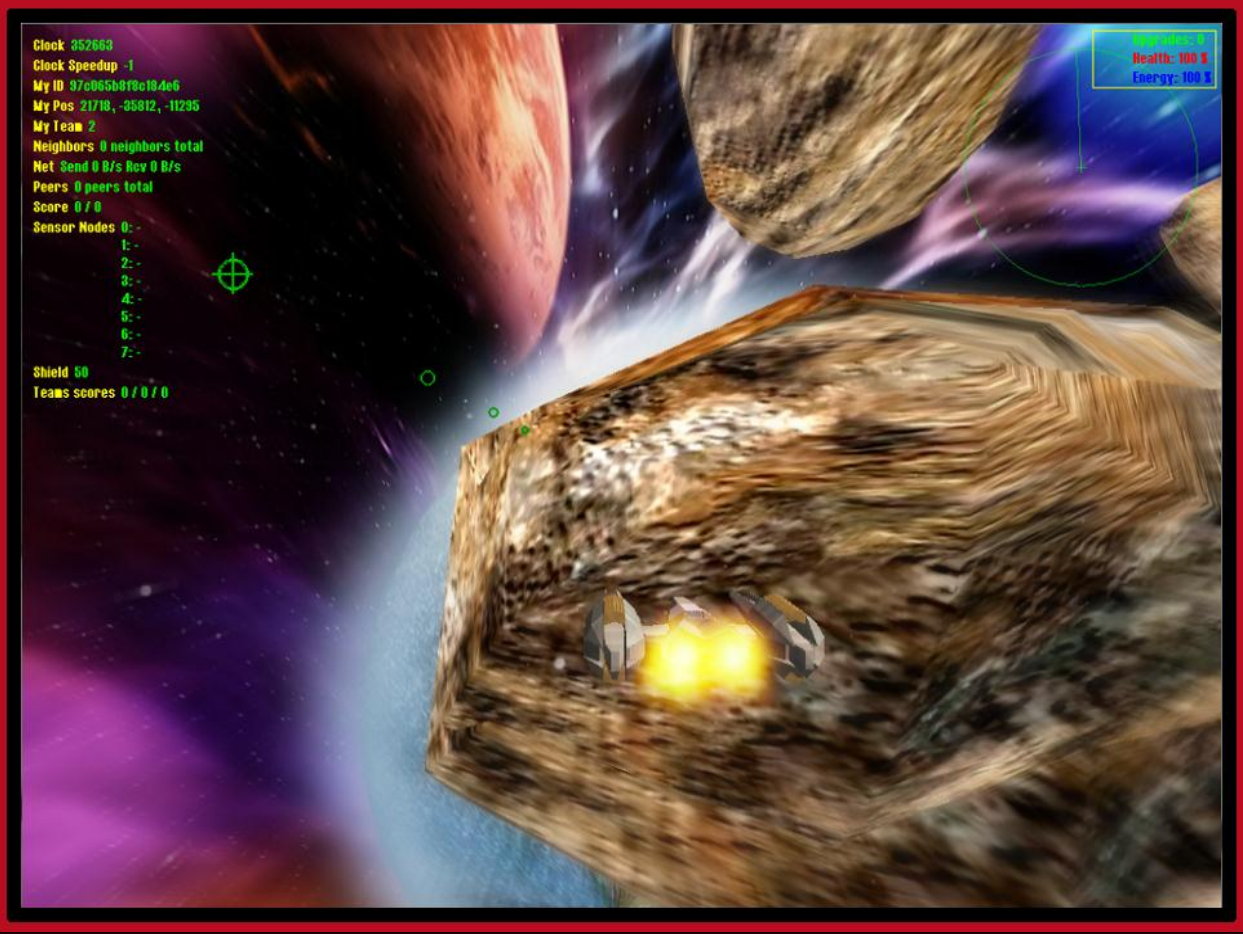
Gameplay Design and Implementation for a Massively Multiplayer Online Game

Gameplay-Entwurf und -Implementierung für ein Massively Multiplayer Online Game

Bachelor-Thesis von Denis Lapiner

März 2011



Gameplay Design and Implementation for a Massively Multiplayer Online Game
Gameplay-Entwurf und -Implementierung für ein Massively Multiplayer Online Game

Vorgelegte Bachelor-Thesis von Denis Lapiner

1. Gutachten: Prof. Alejandro Buchmann
2. Gutachten: Max Lehn

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30th March 2011

(Denis Lapiner)

Abstract

Massively multiplayer online games (MMOGs) are very popular nowadays, but mostly still are implemented in a client-server based approach. This thesis is part of a study on the question: Are peer-to-peer networks suitable for real time applications like games?. This work discusses Planet π4, a prototype for a 3D space shooter game, designed for benchmarking with exchangeable underlying network systems. In this thesis the original game was improved, especially the gameplay and the produced network load, but also graphics were reworked. The gameplay was enhanced with asteroids, upgrade points and shield regenerators, which make it more complex and allow tactical playing and team work. The keybord controls were enriched with the mouse control feature, which in combination with various motion hints and a better game balance have made the game to a dynamic and challenging 3D space shooter. The game became more intuitive and easy to play through the reworked HUD and the aim assistance. Besides, the dynamically generated world made the game's playground scalable and accessible for hundreds of players at once. The more challenging and realistic gameplay implies complex player behaviour and produces complex network load. This network load is much more representative for MMOGs and the significance of a study based on this load is higher. Besides, this works makes some suggestions on the further development of Planet π4.

Kurzfassung

Massively Multiplayer Online Games (MMOGs) sind heutzutage sehr beliebt, aber meist noch mit einem Client-Server-Ansatz implementiert. Diese Arbeit ist Teil einer Studie über die Frage: Sind Peer-to-Peer-Netzwerke geeignet für Echtzeit-Anwendungen wie Spiele?. Diese Arbeit handelt über Planet π4, einen Prototypen für ein 3D-Weltraum-Shooter-Spiel, konzipiert fürs Benchmarking mit austauschbaren Netzwerkimplementierungen. In dieser Arbeit wurde das ursprüngliche Spiel verbessert, vor allem das Gameplay und die erzeugte Netzwerklast. Auch die Grafik wurde überarbeitet. Das Gameplay wurde mit Asteroiden, Upgrade Points und Schild Regeneratoren erweitert, diese machen das Gameplay komplexer und erlauben taktisches spielen und Teamarbeit. Die Tastatursteuerung wurde mit der Maussteuerung erweitert, das in Kombination mit verschiedenen Bewegungsverdeutlichungen und einer besseren Balance des Spieles, hat das Spiel zu einem dynamischen und anspruchsvollen 3D-Weltraum-Shooter gemacht. Das Spiel ist intuitiver und einfacher geworden durch das überarbeitete HUD und die Zielunterstützung. Außerdem hat die dynamisch generierte Welt den Spielplatz skalierbar und zugänglich für hunderte von Spielern gleichzeitig gemacht. Das herausfordernde und realistische Gameplay sorgt für komplexes Verhalten von Spielern und produziert komplexe Netzwerklast. Diese Netzwerklast ist wesentlich repräsentativer für MMOGs und die Aussagekraft einer Studie basierend auf dieser Netzwerklast ist höher. Außerdem macht diese Arbeit einige Vorschläge über die weitere Entwicklung von Planet π4.

Contents

1 Introduction **5**
- 1.1 Glossary . . . 5
- 1.2 Background . . . 6

2 State-Of-The-Art **8**
- 2.1 Peer-to-Peer Gaming . . . 8
- 2.2 Planet π4 Before . . . 9
  - 2.2.1 Game Design . . . 9
  - 2.2.2 Player Behaviour and Network Load . . . 9
- 2.3 Irrlicht . . . 9
- 2.4 Examples of Peer-to-Peer Games . . . 10
  - 2.4.1 Donnybrook . . . 10
  - 2.4.2 SimMud . . . 11

3 Design **12**
- 3.1 Requirements . . . 12
- 3.2 Planet π4 After . . . 12
  - 3.2.1 Game Design . . . 12
  - 3.2.2 Player Behaviour and Network Load . . . 13
  - 3.2.3 Inspiration . . . 14
- 3.3 Feature List . . . 14

4 Implementation **18**
- 4.1 Game Architecture . . . 18
- 4.2 Network Interfaces . . . 19
  - 4.2.1 Implementation . . . 19
- 4.3 Dynamic World Generation . . . 23
  - 4.3.1 Implementation . . . 23
- 4.4 Aim Assistance . . . 24
  - 4.4.1 Implementation . . . 26

5 Evaluation **28**
- 5.1 Graphics . . . 28
  - 5.1.1 Motion Hints . . . 28
  - 5.1.2 Explosions . . . 29
  - 5.1.3 HUD . . . 30
- 5.2 Gameplay . . . 30
  - 5.2.1 Upgrade Points . . . 30
  - 5.2.2 Bullets . . . 32
  - 5.2.3 Static World . . . 33
  - 5.2.4 Respawn . . . 34

6 Conclusion and Future Work **35**
- 6.1 Conclusion . . . 35
- 6.2 Future Work Gameplay . . . 35
  - 6.2.1 Respawn Control . . . 35
  - 6.2.2 Upgrade Control . . . 36
  - 6.2.3 Messaging System . . . 36

6.3 Future Work First Impression 37
6.4 Future Work Alternative Approach 37

1 Introduction

Massively multiplayer online games (MMOGs) are very popular nowadays, the market value of computer games is twice as high as that of movies[5]. Also the number of players in the world is very high. However commercial games are still server based, which makes operating expenses of a game incredibly hight. In addition the server based approach creates a bottleneck and often leads to poor user experience. The bandwidth limits of the servers cut the maximum number of clients to low values like 16 or 32 players. Furthermore higher charges for game companies lead to higher prizes for users. A peer-to-peer approach would sink the costs drastically, it neither has bottlenecks nor costs for the companies. In addition the player limit could be removed or at least raised to a much higher value, which would enable epic-scale battles with lots of players. However the suitability of peer-to-peer networks for realtime applications like games is not proved yet. This thesis plays an important role in the study of peer-to-peer networks for realtime applications.

This work covers the further development of a game called Planet π4[14], which was originally created in collaboration of two Universities (Darmstadt and Mannheim). Planet π4 is a 3D space shooter game. The background of the project is explained in detail later in this chapter. Latest changes to Planet π4 were done by Max Lehn and are described in his work „Implementation of a Peer-to-Peer Multiplayer Game with Realtime Requirements“[10]. However the team has worked mostly on the underlying network features. Therefore as Max Lehn describes in his thesis, the game lacks of „interesting gameplay“ and it is to simple to produce representative network load[10]. The gameplay's complexity must be representative for other MMOGs. Only with a more realistic gameplay the produced network load can be representative enough to make a benchmark based on this network load significant. Besides, the graphics of Planet π4 were improved to create a better first impression of the game in presentations and conferences.

This chapter will first introduce the glossary with words used in this work. Afterwards Planet π4's background will be explained.

1.1 Glossary

This section explains uncommon terms used in this thesis.

Gameplay describes all possible interactions between user and game software. Gameplay does not consider graphics and sound.

AOI area of interest. In the context of this work AOI is always the area near some subject, for example a ship's AOI is the area in which other ships are visible.

Auto aim automatic aim. The player's weapons are directed at his enemy automatically if the player has aimed his enemy accurate enough.

Bot AI which plays a game and tries to simulate human behaviour.

DHT distributed hash map, a method of data storage in distributed peer-to-peer networks.

HUD head-up-display, shows the most important information to the player.

Instance an instance of a program is its representation in the computer's memory.

Latency the time between the user input and its effect in a network game.

Mesh the shape defining component of a 3D models.

MMOG massively multiplayer online game.

P2P abbreviation for peer-to-peer.

POI point of interest. It can be an upgrade point or a shield regenerator.

PR public relationship.

Respawn to respawn means to enter the battlefield again, shortly after your ship was destroyed.

Skybox this term is used in computer graphics to describe the way of clearing the screen before each frame is drawn. It does not mean that a sky picture has always to be drawn first, in Planet π4 the skybox is an outer space scenery with some planets.

Space shooter a computer game which is played in the outer space, the players have usually to fight each other with their spaceships.

1.2 Background

The first version of Planet π4 was created at the University of Mannheim for a workshop of the research group called „Praktische Informatik 4“[14]. Planet π4 was called after the group's name. The game was developed to evaluate and compare scalable peer-to-peer-based MMOG systems. This first version came with different network implementations (a selfmade IP-Broadcast, Skype[15], SpoVNet[8], Peers@Play[7]). **Figure 1** below shows the first version of the game. However further in the text all references to the original Planet π4 mean the version of the game before this work has started.

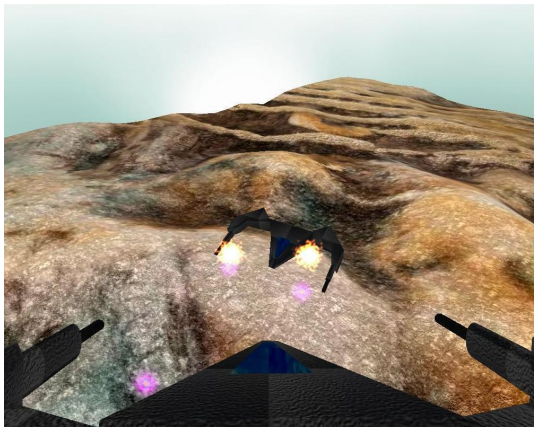


Figure 1: A screenshot from the oldest version of Planet π4[14]

Further development of Planet π4 was made in a collaboration of the „Databases and Distributed Systems“ group at the University of Darmstadt and the „Praktische Informatik IV“ group at the University of Mannheim. The central question of their work is: are peer-to-peer networks suitable for massively multiplayer online games? To answer this question benchmarks of peer-to-peer networks have to be done. In order to do benchmarks the team has started to improve Planet π4. The game implements some interfaces, which allow to exchange the underlying network layer (even from P2P to server-client based). Furthermore Planet π4 is now programmed in a way, that allows to simulate the game with a

large number of players without real time requirements. The ideas on benchmarking and requirements to Planet π4 are described in the paper „Benchmarking P2P Gaming Overlays“[11].

2 State-Of-The-Art

This chapter will first discuss what P2P games are and then present Planet π4's old version. Besides Irrlicht the graphic library used in Planet π4 will be mentioned. Finally other P2P games will be discussed.

2.1 Peer-to-Peer Gaming

What does peer-to-peer gaming mean? Are not all network games peer-to-peer games, because there are always peers connected with each other?

Usually one peer hosts a game and other peers join to it. The host controls and computes the whole game. He sends game state information to the other peers, which just visualize this information. The joined peers send informations like user input to the hosting peer. The host again takes the new user input of the other players into account to compute the next game state. The hosting peer is called server and this approach is widely known as a server-client based approach. In the context of this work peer-to-peer game means that all peers are equal and there is no server. This implies that all peers have to manage their own game instance and share their user input information to all the other peers. This kind of a peer-to-peer network cannot crash because the hosting peer leaves the game and therefore it is more reliable. Furthermore there is no bottleneck on one peer (the host in the server-client approach). Peer-to-peer networks are more scalable then server-client based networks, because every peer can compute the for him important part of the game world and only needs a connection to the nodes he is directly interested in. A server-client system needs every client to connect to the server, which leads to high computational effort on the server. In addition the bandwidth of the server has to be very high to handle all the numerous connections. Furthermore a server-client connection has a higher latency than a peer-to-peer connection. The user input has first to be send to the server, the server has to compute its effect and send information back. In contrary to that, in a peer-to-peer game user input is directly send to every player and those can calculate the effect immediately without replying. Therefore the P2P connection is approximately twice as fast. Besides, a server-client approach implies that the game company has to offer servers to the game community. Servers cost money, especially those high performance servers, that allow a high number of players. In contrary to the server-client approach a peer-to-peer network consist only of nodes brought in by players. There is no need of a high performance central instance that has to be provided by the game company. The company will probably need to host some bootstrap nodes which always stay online to allow players join the huge network. But this bootstraps do not need to be high performance and expensive machines.

However peer-to-peer based concepts are barely used for commercial MMOGs nowadays. One of the reasons for this development is that peer-to-peer network concepts are not studied enough in use for MMOGs yet. In addition the peer-to-peer gaming approach leads to problems like the synchronisation of the game state on all peers. Furthermore this approach leads to security problems. It is easier to cheat for a node, because it calculates the game state and can change anything it does not like. Furthermore every node can disconnect at random time, the nodes are not reliable, but the whole network has to be. In addition nodes are not equal some have more computational performance some less, some have a good internet connection some not. The workload of the network has to be fairly distributed in a heterogeneous network. However there are many papers written, that propose solutions to these problems. See **2.4 Examples of Peer-to-Peer Games** section for more information on already existent peer-to-peer games.

In conclusion peer-to-peer games have much potential for the industry. They could save much money to the game companies and therefore lower the costs for the players. But they are not studied enough yet. They need to prove themselves capable of being a good solution for commercial MMOGs. This work is a one more step in the study of peer-to-peer networks for realtime applications like games.

2.2 Planet π4 Before

In this section the original Planet π4 game will be introduced and the need of further development will be justified. To achieve this, first the old game design will be explained and its drawbacks will be named. Finally the generated network load will be reviewed and it will be explained why this load is not sufficient for a good benchmarking.

2.2.1 Game Design

The fights of Planet π4 initially took place directly at the planet. A little limited terrain mesh was under the fight area. However there was no collision between terrain and aircraft. The scenery was surrounded by an earth-like blue skybox. In addition a simple HUD was already included in the original version of the game. It showed a radar indicating friendly and hostile targets in player's AOI. Furthermore it showed many lines of debug output, including scores, player's team and players health points. Also there were no motion hints.

The original gameplay of Planet π4 allowed only simple dogfighting. The aircraft was directed by the arrow keys, moved with the W-A-S-D keys and was able to shoot by pressing space. However since only arrow keys were used to direct the aircraft the gameplay seemed a little tardy. In addition there was only the possibility to shoot in the direction of the ship. This all made it quite impossible to kill a bot in a fair fight. However the implementation of the bots was very simple, which allowed to outsmart the bots by strafing left or right. Moreover, the motion and steer velocity of the aircraft was quite low, the fire rate was hight and the velocity of the bullets was low, through this the impression of bullets being everywhere and that there is no way to avoid these was created.

Consequently the first impression of the game was not that good and considering all the other games of this genre the game was not that much attractive to a human player, nevertheless the game was designed to be played only by bots to analyse the network load which the game generated.

2.2.2 Player Behaviour and Network Load

Since the initial gameplay included dogfighting only the generated network load was quite simple. It consisted of position updates and few events like hitting a ship or being killed. Besides the crude implementation of the bots lead to situations where two bots would fight each other without any success. They would circle around each other and move far away from the actual battlefield. In conclusion the network load was too simple to be representative for all MMOGs and therefore a study based on this gameplay would be not significant.

2.3 Irrlicht

Planet π4 is based on the Irrlicht Engine. Irrlicht is a free high performance realtime 3D engine. It is open source and written in c++. Its source is easy to expand with additional features. Besides it is cross-platform, the supported platforms are Windows, Linux, Mac OS X and Sun Solaris/SPARC. It can render via Direct3D (8.1/9.0), OpenGL (1.2-3.x) or one of two build-in software renderers. Furthermore Irrlicht supports common file formats for textures and 3D meshes, this allows to easily include additional content. Moreover the engine has a built-in collision detection functionality. In a nutshell Irrlicht is a powerful graphic engine which supports all the state-of-the-art features and makes it as easy as possible to create an appealing graphical user interface for a game. But above all Irrlicht has a huge active community. The Irrlicht developers arrange contests and support projects which use the engine. Such community makes it easier to develop software with the engine. Almost every possible problem was already discussed on the Irrlicht forum and will be solved quickly if it was not before. In addition there are many additional features and implementations, which the community shares on the internet.

2.4 Examples of Peer-to-Peer Games

This section will discuss two already existing peer-to-peer games. However both games are scientific tests. There is no popular or at least freely available peer-to-peer game on the internet. Most papers discussing peer-to-peer games are discussing their network architectures that could be used for peer-to-peer games. They use selfmade games to evaluate their systems. This games are mostly not even designed to be played by real humans, which arises doubts on the significance of the generated network load. In addition there are no real human studies, all evaluations of networks with a high number of players are simulated with work load approximated from small user tests.

2.4.1 Donnybrook

Donnybrook[5] was published by Bharambe et al. in 2008[10]. The developers of this network system want to surpass the player number limitation of usually 32 players and enable epic battles with player masses up to 900 players. The system uses two important features to achieve its aims. Firstly it reduces the needed bandwidth and secondly it overcomes resource and interest heterogeneity.

To reduce the bandwidth frequent information exchange takes place only between players interested in each other. The system decides itself who is interesting with a player based algorithm instead of a simple AOI. This overcomes the limitation of the player number in the AOI, since the number of players which can have the attention of one human is limited to a fixed number, which can be lower than the number of players near him. All the other visible players (those who are in the AOI) send their updates with a much lower frequency and their movement between the updates is approximated with so called doppelgängers. However doppelgängers used in Donnybrook can not generate the exactly correct information in all cases. For example like described in [10] doppelgängers cannot generate realistic footstap noises.

To overcome the resource heterogeneity the workload of the network is fairly distributed between high- and low-performance machines. For that purpose multicast trees are used where machines with spare capacity forward messages send by other machines. A similar system of using spare capacity of other machines is used to solve problems arising from the interest heterogeneity. In some cases a player can become interesting for a large number of other players. For example the flag carrier in a capture the flag match. In this case his capacity could be to low to inform all the other player on his updates. Again here machines with spare capacity help this player to distribute the information.

Donnybrook was successfully tested in a user study with a modified Quake III game. However Quake III is not a MMOG since the player number is limited by the gameplay design, therefore this test shows only that Donnybrook can handle games at least with the same success like the original server-based approach. Furthermore the system could handle battles with up to 900 players in a simulation. However the simulation does not use real work load which arises from hundreds of players playing a real game, because there is simply no such game. Quake III cannot be played with more than 32 players, because the capacity of its maps is limited. The work load of the simulation is approximated and created by a work load generator. In contrary Planet π4 uses work load, which is generated by many players without approximations. In contrary to Quake III it is possible to make a user study with over two hundred participants in order to confirm the results of a simulation. In addition Donnybrook generates an interest set for each player with all the other players that this player is interested in. This implies that the generator must know much information on game mechanics and therefore needs to be adjusted to every game and even every game mode. This is one reason for Planet π4 to stay with the convenient AOI approach and keep the complexity low.

2.4.2 SimMud

SimMud[9] was made and released by the Department of Computer and Information Science at University of Pennsylvania in the year 2004. Its game world is separated in different regions, where players can move and fight each other. Only players of the same region can see each other, since updates on player's position are multicast inside the region, this surely limits the number of players inside the region. In addition there are food objects spread around the world, these can be collected and eaten. The **Figure 2** shows a sketch of the game. However SimMud is a game which is not intended to be played by humans, indeed it is not designed to be fun, but to generate network traffic of a typical role play game in order to test the DHT based architecture for peer-to-peer massively multiplayer games developed at the University of Pennsylvania. In contrary to that the new Planet $\pi 4$ is designed to be fun, in order to generate more realistic traffic and allow human user studies, which can be compared to the results of simulations.

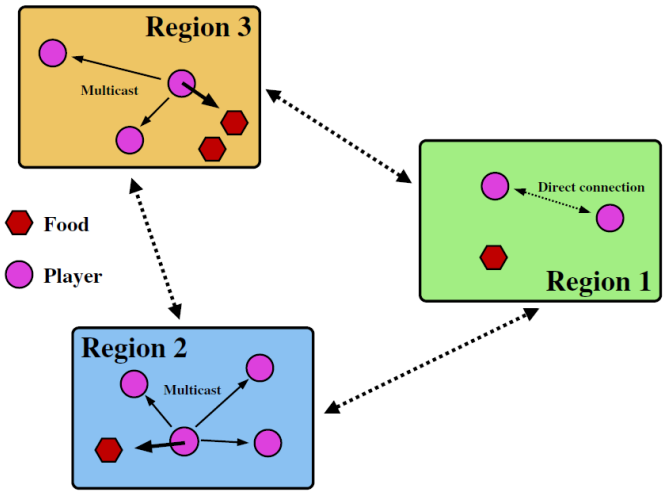


Figure 2: A sketch of SimMud's game design[9]

The paper considers issues like fault tolerance and security. Most other papers on peer-to-peer networks for games ignore this issues and identify these problems as separate issues that do not need to be considered. Security is reached through random coordinators that control the regions of the world. It is unlikely that a coordinator is interested in the random region he is assigned to. Fault tolerance is achieved through replication of the game state. In the case where a coordinator crashes or disconnects the first node with the replicated game state can take over instantly.

The network architecture of SimMud is designed for role play games, without realtime requirements. The developers use the advantage that there are many papers and effective algorithms on the DHT approach, however this approach is not optimized for games with realtime requirements and is barely usable for those. Furthermore the network system is half server based, because the regions are completely calculated by coordinators, which need to be registered at a central server in order to avoid data corruption through situation where two coordinators are assigned to one region by mistake. Furthermore the coordinators create a bottleneck just like the conventional server-based systems. Hence a region has a limited number of players, which depends on the bandwidth and computational power of the coordinator machine.

3 Design

This chapter will specify the requirements, which arise from Planet π4's background. Thereafter most important game design changes will be presented. Finally this chapter presents a feature list that has to be done during this work.

3.1 Requirements

From Planet π4's background of being a scientific prototype game used for benchmarking there a two main requirement types to the game. On the one hand there are network load requirements and on the other hand graphic requirements.

The network load generated by Planet π4 must be representative for MMOGs, otherwise the benchmarking would have no significance. Therefore it should not be simple dogfight network load, but be more related to MMOGs. Points of interest must generate areas with high and low traffic, like city and desert in an role play game, therefore shield regenerators and upgrade points are the AOI of Planet π4. Furthermore there must be more diversity in the traffic types. Firstly there is the AOI traffic, which takes place between players that see each other. Besides there is traffic which connects team members independent from their position in the world, for example the traffic which is needed to broadcast the number of captured interest points between all team members. Finally there are active objects, that are not controlled by players. Their state needs to be saved persistently and be available to every player who wants to know it. Planet π4 generates this kind of traffic for the upgrade points which need to save their owner team. In conclusion the game needs gameplay which produces diverse traffic and is fun to play at the same time.

The graphics of the game will be used rarely since the game will be mostly played by bots, which do not need any graphical user interface. Nevertheless a graphical user interface is indispensable in order to test the behaviour of the bots and verify the gameplay to be fun. In addition user studies need to be made to validate the behaviour of the bots. Moreover the game will be possibly presented on scientific conferences where the first impression of the game plays a significant role. Therefore it is obvious that despite the fact that the graphics of the game will not be seen often they need to be sufficient enough to impress players in user studies and scientists in conferences.

3.2 Planet π4 After

In the following the new concept of Planet π4 will be presented. Thereafter network load generated by the new concept will be discussed.

3.2.1 Game Design

The scenery has to change completely. The old scenery with a limited terrain is not scalable, furthermore the first impression of a terrain without a collision seems to be unprofessional. Besides, the few graphic resources, which Planet π4 has are not sufficient to create an impressive earth-like landscape. First of all the plot of the game will move from a war on the planet's surface to its ringlike asteroid belt. There will be asteroids everywhere and the scene will be surrounded by an outer space skybox. The aircraft fighters will become spaceships.

Planet π4's current debug HUD is confusing and incomplete. Furthermore its differentiation between friendly and hostile targets is inadequate. Therefore the HUD also needs a complete rework. First of all there will be a target selection feature, which highlights hostile and friendly targets. Besides, an aim assistance feature will be included. It will have the following functionality: if a target in range is selected and the mouse is near the circle, which indicates the predicted shooting direction (see **Figure 4**), the

aim assistance will take over and direct the ship's weapons at the target. Moreover, a status box will be placed in the right upper corner of the screen. It will summarize all the important information for the player. Its border will indicate the team color. This status box will show ship's health, weapon energy and team's upgrade point count. Health and energy will be shown as percent values to be more intuitive than numbers. Furthermore a flashing red coloured „low" sign will be shown when player's health or energy goes below 25%.

Beyond that, general graphic effects will be extended to improve the first impression of the game for presentations on conferences. Explosions will look much more dynamic and realistic. Besides, many motion hints like star dust particles, motion blur while boosting and jet engine particles which adapt to ship's motion and acceleration will be added. In addition the camera will not be fixed to the spaceship. This will result in the spaceship moving on the screen when it changes its direction. The camera will fall back if the ship is accelerating and come close when the ship reverts.

Also the too simple gameplay needs to be extended with various new features. In contrary to the initial game of pure dogfighting, fights will be only a tool to achieve goals. The main task in the new game will be to capture upgrade points, which will improve the spaceships of the whole team. The number of team's upgrade points will have an effect on ship's health, weapon energy and maximal boost speed. Upgrade points will be distributed randomly over Planet π4's asteroid belt. If a team will stay near the upgrade point and hold back all the other teams for a few seconds, the upgrade point will be captured by the team. If the controlling team will have no spaceships near by, other teams will be able to conquer and capture the points back. Upgraded spaceships will have more chance to win a fight. The level of the ship will be synchronised with the number of team's upgrade points every time the player is respawned. Besides, it will be possible to repair the spaceships at shield regenerator points. It also will be possible to use shield regenerator points to safely gather a group of ships before entering a fight, since it will be almost impossible to destroy a ship inside a shield regenerator. Furthermore human user interaction with the game will be more intuitive, since the user interface will be more similar to other space shooter games (inspired by Freelancer[6]). The direction of the ship and its guns will be controlled with the mouse. The left mouse key will be used to shoot. Moreover, the ship will be able to shoot at any point on the screen and not only in its direction. Also the controls will be extended by the boost key, which will drastically increase ship's acceleration, but it will stop the recharge of weapon energy as long as boost is on. In addition the game will get a more dynamic feeling through many motion hints. Moving around in the world will seem to be much faster, but also dodging away from incoming bullets will became much easier through the boost feature and the mouse control. Besides, the bots will be improved by Dimitri Wulffert in his work [16]. They not only will learn to control their ships in a much more complex way including the use of POIs, dodging from enemy's fire and avoiding asteroids, but also to interact as a team.

Finally the new Planet π4 will become a challenging game. With its new features it will be possible to play it with complex tactical team play. In addition the first impression of the game will improve a lot through the new graphic features and the more varied gameplay.

3.2.2 Player Behaviour and Network Load

The generated network load will become more representative for MMOG's. The dogfighting will be concentrated at certain areas (upgrade points) and therefore the position dependent network load at these areas will be more intensive than in areas between POIs where ships will just move around. This effect is known from other popular games. For example the player density in cities is higher than the player density between those. Furthermore there will be three types of generated network traffic. The traffic in the view range of the players, the traffic inside a team and the traffic generated by the upgrade

points. The implementation chapter contains more information on the generated network load. In summary the network load will become much more complex and therefore more realistic.

3.2.3 Inspiration

Figure 3 shows a screenshot from Microsoft's Freelancer[6] game. This game is the inspiration for Planet π4's new HUD. The screen coordinates of the spaceship are not fixed and the camera indicates changes in the motion of the spaceship (see marker 1). Marker 2 shows the shoot direction prediction cross, which works just the same way like the circle will in Planet π4. In addition Freelancer marks enemy targets with a red rectangle (see marker 3), similar to the red box planned for Planet π4. Also Planet π4's HUD box is inspired by Freelancers HUD marked with a 4. In contrary to Freelancer, Planet π4's HUD box will show percent values instead of bars, this needs to be done in order to show values which are higher than 100%. The values will be able to exceed 100% if the ship will have upgrades.

Figure 3: Screenshot from Microsoft's Freelancer[6]

Also the mouse control feature is used in Freelancer. With the mouse control it will be possible to flip around the spaceship so it will move upside down through the space. With the actual keyboard control this is not possible since the angle on ship's lateral axis is limited. To fix this problem Freelancer rotates the ship around its longitudinal axis until the ship's up vector and the world's up vector are in the same plane. This approach will also be used in Planet π4.

3.3 Feature List

This section gives a brief summary of the features that need to be done. A feature entry is structured in the following way: **feature name in bold**, feature description, *feature's purpose with cursive font.*

Graphics

HUD Cross-Hair
A cross-hair indicating the shooting direction (mouse control allows the cross-hair to be everywhere on the screen).
More intuitive for human players. Indispensable for mouse control.

HUD Box
A small box summarizing the most important information for the player (health, energy, upgrade point count and team).
More intuitive for human players.

HUD Ship Identification
Enemy ships are placed inside a red box and friendly ships inside a green box.
More intuitive for human players.

HUD Shoot Direction Prediction
Identification box of the selected enemy ship is enhanced with a red circle identifying the predicted shooting direction for auto aim.
More intuitive for human players.

Ship Up Vector Reset
The ship is slowly rotated around its longitudinal axis until the ship's up vector and the world's up vector are in the same plane.
More intuitive orientation for human players.

Motion Blur
Applies a radial blur post screen effect to the whole scene except of GUI.
Motion hint for human player.

Random Star Dust
Generates stagnant random star dust particles. If the camera is moved, these particles move on the screen and emphasise the cameras motion.
Motion hint for human player.

Jet Engine Flames
Particles coming from the ship's engine. These particles adjust to ship's acceleration and movement.
Motion hint for human player.

Following Camera
The camera is further away from the ship and is not fixed to it. It rather follows the laws of inertia.
Motion hint for human player.

Improved Explosion Effect
Dynamic explosion effect with burning ship wreck parts that move with the velocity of the exploded ship. Smoke cloud after the explosion.
Impressive effect for presentations. More rewarding user feedback for killing an enemy.

Kill Camera
A player always sees his own explosion.
Impressive effect for presentations. More penalizing user feedback for being killed.

Outer Space Skybox
The background of the scenery is an outer space picture with stars, planets and space dust.
Improves overall graphical appearance for presentations.

Gameplay

Mouse Control
Ship is controlled with the mouse. Weapon direction is not fixed to the ships direction.
Common interface used in many other games and therefore more intuitive for human players. Increases the dynamics of the game.

Auto Aim
Ship's weapons and its motion are automatically directed at the selected enemy if the cross-hair is close enough to the shoot direction prediction circle.
It is easier for a player to shoot on enemies and helps to handle the increased dynamics of the game.

Upgrade Points
Upgrade points can be captured by a team. Upgrade points improve all ships of the owning team.
POIs create more realistic network load, because the player density is more heterogeneous, since there are often more players near a POI then in between. Allows tactical gameplay.

Shield Regenerators
Shield regenerators heal nearby ship's.
POIs create more realistic network load, because the player density is more heterogeneous, since there are often more players near a POI then in between. Allows tactical gameplay.

Asteroids
A potentially endless heterogeneous asteroids field. Ships and bullets collide with asteroids. Ships loose health if they touch an asteroid.
Allows tactical gameplay. Improves overall graphical appearance for presentations.

Bullet Balance
The balancing between bullet's velocity, weapon fire rate, ship's velocity and ship's manoeuvrability decreases both the density of ships and those of bullets.
More chance to dodge from enemy bullets. Lower probability for hits by random bullets.

Spawn Protection
Shortly respawned ships are invincible for a short time.
Prevents spawn kills (players wait for other players to respawn and then kill those from behind).

Game Mechanics

Dynamic World Generation
The part of the static world, which is seen by the player is generated from random seeds. The game's world is potentially endless. It size is defined by a hard coded constant and does not affect the used memory.
Scalable world size and therefore scalable number of players.

Team Chat Functionality
A ship can send messages to its team players.
Bots use this feature to coordinate their actions. A human chat functionality can be build upon this feature.

Player Ship State Information Functionality
Player ship's health points, weapon energy and upgrade point count can be fetched by bots.
Bots use this feature to adapt their behaviour to their ship's state. For example visit a shield regenerator on low health.

4 Implementation

This chapter will briefly introduce the game architecture and then take a detailed look at the most important features, which were implemented in this work.

4.1 Game Architecture

In this thesis Planet π4's game architecture basics were not changed. Therefore they will not be explained in detail, however a more detailed view on the game architecture can be found in Max Lehn's work [10].

The main architecture guideline is exchangeability. All separated modules implement interfaces, this allows to keep them exchangeable. An example for the necessity of this design decision is that the network components need to be exchanged to create benchmarking of the same use case with different network approaches. Besides, Planet π4 is optimized for being used in simulations. It is possible to disable all graphical user interfaces to save memory and reduce computational overhead. In addition the infrastructure of the project allows one application to run multiple instances of the game. Therefore, one simulator application can run many bot-controlled game instances. However, a simulation with thousands of players will need much performance, actually it is impossible to simulate it in realtime and therefore Planet π4 does not has realtime requirements. This means that a simulation can take days, but the simulated game time can be only a few hours. Beyond that, the game can be played in fast-forward mode for testing purposes.

In the following the main components of the game are explained in more detail.

Irrlicht Device
This component handles the Irrlicht library. It initializes the graphic engine and takes care of the rendering. Furthermore the head up GUI and the debug GUI are implemented in this component. Besides, it allows access to Irrlicht's scene manager and its video device. The update callbacks of all game objects like spaceships, bullets and explosions are called from Irrlicht. In addition Irrlicht provides collision detection functionality. Therefore Planet π4's Irrlicht device component is used even when the graphical user interface is disabled.

Task Engine
This component defines the global time and makes it possible to run Planet π4 in simulated time. Components can register at the task engine for being called back after a particular time period, for example every 50ms for rendering. The task engine will callback the component when the time has passed. The callback can be called after a longer realtime period than defined, but will always be approximately right for the game time. In a simulation a frame which usually would take 50ms, can take half of a day if thousands of players need to be simulated, but the task engine's global game timer will show that only 50ms have passed in game time, despite the fact that it was a longer timespan in realtime.

Game Instance
This component manages the game logic. It keeps track of currently visible players and processes user input. In addition it handles events between players like hits, explosions and respawns. Besides, the game instance creates and keeps track of statistics.

Network Engine
The network engine distributes and synchronizes information between players. It can be implemented by a server-client based network engine or a peer-to-peer engine. This is the most important component for benchmarking. To make benchmarking of different network approaches with the same global use

case only this component needs to be replaced.

Artificial Intelligence
This component is used by the bots. There are many implementations for bots of Planet π4 the latest most powerful and realistic implementation was designed by Dimitri Wulffert [16].

4.2 Network Interfaces

Planet π4 is designed for network traffic benchmarking on different underlying network layers. To keep these replaceable the game interacts with a few interfaces, without the knowledge of their implementation.

Basically Planet π4's traffic can be differentiated in three types. First of all messages that are limited to the players view range. Secondly information which needs to be saved in relationship with world objects that are not controlled by players and finally messages between team members.

In a peer-to-peer approach, the first and last message types can be sent directly between connected peers, which implies that all peers need to establish a connection if they see each other or play in the same team. However this would limit the number of players in teams. In addition the vision range itself would be limited to keep the number of its players low. The second type of messages can be saved in a distributed database across the whole network independent of world position to assure that data inside abandoned world sectors will not be lost. However the implementation of data transfer on network layer is not part of this work and therefore will not be considered further.

4.2.1 Implementation

In order to keep Planet π4's network exchangeable all communication is done via the following three interfaces:

- **„ISpatialMulticast“:** handles area of interest communication. It is used to find nearby ships and handle position and status updates. In addition it can be used for direct communication inside the AOI.
- **„IChannelPubSub“:** deals with the team intern communication. It is used by bots to coordinate team actions. In addition it is used to publish the number of upgrade points captured by the team.
- **„IActiveObjectManagement“:** manages distributed objects, which are not controlled by players. At the current stage of development only upgrade points use this interface, but it can be used for other features in the future.

In most cases all three interfaces are wrapped with the container INetwork. The definitions of all these interfaces can be found in the folder „src/network/“, the files have the name of the interface and the extension „.h“. In the following the interfaces will be explained in detail.

ISpatialMulticast

This is the oldest interface, it was already used in the original Planet π4 version. Hence its functionality is more mature than those of the new interfaces IChannelPubSub and IActiveObjectManagement. Below is a short description for every method of the interface.

void init(ITaskEngine* taskEngine)
Initializes the spatial multicast interface. The task engine is passed as parameter, this allows the network interface to register a callback at the task engine to proceed periodical updates.

void setPlayerId(const PlayerId& id)
Assigns the player ID of the current game instance. The player ID can also be the ID of an active object, which is not controlled by the player, but handled by the same game instance.

void setTeam(int team)
Sets players team. A team ID lower than zero indicates that this instance of spatial multicast is not a player, especially not a spaceship. Upgrade points use the team ID „-1“.

void setVisionRange(float radius)
Sets the desired vision range. Other spatial multicast nodes inside this range are visible in the interest set.

void getVisionRange(float radius)
Returns the current vision range. Other spatial multicast nodes inside this range are visible in the interest set.

void updateAvatarPosition(const Position& pos)
Sets the current position of the local player's avatar. This method does not guarantee to immediately push the update to the network.

IPlayer* getPlayerById(const PlayerId& id) const
Returns the player with given ID from the interest set, if available, NULL otherwise. IPlayer is defined in the same file.

void connect(IConnectHandler* handler)
Connects to the network, returning immediately. The callback is called on success or failure. IConnectHandler is defined in the same file.

void disseminate(MsgType type, const void* data, size_t len)
Disseminates the given message to all players within the vision range.

void sendToPlayer(const PlayerId& to, MsgType type, const void* data, size_t len)
Sends the given message to a specific player.

ISpatialMulticast* newInstance()
Creates a new spatial multicast instance on the local machine. An example of usage is generating a new spatial multicast for an upgrade point handled by this game instance. The point needs its own interface since it can be on a different position than the player.

int64_t getBytesSent()
Returns send bytes count. Can be used for statistic purposes.

int64_t getBytesReceived()
Returns received bytes count. Can be used for statistic purposes.

The following six methods expect a pointer to a listener as a parameter. The definitions of the listeners are in the same file. The advantage of using listeners is that there is no need to poll data, an event is generated every time data changes or arrives.

void addMessageListener(IMessageListener* l)
Adds an incoming message listener. IMessageListener::onMessage will be called every time a new message arrives.

void removeMessageListener(IMessageListener* l)
Removes the incoming message listener.

void addInterestSetListener(IInterestSetListener* l)
Adds an interest set listener. IInterestSetListener::addNeighbor or IInterestSetListener::removeNeighbor will be called when the interest set changes.

void removeInterestSetListener(IInterestSetListener* l)
Removes the interest set listener.

void addStatusMessageListener(IStatusMessageListener* l)
Adds a status message listener. IStatusMessageListener::statusMessage will be called every time a new status message arrives.

void removeStatusMessageListener(IStatusMessageListener* l)
Removes the status message listener.

IChannelPubSub

This interface allows to subscribe to channels and publish messages on those. It is used for team communication. The methods of the interface are described in detail below.

void setPlayerId(const PlayerId& id)
Assigns the player ID of the current game instance. The player ID can also stay unset if the network interface is used by an active object, which is not controlled by the player, but handled by the same game instance.

void publish(const std::string& pChannel, ContentType pContentType, const void* pData, size_t pDataSize)
Sends the given message to every subscriber of the given channel.

SubscriptionId subscribe(const std::string& pChannel, ISubscriber* pSubscriber)
Subscribes to a given channel. ISubscriber::onPublish will be called every time a message was published on the subscribed channel. Returns a subscription ID which is needed to unsubscribe from the channel.

void unsubscribe(SubscriptionId pSubscription)
Removes the subscription with the given subscription ID.

IActiveObjectManagement

IActiveObjectManagement makes it possible to save data permanently in the network. It is used to save data of the upgrade points, especially the controlling team ID. In order to allow upgrade point captures, these need to be processed by coordinator nodes. In addition the coordinator of an upgrade point needs to inform the players of the owner team, that the point is still in team's occupancy or was captured by another team. To solve this, random nodes are picked from the network and assigned to specific upgrade points. This player nodes handle the upgrade points independent from players position and actions. Actually this handling nodes can be seen as separated nodes that only handle the upgrade points which they are assigned to. Active object management has three main interfaces: IActiveObjectManagement,

IRemoteObject and IManagedObject. These are described in detail below.

First of all the most important interface IActiveObjectManagement:

void setPlayerId(const PlayerId& id)
Assigns the player ID of the current game instance.

void registerHandler(IActiveObjectHandler* handler, ISpatialMulticast* spatialMulticast)
Registers a handler for running active objects. IActiveObjectHandler::addActiveObject will be called if an active object will need to be handled. In this callback a pointer to a IManagedObject will be passed, which allows to read the current data of the handled object and write changes to it. In addition the INetwork container will be passed with new network interfaces for the active object. Those new network interfaces are needed, since the active object can be on a different position than the player himself and the active object will not want to listen to the same channels in the publish subscribe system like the player.
IActiveObjectHandler::removeActiveObject will be called if one of the assigned active objects does not need handling any more.

void retrieveObject(ObjectId pObjectID, IRetrieveObjectCallback* pCallback)
Searches for saved data for this object, if there is no data stored yet, a default payload (null) will be generated. This method is asynchronous, i.e., the object is returned via the IRetrieveObjectCallback::onRetrieveObject callback method. Note that you'll have to drop() the returned pointer, after you don't need it any more, see IReferenceCounted::drop() for more information.

The next interface of the active object management is IRemoteObject, it is in the same file as IActiveObjectManagement. This interface allows to read data of remote objects.

const ObjectId& getID() const
Returns the object's unique ID.

const void* getData() const
Returns the data associated to this object. See also getDataSize().

size_t getDataSize() const
Returns the length of the data array.

void addChangeListener(IObjectListener* pListener)
Adds an object update listener. This listener is notified (IObjectListener::onObjectUpdate is called) whenever the data associated with the object is changed (locally or remotely).

void removeChangeListener(IObjectListener* pListener)
Removes the object update listener.

Finally the last interface of the active object management is IManagedObject, it inherits from IRemoteObject. In addition to the functionality defined in IRemoteObject this interface allows to write the data of remote objects.

void setData(void* pData, size_t pDataSize)
Changes the data of the remote object. The network layer will (more or less) immediately distribute the data and notify all listeners (including local listeners).

4.3 Dynamic World Generation

Since the world of Planet π4 is very large and has almost no limits, it is impossible for one game instance to save the whole static world information like asteroids and POIs. To solve this problem a game instance only generates the world information for the particular position of the own spaceship. For that purpose the world is represented by a regular grid. In the current version the grid is two dimensional but the implementation allows to easily extend the grid to 3D by changing the „World“ class. Every cell generates its static objects with a specific random seed as soon as the player could possibly see it. All game instances have the same random seed, since the random seed is calculated from the grid cell index and therefore the world is generated in exactly the same way on all computers. In addition cells which are not visible any more can be erased to save memory and reduce computational complexity. Furthermore this approach relaxes the problem of collision detection, since it is possible to make brute force collision and distance calculations on the relatively small number of asteroids and POIs in vision range.

However, there are different methods to save world data and allow fast collision detection. For example bounding volumes could be used. The simplest implementation of a bounding volume algorithm would be to wrap each asteroid and POI with a sphere. The nearest of these spheres will again be wrapped by a sphere. Repeating this algorithm results in the whole world bounded with a sphere tree (bounding volume tree). Unfortunately this tree must be calculated for the whole world before the game can start, this results in a much longer loading time and higher memory requirement. Besides, this approach implies an overhead of recursively searching objects in the tree. This overhead grows logarithmically with the size of the world. Therefore this approach is not scalable for an almost endless world. Despite this, the bounding volume algorithm can be applied inside one grid cell of the current implementation to accelerate collision detection. However this would result in an overhead of calculating the bounding volume tree when a new grid cell becomes visible and needs to be created. Another approach are quad trees. In this trees the world is divided in four equal parts. Every part that contains more than one object is divided again. Unfortunately this approach does not work well for a randomly generated world, since the tree would look very similar to a regular grid, like the one Planet π4 uses. However it is again possible to use this approach to speed up collision detection inside a grid cell. Finally, the current implementation of Planet π4 does not use a smart collision detection algorithm inside grid cells, because they are charged with overhead calculations like searching in trees. But the number of objects inside a grid cell is not very high and therefore the complex algorithms would not significantly improve the performance. However, if the number of object should increase in the future a collision detection algorithm can be implemented for faster detection inside grid cells. Detailed information on state-of-the-art collision detection algorithms and their implementation can be found in Ian Millington's book „Game Physics Engine Development“[12].

4.3.1 Implementation

All implementation of the dynamic world generation system is done in the „World“ class, which can be found in „src/core/World.h(.cpp)“ files.

The private array „m_activeCells“ contains pointers to instances of „GridCell“. A „GridCell“ saves its information about asteroids and points of interest. There are nine grid cells in „m_activeCells“, one in the middle and eight around it. The player's spaceship is always in the middle cell. In order to hide world generation effects like asteroids popping up from nowhere all neighbour grid cells must be generated as well. To achieve the desired effect the size of a grid cell must be at least half of the players view frustum radius.

Each grid cell has an index value, which is used to gain a specific random seed for every grid cell and make each of them unique. Furthermore the index based random seed makes sure the cell content

is generated in exactly the same way on every machine.

Finally the public methods of the „World“ class are described below:

void UpdateGrid(const core::vector3df& pCurrentPlayerPos)
Generates and removes cells from the „m_activeCells“ array and makes sure the middle cell contains the „pCurrentPlayerPos“.

vector<AsteroidData>& GetAsteroidList()
Returns all potentially visible asteroids (those from all cells of „m_activeCells“ array).

vector<POI_ShieldRegenerator*>& getShieldRegeneratorList()
Returns all potentially visible shield regenerators (those from all cells of „m_activeCells“ array).

vector<POI_UpgradePoint*>& getUpgradePointsList()
Returns all potentially visible upgrade points (those from all cells of „m_activeCells“ array).

static ObjectId GetDistributedObjectID(u32 pGridIndex, u32 pObjectIndex, DistributedObjectType pType)
Generates a unique ID from given grid index, object's index inside the grid cell and object's type. These ID can be resolved to those three components afterwards which again can be used to find the default generated values like position. This is used for default data generation when a distributed object is assigned to a peer and has no data set.

static core::vector3df GetUpgradePointPos(ObjectId objID)
Returns the position of the upgrade point with the given object ID. To do so the given ID is resolved to its three components which are used to find the default generated position.

4.4 Aim Assistance

Inspired by the aim assistance of Microsoft's Freelancer[6] game, Planet $\pi 4$ identifies the predicted shoot direction on the screen. A little circle is drawn near the selected target, see **Figure 4**. If the target does not change its direction and velocity during the bullet travel time the bullet will hit it. If the player has his mouse near enough to that circle the ship will automatically be directed to the center of this circle.

Prediction Algorithm

Planet $\pi 4$ uses a simple algorithm to predict the target's next position. It is necessary to predict the position which the player can shoot at, since otherwise it would be very difficult to hit a ship, because both, ships and bullets move quickly. In addition most space shooter games offer a shoot direction prediction assistance.

The basis of the prediction algorithm is to predict the bullet's travel time ($\mathbf{t}_{bullet}$). Afterwards the target's velocity is simply multiplied with $\mathbf{t}_{bullet}$ to get the offset from target's position. This formula assumes that the target will not change its direction and velocity over this time. Usually the time will be very small and the simplified formula will do its job, however there are some cases were the formula is not working correctly, for example if the target is relatively far away and additionally moves away from the player's ship, in this case the passed time will be high enough for the target ship to avoid the incoming bullets.

The basic approach to find $\mathbf{t}_{bullet}$ is to divide the distance $\mathbf{d}$ between target's position $\mathbf{p}_{target}$ and player's ship position $\mathbf{p}_{player}$ by bullet's velocity $\mathbf{v}_{bullet}$.

Figure 4: A screenshot from the new Planet $\pi 4$, the arrow points at the shoot direction prediction circle

$$\mathbf{d} = \left|\mathbf{p}_{\mathbf{target}} - \mathbf{p}_{\mathbf{player}}\right| \tag{1}$$

$$\mathbf{t}'_{\mathbf{bullet}} = \frac{\mathbf{d}}{\mathbf{v}_{\mathbf{bullet}}} \tag{2}$$

However this approach does not take into account that the target ship moves during $\mathbf{t}_{bullet}$. To solve this problem it is possible to apply the algorithm above multiple times. The next $\mathbf{t}_{bullet}$ can be calculated from the distance to the shortly predicted position. The algorithm can be repeated until the change in $\mathbf{t}_{bullet}$ falls below a threshold. But this approach would cost more performance, which would slow down the simulation. Planet $\pi 4$ again uses a little approximation here. If the target is far away only the component of the the targets velocity ($\mathbf{v}_{target}$) in the direction between player and target will have a considerable impact on $\mathbf{t}_{bullet}$. This can be explained with some examples. If a ship that is far away moves to the side and stays at the same distance then the time which the bullet needs to reach the target will stay exactly the same, since it has to travel the same distance. If the ship is approaching the player directly, the bullet time will be less, because the approaching ship will reduce its distance to the player over the time which the bullet travels. The following dot product calculates the amount of target ship's velocity in direction of the player's ship ($\mathbf{v}_{target\ directed}$):

$$\mathbf{v}_{\mathbf{target\ directed}} = \mathbf{v}_{\mathbf{target}} * \left(\frac{\mathbf{p}_{\mathbf{target}} - \mathbf{p}_{\mathbf{player}}}{\left|\mathbf{p}_{\mathbf{target}} - \mathbf{p}_{\mathbf{player}}\right|}\right) \tag{3}$$

Now $\mathbf{v}_{target\ directed}$ can be added to $\mathbf{v}_{bullet}$ in order to get the real approach speed of the bullet and the target ship. Inserting this in the old formula of $\mathbf{t}'_{bullet}$ leads to the final formula for $\mathbf{t}_{bullet}$:

$$\mathbf{t_{bullet}} = \frac{\mathbf{d}}{\mathbf{v_{bullet}} + \mathbf{v_{target\ directed}}} \tag{4}$$

Finally $\mathbf{t}_{bullet}$ is multiplied with the target's velocity to get the offset. This is added to the target's initial position to receive the predicted position ($\mathbf{p}_{predicted}$) and the final formula used by Planet π4's aim assistance:

$$\mathbf{p_{predicted}} = \mathbf{p_{target}} + \mathbf{v_{target}} * \mathbf{t_{bullet}} = \mathbf{p_{target}} + \mathbf{v_{target}} * \frac{\left|\mathbf{p_{target}} - \mathbf{p_{player}}\right|}{\mathbf{v_{bullet}} + \mathbf{v_{target}} * \left(\frac{\mathbf{p_{target}} - \mathbf{p_{player}}}{\left|\mathbf{p_{target}} - \mathbf{p_{player}}\right|}\right)} \tag{5}$$

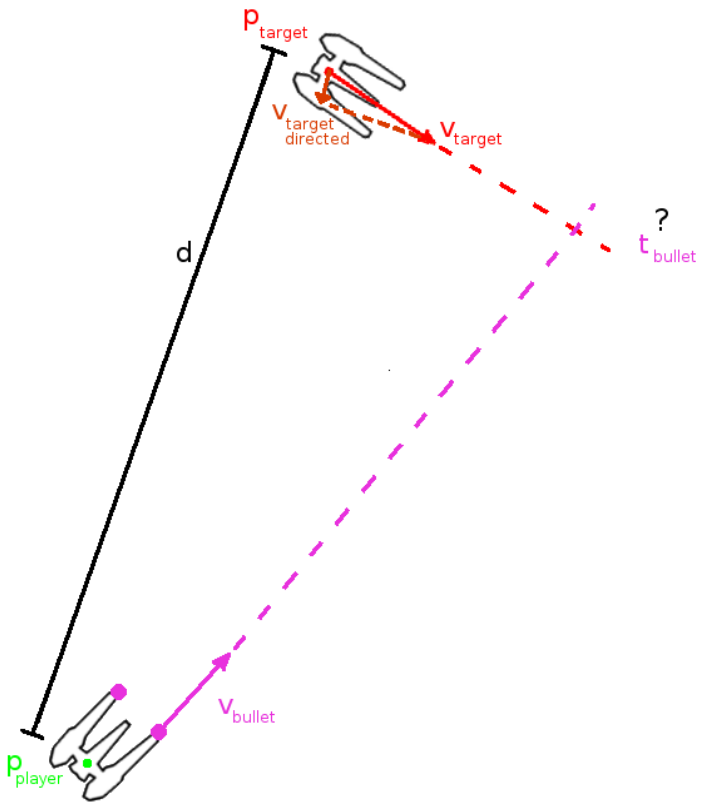


Figure 5: A sketch of Planet π4's motion prediction

4.4.1 Implementation

The GUI of the aim assistance is drawn by the IrrlichtDev (Irrlicht Device), its files can be found in „src/frontend/IrrlichtDev.h(.cpp)“. The function below handles the aim assistance GUI and locks the ship on the target.

void IrrlichtDev::GUI_HandleShipIdentification(video::IVideoDriver* pDriver)
This function iterates through the ships in the area of interest. If a ship is on the screen an outline box will be drawn around it, otherwise a filled box will be drawn at the border of the screen. The color of the box is green if the ship is friendly and red if it is hostile. Besides, the enemy ship whose screen position is nearest to the player's mouse position is saved. If this selected target is on the screen and its distance to

the player's mouse position is below a threshold the ideal shooting direction will be calculated and the shooting direction circle will be drawn. In addition if the player's mouse position is near enough to the circle, the ship will be locked on that target, see IShipControl::lockOnTarget below for more information.

The lock on target functionality of the ship is defined in the IShipControl and is implemented by the Spaceship class. Both files can be found in the „src/core"folder.

void IShipControl::lockOnTarget(IShip* pTargetShip)
Locks on the given target, therefore the ship will direct itself on the target. If target is set all the other functions to set ship's direction are disabled. If target is NULL ship is controlled in the usual way. A target can only be locked if it is in bullet range, this means the distance to the target is less or equal to the maximal travel distance of the bullets. This function is also used by the current implementation of the bots.

The ideal shooting direction calculations can be found in the ExtMath class. The functions are defined as static to allow the implementation to access it from different code parts. The GUI, the spaceship it self and the bots use this prediction. ExtMath can be found in „src/util/ExtMath.h(.cpp)".

static float GetBulletRange()
Returns the maximal distance, which the bullets can travel. The value equals the product of bullet's lifetime and velocity. This function is used to determine if the ship can be locked on a target. It can only be locked if the target's distance is equal or less than the bullet range.

static vector3df PredictShootPosition(const vector3df &pOwnPos, const vector3df &pTargetPos, const vector3df &pTargetVelocity)
Predicts the position which the player's ship should shoot at. Returns a zero vector if target is out of range.

static vector2df GetShipControlsFromDirection(IShip* pShip, vector3df pDirection)
Transforms a (not normalized) direction in the 3D world to the according values for the ship controls. If the direction can not be represented with ship control coordinates, the nearest values will be returned. Therefore if the direction shows in the opposite direction of the ship, the returned controls will first turn around the ship over time, before the direction can be faced properly. This function is used if the ship is locked on a target.

5 Evaluation

This chapter will evaluate Planet π4's progress, which was made during this thesis.

5.1 Graphics

This section will show graphic improvements between the old and the new version of Planet π4.

5.1.1 Motion Hints

In the old version of Planet π4 the ship's movement seemed to be slow but through the motion hints the game appears to be faster now. In contrary to the old version the camera's position is not fixed to the player's ship anymore. The ship moves further away from the camera if it accelerates, it also moves on the screen when it changes its direction. The image sequence in **Figure 6** shows the effect.

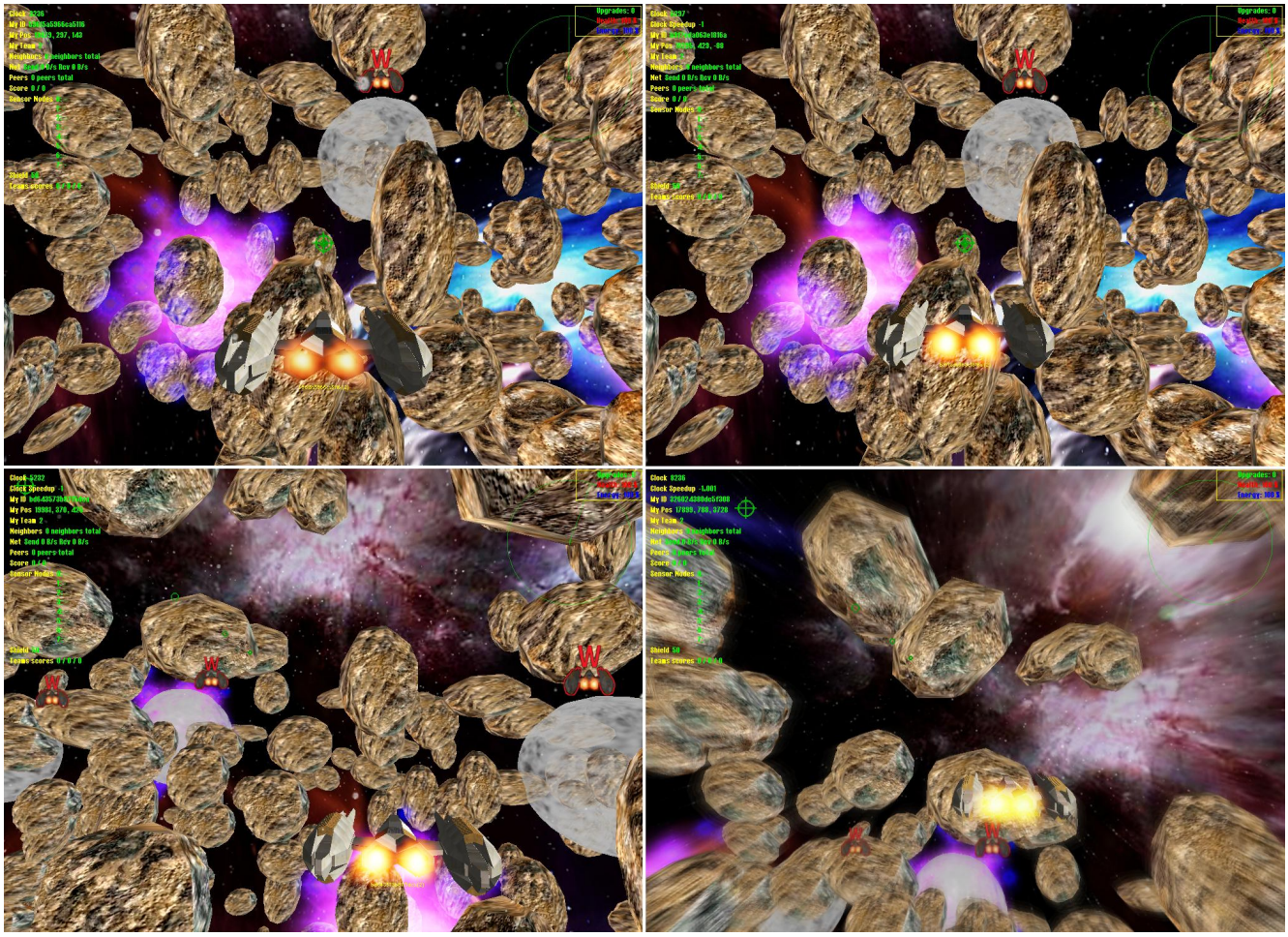


Figure 6: Top left: idle ship. Top right: ship accelerates. Bottom left: ship accelerates and makes a curve. Bottom right: ship boosts and makes a curve

Another motion hint is the motion blur, which is shown when the ship is boosting. **Figure 7** shows the new effect in comparison with the old game version.

Furthermore there are other secondary motion hint effects like the engine fire particles, which adapt to ship's acceleration or random star dust particles, which rush past the screen when the ship is in motion. However these effects are not easy to show in a screenshot.

Figure 7: Left: original version. Right: current version with motion blur (on boost)

5.1.2 Explosions

Planet π4's explosions were improved a lot during this work. The old explosion effect lacked of dynamics, it was always the same effect independent of ship's velocity. Furthermore it had no fade out. The particle system build up, stayed for a few seconds and than just disappeared. The old explosion effect can be seen in **Figure 8** below.

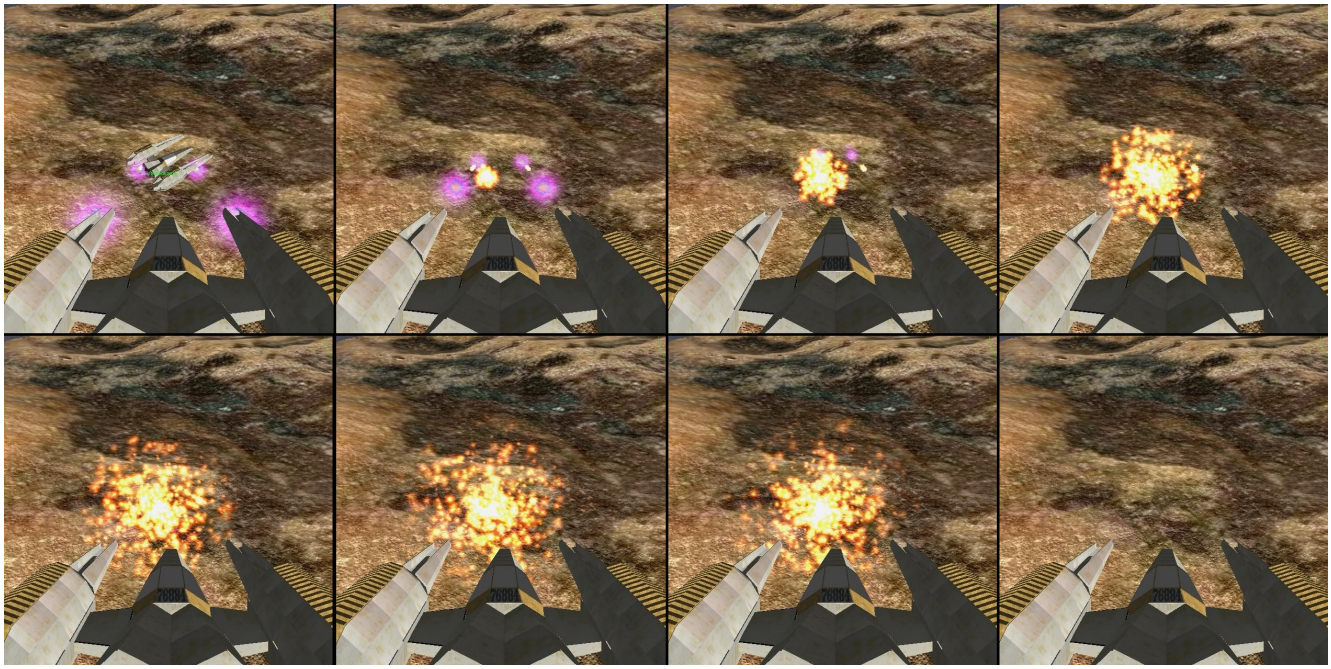


Figure 8: An image sequence of the original Planet π4 explosion

The new explosion effect (**Figure 9**) is composed of 3 different particle system types. The first one is taken from the old effect, but its emission time is reduced. It creates an explosion wave which moves away from the explosion center. This wave is still independent of spaceship's velocity. However the second particle system type simulates burning wreck parts of the ship. These particles keep the ship's velocity and move on a slightly randomized trajectory leaving a fire trail behind. There are six particle emitters that simulate the wrecked ship parts. Finally the last particle system creates a slowly dissolving smoke cloud, which indicates the position of the recent explosion. The new explosion effect increases

the dynamics of the game a lot. In addition when the player destroys an enemy ship he is rewarded in a much appealing way.

Figure 9: An image sequence of the new Planet π4 explosion

5.1.3 HUD

Original Planet π4 had no HUD, actually it only could show a debug output list, which was hard to read especially for inexperienced players. Besides, the debug output has shown only the current health point and not the weapon energy. Therefore a HUD box, see **Figure 10**, was added to the current version of the game. It shows only the important information and is more readable then the debug output list. Additionally it shows percent values which are more intuitive then numbers. Furthermore it informs the player if his health or energy falls below 25%. Moreover, the new HUD shows the current number of upgrade points and the current upgrade state of the ship can be extracted from the percent values of health and energy, since values can go over 100% percent. For example if the ship has full health and three upgrades it will have 115% health. Finally the new HUD also shows the team color in the outline of the box, which again is more intuitive than the team ID.

Besides the old HUD had not even a cross-hair, which is presumably the most important HUD element of a spaceship game. However this can be explained with the fact, that the old game was played only by bots and these do not need a cross-hair to aim.

5.2 Gameplay

In the following gameplay changes to Planet π4 during this work will be evaluated.

5.2.1 Upgrade Points

Planet π4's bots were redesigned by Dimitri Wulffert and documented in his bachelor thesis „Artificial Intelligence for a Massively Multiplayer Online Game“[16]. Besides, Dimitri has extensively simulated their behaviour to evaluate his bots. Some of this simulations reveal important information on the relation between the team's number of upgrade points, the kills and the deaths of the bots. First of all a simulation of two teams (Zero and Alpha) over 100 minutes reveals that the basic concept of upgrade

Figure 10: Left: debug output list of the original game[10]. Bottom right: new HUD box

points works. Dimitri used a metric called kills-deaths-ratio (KD-ratio), which is calculated by subtracting the deaths number from the kills number. This metric is positive if a team makes more kills than deaths. In a team versus team match this value is always positive for one team and negative for the other team if suicide deaths (for example through asteroid collision) and team kills are ignored. The **Figure 11** shows the number of upgrade points owned through the simulation and the graph in **Figure 12** shows that the kills-deaths value is negatively affected with a low number of upgrade points and slightly positively affected with a high number of those. In the first 20 minutes the two teams fight for the upgrade points, but the number of those stays equal. However the Alpha team even starts to make more kills. In the middle of the simulation the Zero team clearly gets the upper hand on the number of upgrade points. Therefore the KD-ratio of team Alpha decreases far below 0 and the KD-ratio of team Zero is clearly positive. From that point on team Zero has more kills than team Alpha and therefore wins the match. However in the last 10 minutes team Zero looses its advantage and team Alpha captures more upgrade points than the winning team. This clearly affects the KD-ratio of team Zero, which falls to negative values and team Alpha's KD-ratio starts to climb back up. Unfortunately the simulation ends at this point with the Zero team winning the match and the Alpha team having not enough time to catch up the Zero team's kill number.

Unfortunately the simulation of three teams showed that the number of upgrade points does not play an important role in winning the match by the number of kills. The upgrade point count affects the KD-ratio in the same way like described above. However if one team has upgraded ships and tries to keep their death rate low, nothing stops the other two teams from making a massacre around the spawn position. They surely will have a negative KD-ratio, because also the upgraded team will kill their ships, however they will have much more kills than the team which invests time in capturing upgrade points. Section **6.2.1 Respawn Control** offers a solution for this issue. However the fact that the team success can not be measured with the kills number any more is not a problem. The upgrade point count regulates the KD-ratio, which is the real factor that shows the success of a team. Therefore the ranking of the team needs to be based on the team's upgrade point count and not the kills. This is not a new approach in the gaming world, for example the Battlefield[2] series ignore the number of kills to determine the winner of a match. In this games there are bases, that work similar to upgrade points in Planet π4. A base provides the holding team with weapons. In these games the team that has hold the higher number of bases for the longest time wins.

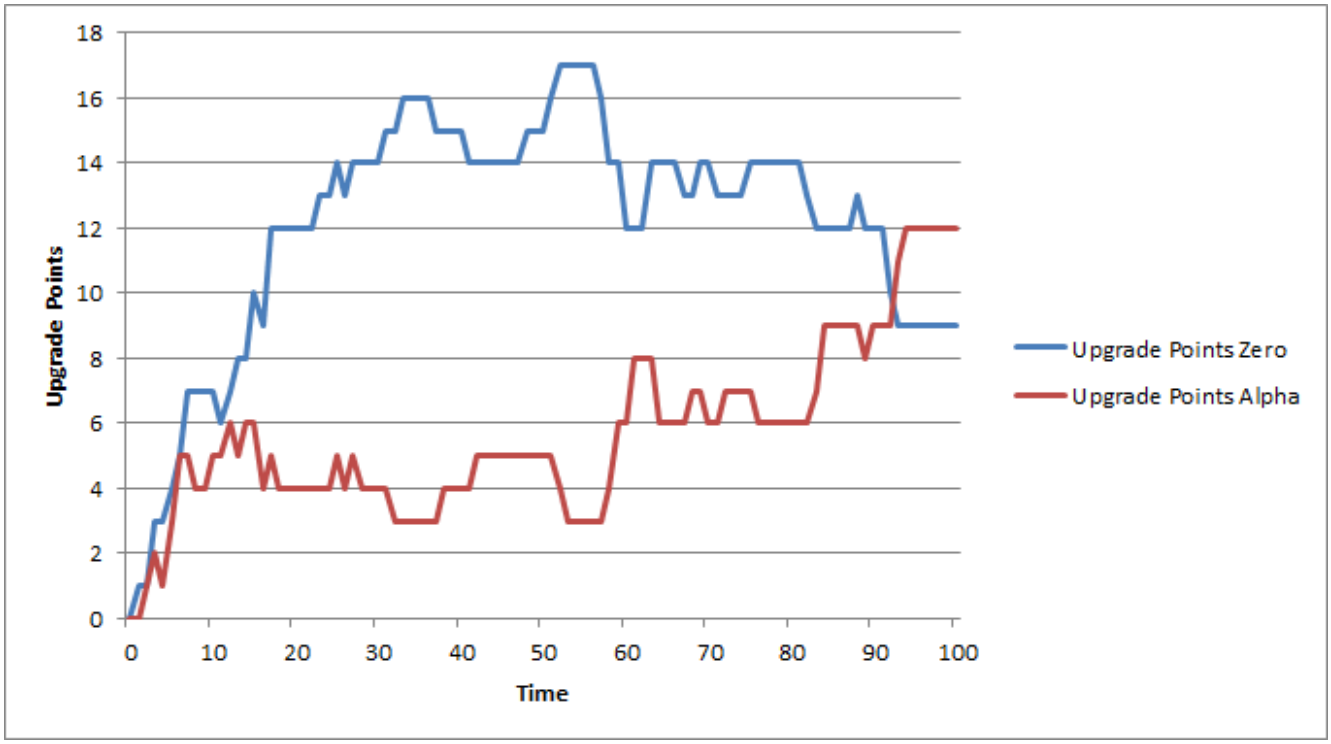


Figure 11: This graphic represents the amount of upgrade points captured over time.[16]

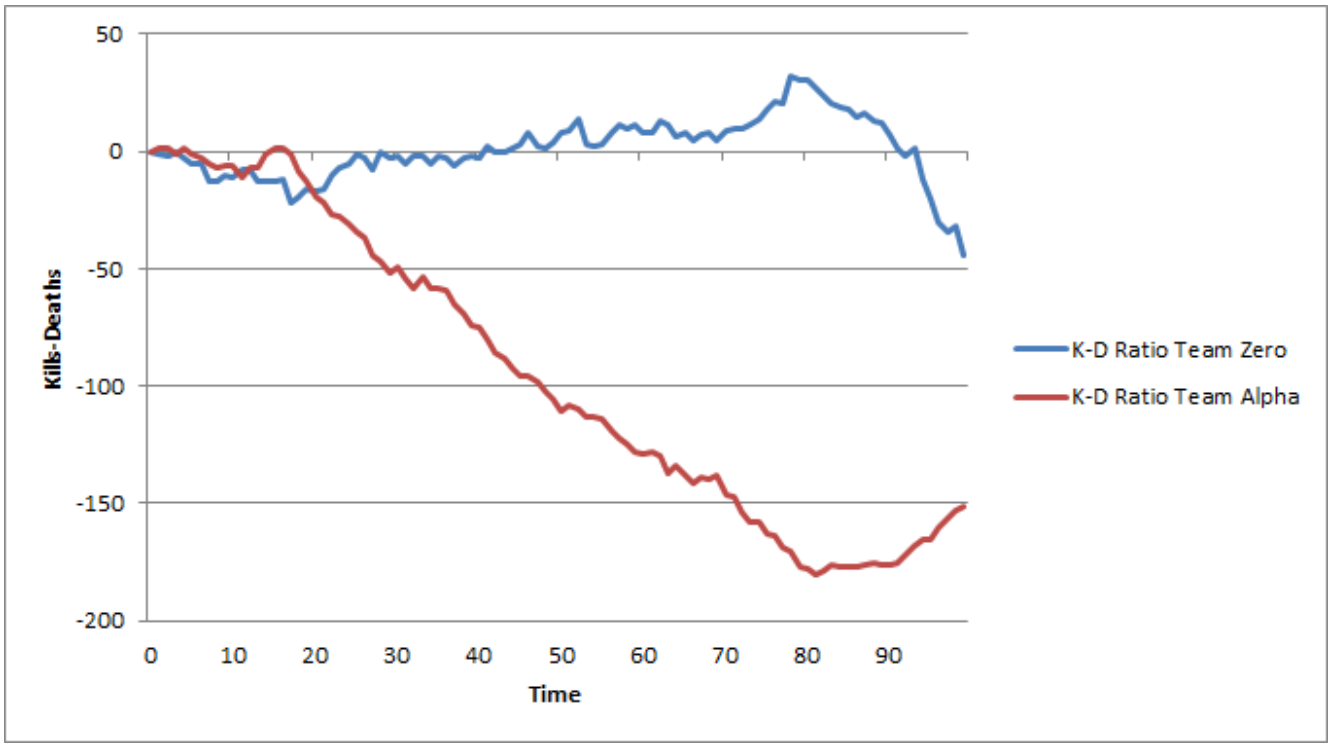


Figure 12: This graphic represents Kills - Deaths ratio. Making more kills than deaths results in a positive kill death ratio.[16]

5.2.2 Bullets

Like it was mentioned before the constellation of ship's low manoeuvrability, bullet's high fire rate and velocity, created the impression of bullets being everywhere (see **Figure 13**) in the old version of the

game. It seemed to be unavoidable to get hit by random bullets. Also the number of team kills was very high.

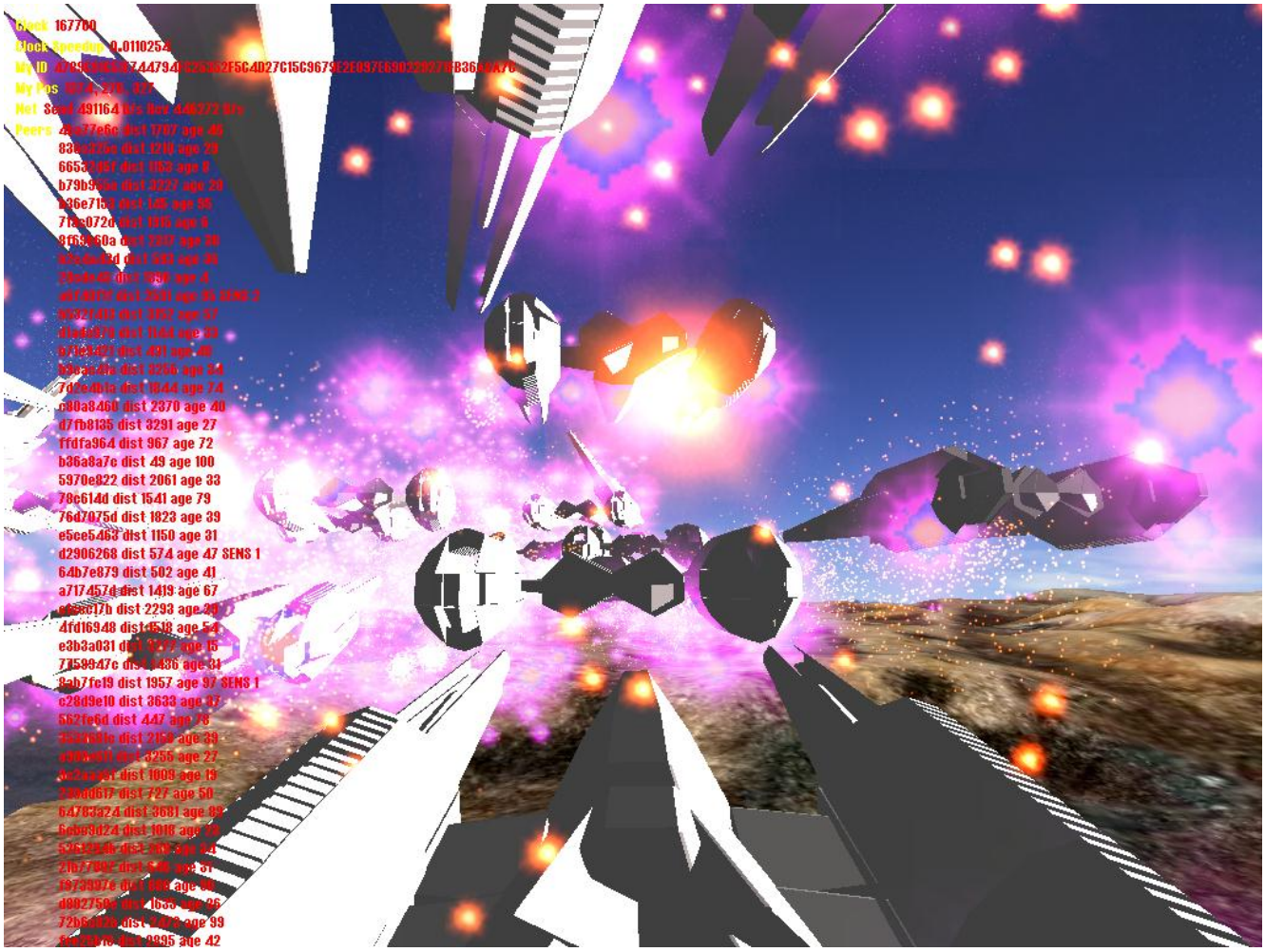


Figure 13: Screenshot from old Planet π4 with many players[10]

In contrary to that chaos, the new version makes sure to keep the ship density low. The ships are much faster and can steer quicker, therefore it is almost impossible to create a situation like in **Figure 13**. Besides the boost feature allows the player to escape such situations quickly. Moreover the bullet velocity is higher now. This consequently means that bullets have to be shot at a specific point and can not be just sprayed in the space in the hope that some other ship will collide with them later. However better balanced game properties are not the only reason for the lower ship density of the new Planet π4. Upgrade points and shield regenerators make sure to keep the players from dogfighting only. In addition this POIs separate the big fighting cloud from the old version of the game in many little areas of interest, where only a small number of players fight at once.

The **Figure 14** below shows 10 ships spread on the screen. The density of the ships in a relatively small area (like the area with the ships in **Figure 13**) is not higher than 3, while in **Figure 13** more than 10 ships are in a very small area.

5.2.3 Static World

The old version of Planet π4 had first signs of a static world. However the terrain under the fight area was not only limited, but also free of collision. The main component of Planet π4's new version's static

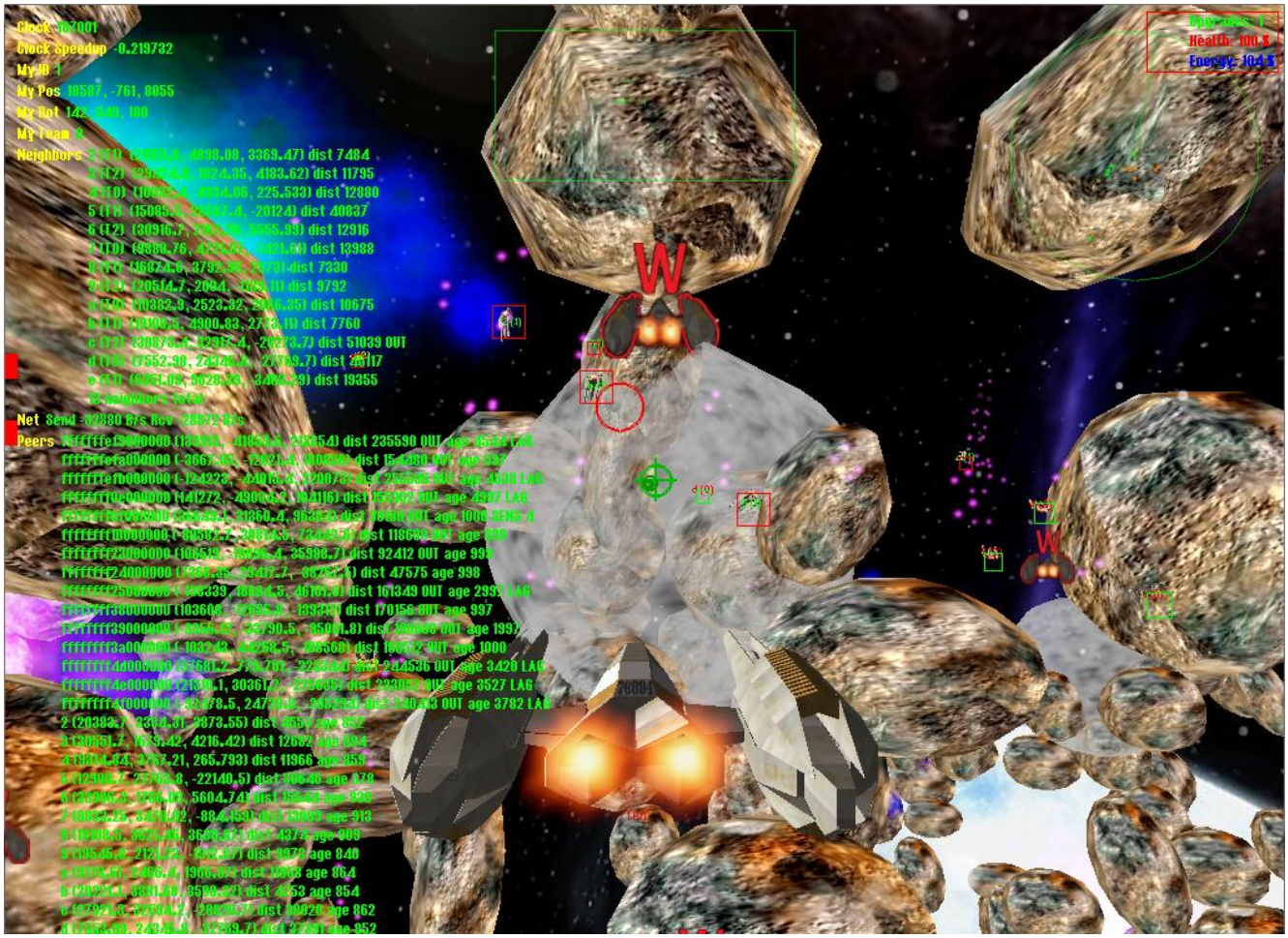


Figure 14: Screenshot from new Planet π4 with many players

world are the asteroids. They are only limited by a constant defined in the code. Potentially the world of the new version is unlimited, since the memory requirements do not raise with a bigger world. The asteroids allow a more challenging and tactical gameplay. Players can use them to hide and they have to avoid them in order not to take damage. Besides POIs make sure players have multiple aims and not only the big fighting ship cloud from the old version. Like mentioned before this game design keeps the ship density low.

5.2.4 Respawn

If a player dies in the old version of Planet π4 then he sees a red screen for a few seconds while his new ship stands unprotected at the respawn position. This behaviour of respawning the ship before the player controls it, is done to give the network some more time to handle the teleportation of the ship and give it time to inform all neighbour players of the new ship. However it can happen that an other player would hit or kill the respawning player while he just sees a red screen. Hence this, in the new version the respawning ships are invincible for a short time. In the new version the player sees his own ship explode before he can control his new ship. His newly respawned ship is invincible during this time. However a player could move to an advantageous position, for example behind the respawning ship. This is a known problem of fixed respawn positions. Players who kill repwaning players are called spawnkillers in the game world. Therefore most games have a spawn protection mechanism that gives advantage to shortly spawned players. In most cases the players are invincible for a short time, which dispossess spawnkillers of their advantageous position. Also Planet π4's new version has this spawn protection. Shortly respawned ships stay invincible after the player starts to control it for a few seconds.

6 Conclusion and Future Work

This final chapter will firstly conclude the achievements of this work and later present suggestions for the future.

6.1 Conclusion

Planet $\pi 4$ is now a better game in all aspects. Over a dozen new features were implemented. Asteroids, upgrade points and shield regenerators have made the gameplay more complex and allowed tactical playing and team work. Mouse control, various motion hints and a better game balance have made the game to a dynamic and challenging 3D space shooter. The game became more intuitive and easy to play through the reworked HUD and the aim assistance feature. Besides, the dynamically generated world made the game's playground scalable and accessible for hundreds of players at once. Also the bots are now supplied with more information on their spaceships and their environment. The game's graphics have improved a lot and therefore the first impression of the game is enhanced. The game has better chances to attract attention on scientific presentations now. Finally the more challenging and realistic gameplay implies complex player behaviour and produces complex network load. This network load is much more representative for MMOGs and the significance of a study based on this load is higher now.

However improving the game was an important, but still small step for the study on the usability of peer-to-peer networks for realtime applications like games. The underlying network implementations have to be finished. Benchmarks of these different network systems have to be done. Besides, user studies must be made in order to validate the bot behaviour and the results of simulations.

6.2 Future Work Gameplay

The gameplay of Planet $\pi 4$ has improved, however there is still much room for further development. This section will show some of game's aspects that can be reworked or improved.

6.2.1 Respawn Control

The players are spawned at a fixed position. Each team has its own position. However this repsawn positions are so close to each other that AOIs of players from different teams are overlapping. Therefore players can see each other and it comes to endless fightings on the start position. This explains the problem named in Section **5.2.1 Upgrade Points** where Bots have massacred each other near the respawn position. Besides, this all has the effect that players stay in a limited area. However the world of Planet $\pi 4$ is potentially endless and this feature is not used by players if they stay in a limited area. Furthermore if a group of players leaves this area and captures some upgrade points far away from the respawn area. They will possibly never find each other again if one of them dies, since there is no way to share the own position with other players.

To solve this problem, the player needs more control on the position he is spawned at. If a player dies then a respawn control pop-up should appear while the player sees his ship explode. In this pop-up he should be able to select one of the team's captured upgrade points in a world map to indicate the position where he wants to be spawned. This feature would allow players to reinforce their team near all upgrade points which they own. However it should not be allowed to spawn in an upgrade point while it is being captured, since it would make it almost impossible for the other team to capture the upgrade point if enemies spawn inside it. However this would not solve the problem of finding specific players. To solve this, parties inside a team could be created. If the player is in a party his friends could be shown on the world map to make it easier for him to find the right upgrade point that he can spawn at. In addition the upgrade points could even allow players to teleport to other upgrade points. This would improve the dynamics of the game and again generate more complex network load, since players are teleported often and their AOI is changed frequently. More complex network load would rise the

significance of the benchmark. The teleportation should surely happen only between two upgrade points which are not being captured right now.

6.2.2 Upgrade Control

The improvements of the ships due to the upgrade point count of the team are fixed to the number of captured upgrade points. The upgrade points affect ship's health, weapon energy and maximal boost speed. A nice feature would be to allow players to give priorities on upgrades, for example a player could spend all his upgrade points on health, instead of energy or speed. It would be possible to include a menu for this in the pop-up described in the **6.2.1 Respawn Control** section. This feature would allow players to customize their ships, which is an important game mechanic of today's games. Also the ability to customize the own avatar makes a game much more appealing and personal. In addition, it would create more diversity in the player and bot behaviour, since the tactics will depend on the ship's capabilities.

However the ship customization mentioned above is limited to the current game session and does not generate additional traffic. Nevertheless the avatar customization needs to be taken serious. Massive Multiplayer Online Role-Playing Games (MMORPG) are very popular and have many players. The most important feature of this games is the creation of an own character. It is important to benchmark the performance of peer-to-peer networks on this issue. The network traffic would be permanent saving of character (player) data, it is similar to saving active game object data, but it is much more important to make this system reliable. Also cheating prevention and account napping could be interesting aspects. To create such a feature it would be possible to create some kind of credit earning, for example for kills and upgrade point captures. With this credits players could buy instant upgrades for their ships, the lifetime of this upgrades must be limited to the lifetime of the ship or some other timing like one or two days. Static objects like defence platforms could also be bought with same limitations like spaceship upgrades. Actually it could be possible to extend the game with different ships and weapons, which can be customized by the player. A good example for a shop system that sells items for a limited time is Battlefield Heroes[3]. In this free third person shooter weapons, armour parts and abilities can be bought for a limited number of days. The credits are earned when the user plays the game.

Another improvement would be to allow more upgrade types like weapon damage, fire rate, own shield regeneration or manoeuvrability. This is not possible now, because all possible upgrades are applied if one upgrade point is captured, the ships would be much too powerful after a few upgrades. However with the customization system one upgrade point could be spend only on one of the ship's features. This will make sure that ships are not getting too powerful with a low number of upgrades. Furthermore the upgrades should be categorized in branches like weapons, engine and shields to create a better overview on the large number of ship's features.

6.2.3 Messaging System

The bots of Planet π4's current version send messages between each other with information like enemy positions. In addition the team members inform each other about their positions. This information distribution is needed, because the AOI radius is relatively low and it would be hard to find other ships once a player is in an abandoned area. However there is no system for the human users like team chat, which is used by the bots. Therefore a chat feature should be created for Planet π4. However since the game is not designed to be played by humans except on presentations and conferences a fully functional chat is not needed. Instead a message system with predefined messages like „enemy nearby", „help needed" or „assistance for upgrade point capture needed" can be made. The messages should be associated with a position. For human users it should be possible to select a message and make the GUI place a waypoint on the screen, radar and the overview map. Furthermore these messages can be

visualized with icons on the world map of the pop-up discussed in **6.2.1 Respawn Control**. This feature would give the players (or bots) a good overview on the activities in the whole world.

6.3 Future Work First Impression

The next important step to improve the first impression of the game is sound. The game with sounds will attract listeners in presentations on conferences and make a much more professional impression. Fortunately Irrlicht allows easy integration of sound libraries and offers a list of such. Moreover there is the irrKlang[1] library which is free for non commercial projects and was developed by the same people who have created Irrlicht.

Also the graphics of Planet π4 are far from perfection. First of all the game needs visualisation of ship's damage. Especially the damage of enemy ships needs to be shown, since the health point of the own ship can be read in the HUD box. This effect would give the player feedback on their enemies state, which is important to calculate the chances to win a fight. The damage can be visualized via a health bar, but also with particles, for example burning wings when the ship is badly damaged. Furthermore the hit effect can be improved with a semitransparent shield animation when a bullet hits a ship. Besides the spaceships could use a trail effect like the ship in **Figure 3**. On the one hand it would work as another motion hint for the players ship and on the other hand it would be easier for the player to predict the motion of his enemies. In addition the HUD could be visualized with textured elements. This creates a much more professional impression than simple geometrical shapes. Finally the asteroids could be replaced with custom meshes instead of scaled spheres.

6.4 Future Work Alternative Approach

An alternative approach to study peer-to-peer networks for games is to offer a free peer-to-peer network library fitted to be used for games to a game developing community and use the feedback of the developers to evaluate the network architecture. For this purpose a game engine instead of a render engine like Irrlicht should be selected. For example Unity 3D[13] is a very powerful game engine, however not the features of this engine are the main reason to take it, but its community. The community is growing fast and the developers of the engine put great effort to let the community grow bigger. Even the most popular game developing companies like EA have used Unity 3D for their games. For example Tiger Woods Online[4] is an example of a good commercial game made with this game engine. The library needs to be integrated into the game engine. It must be easy to access and should not take long to install otherwise the users will not want to program with it. Also the library's interfaces must be easy to use, but complex enough to satisfy the requirements. Finally a test application needs to be done, it can be just a simple game. This is needed to show the developers that the network library works and can help them in their projects. The test game needs to be published on popular gaming sites like Kongregate.com.

When all this is done the PR work has to be started. Information on the library must be spread around the world. Fortunately this can be easily done through the internet. Posts on community cites, forums and even Facebook must be done. Also the library needs its own internet portal with a forum where all questions of the people need to be answered. If developers successfully create games then there will be no better prove that peer-to-peer networks are suitable for games. If one of the games that uses the library becomes popular, there will be many other developers who will create games and give feedback to the library and its performance. This approach will reveal information in a more realistic use case than a simulation of a self made game. However Planet π4 can also be released to the public, after some improvements. With some PR work it will gain a community with real players that generate traffic which can be evaluated. This would make it to a realistic use case, since it is then played by real humans.

References

[1] Ambiera. irrKlang, http://www.ambiera.com/irrklang/.

[2] Electronic Arts. Battlefield 1942, Battlefield 2 and Battlefield Heroes.

[3] Electronic Arts. Battlefield Heroes, http://www.battlefieldheroes.com/.

[4] Electronic Arts. Tiger Woods Online, http://tigerwoodsonline.ea.com/.

[5] A. Bharambe, J.R. Douceur, J.R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. *ACM SIGCOMM Computer Communication Review*, 38(4):389–400, 2008.

[6] Microsoft Corporation. Freelancer, http://www.microsoft.com/games/freelancer/.

[7] http://www.peers-at-play.org.

[8] http://www.spovnet.de.

[9] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

[10] Max Lehn. Implementation of a Peer-to-Peer Multiplayer Game with Realtime Requirements. Master's thesis, Technical University of Darmstadt, 22.10.2009.

[11] Max Lehn, Tonio Triebel, Alejandro Buchmann, and Wolfgang Effelsberg. Benchmarking P2P Gaming Overlays. Technical report, QuaP2P Workshop, Darmstadt, Germany, 2010.

[12] Ian Millington. *Game Physics Engine Development*. Morgan Kaufmann (Elsevier), 2007.

[13] Unity Technologies. Unity 3D, http://unity3d.com/.

[14] T. Triebel, B. Guthier, R. Sueselbeck, G. Schiele, and W. Effelsberg. Peer-to-peer Infrastructures for Games.

[15] T. Triebel, G. Guthier, and W. Effelsberg. Skype4Games. In Proc. of the 6th Annual Workshop on Network and Systems Support for Games: Netgames 2007, Melbourne, Australia, 09 2007.

[16] Dimitri Wulffert. Artificial Intelligence for a Massively Multiplayer Online Game, 31.03.2011.