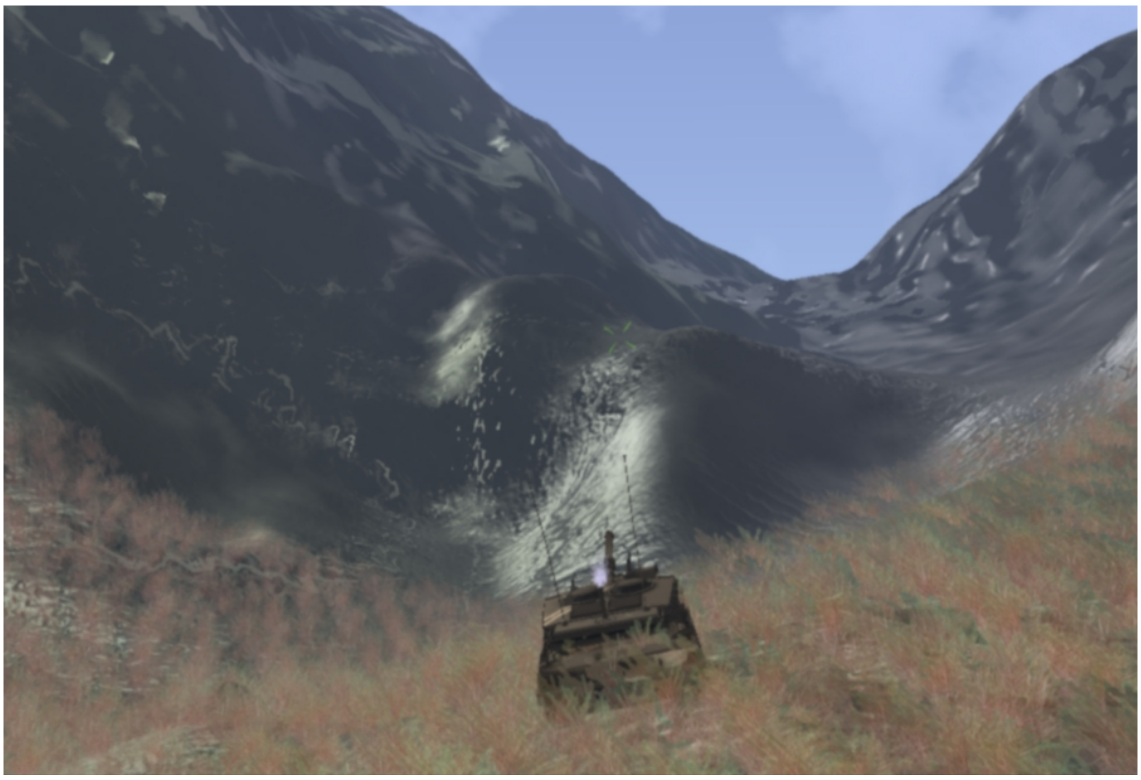
CHALMERS



TankAction

A 3D-Action Game with Network Capabilities

Bachelor's Thesis
Computer Science and Engineering Programme

SEBASTIAN BJURMAN DAVID KARLSSON
JOHAN KNUTZEN SHAHROUZ ZOLFAGHARI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Abstract

This bachelor thesis deals with the fundamental aspects of implementing a 3D computer game: real-time graphics, collision detection, game dynamics, network and design. We discuss our own solutions in relation to those used in contemporary games and in scientific papers.

This thesis shows that an iterative, incremental software process is well suited to game development. We demonstrate that quite simple methods can provide adequate solutions to problems that arise in game development.

Contents

1 Introduction **3**

2 Method **3**

3 Real-time Computer Graphics **4**
- 3.1 Terrain rendering 4
- 3.2 Lighting and shadowing . . 9
- 3.3 Sky rendering 13
- 3.4 Effects 16

4 Design **18**
- 4.1 Background 18
- 4.2 Method 18
- 4.3 Techniques 18
- 4.4 Comparison 27
- 4.5 Results and Conclusions . . 28
- 4.6 Discussion 28

5 Game Dynamics and Implementation **29**
- 5.1 Background 29
- 5.2 Techniques 29
- 5.3 Results and Conclusions . . 37
- 5.4 Discussion 38

6 Collision Detection **39**
- 6.1 Background 39
- 6.2 Techniques 40
- 6.3 Results and Conclusions . . 45
- 6.4 Discussion 46

7 Network **47**
- 7.1 Background 47
- 7.2 Techniques 47
- 7.3 Result and Conclusion . . . 54
- 7.4 Discussion 54

8 Results and Conclusions **54**

9 Discussion **55**

A Abbreviations **56**

1 Introduction

Games have become a huge and increasingly important industry. In 2005, the gaming industry annual income nearly reached the same levels as the movie industry, and the gaming market is growing rapidly. As a result of the increasing demand, more and more research has gone into improving the performance and visual quality of games.

The aim of the Tank action (TA) project was to create a visually impressive game in three months, by using relevant technology, a robust game engine architecture, and a suitable project model.

The requirements on the game was that it should be visually pleasant, relatively free of serious bugs, and provide multiplayer support over network connections. The problems experienced during the implementation can naturally be divided into five different parts:

Realtime Computer-Graphics Rendering realistic scenes at realtime frame rates.

Design Robust architecture without decreased performance.

Game Dynamics and Implementation Modeling the physics and laws of the game world.

Collision Detection A large subfield of game dynamics for determining whether collisions between objects occur.

Network To allow players to play in the same game world over a network connection.

2 Method

The TA team decided to use the Rational Unified Process (RUP) [1] model to organize development. The RUP emphasizes agile, iterative development with flexible planning and design. In the RUP, programming starts early on, in order to discover constraints and problems which are difficult to predict in purely theoretical design reasoning. The RUP is also highly configurable to the individual needs of the project, rather than being as strict as for instance eXtreme Programming [2]. The RUP, like other iterative project models, also recognizes the advantages of informal, frequent discussion rather than only formal meetings.

C++, due to its high performance in speed, and many available libraries is the de facto standard for game development, and was used in the TA project. Early on in the development process, the decision was made to make the game cross-platform. Simple Direct Media Layer (SDL) was chosen for a common interface for audio, keyboard, mouse, and 3D hardware via OpenGL. SDL was originally developed to port games from Windows to GNU/Linux, which are the main platforms which we wanted to support. Other libraries were used for loading e.g. images, mixing sound and loading 3D-models. The following libraries were integrated:

- SDL_image
- SDL_mixer
- SDL_net
- lib3ds
- Boost C++ Libraries

For synchronizing work and integration, Subversion [3] was used. This made concurrent work on different areas of the code possible and also served as a repository for shared documents.

3 Real-time Computer Graphics

In real-time applications, physically accurate algorithms for computer graphics are too slow. As a result, real-time computer graphics often employs empirical algorithms to implement features that can be seen in reality, such as shadows, lighting etc.

The choice of which features to include in a game is usually arbitrary, based solely on personal preference. One common method is to simply put a limit on minimum frame rate, and implement as many features as possible until the limit is reached. This approach was used in TA.

Below, the rendering is divided into four problems: terrain, lighting, sky and effects.

3.1 Terrain rendering

3.1.1 Background

Terrain rendering in real-time is a central and challenging problem in computer graphics. In fact, for a long time games simply avoided the problem altogether by selecting a setting for the game in indoor environments or dungeons.

Rendering of outdoor landscapes is challenging for several reasons:

- a significant amount of triangles are needed to cover a huge world with decent detail
- there is a risk for potentially huge amount of overdraw if the world contains many valleys and mountains
- it is difficult to perform occlusion culling, as the terrain is open, and should it be possible to perform occlusion culling in only some regions, frame rates may vary drastically during movement in the world
- there are drastic differences in light strength between bright and shadowed spots
- texturing level of detail is required to avoid high-frequency noise problems and too repetitive look in the distance
- atmospheric scattering (the sun light being reflected in random directions by particles in the atmosphere) makes realistic-looking lighting systems complex
- shadow generation techniques run into performance or precision problems when dealing with such large worlds
- generating a realistic-looking terrain can be more difficult than generating indoor environments with flat walls

The problem of generating a convincing terrain can be solved either by creating a mesh, or by so-called heightmaps - simple grayscale bitmaps showing the landscape from above, where the color determines the height of the landscape at that point. Both of these can be generated either automatically through fractals or similar solutions, or manually by an artist.

Most work within the field of terrain rendering has gone into developing faster algorithms for the heightmap approach, which is preferable for a number of reasons. Not only is it very easy for anyone to create a heightmap, but some forms of simplistic collision detection suitable to games is also easier performed against a heightmap than against a general, modeled mesh.

There are, however, two major drawbacks. First, most image file formats allow only 256 levels of grayscale, meaning a very limited vertical precision for the heightmaps. Second, it's impossible to render cliffs which hang out over the ground below. Both of these problems are however possible to solve, as will be demonstrated below.

3.1.2 Solutions to heightmap problems

The vertical precision problem of heightmaps can be compensated by filtering the height values with a Gaussian filter, and saving the heightmap in a custom file format with floating point values for the heights. This approach was used in the TA implementation. Alternatively, some of the existing floating point texture file formats that have been released recently, for instance OpenEXR [45], can be used. Another method is to simply use a very sparse heightfield or a terrain without very high mountains, but this is less elegant and less general.

The problem with overhanging cliffs can be solved by putting extra objects in the terrain on top of the heightmap. As long as the terrain doesn't contain excessive amounts of overhanging cliffs, this doesn't reduce performance notably.

In short, there are good reasons why the heightmap is the most commonly used method for representing and storing terrain, and why so much work has gone into optimization of heightfield rendering.

3.1.3 Rendering terrain

Below, some of the existing work on optimization of terrain rendering is presented.

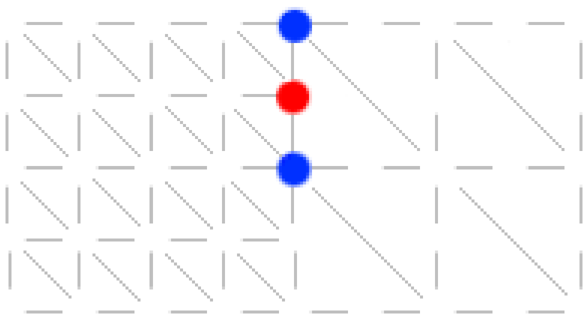
ROAM The acronym ROAM stands for Real-Time Optimally Adapting Mesh. This algorithm, published in 1997 [46], processes the entire heightfield mesh and merges triangles in an optimal way depending on the movement of the camera, and depending on whether the merging of two triangles will cause a small enough change that it won't be visible.

The algorithm works on a per-triangle level and the calculations are performed on the CPU, meaning it's not suitable to modern hardware since it doesn't allow batching, and competes with AI and game dynamics for CPU resources.

SOAR Several optimized versions of algorithms like ROAM have been proposed, for instance in [47]. A significant improvement in data storage layout ensures that despite performing per-triangle processing which makes batching impossible, the SOAR (Stateless One-pass Adaptive Refinement) algorithm, which is a simplification of the techniques described in [47] still performs well. However, it still has the disadvantage of competing with AI and game dynamics for CPU resources, as well as poor batching.

Geomipmapping A more batching-friendly algorithm for terrain LOD was proposed in [48]. The terrain is divided into a grid of patches, and different detail level versions of each patch are generated offline. During rendering, the version with as low LOD as possible that doesn't exceed a maximum tolerable height error is chosen to be rendered. The algorithm saves further CPU time by precomputing these distances (for the worst case, which occurs when the viewer sees the terrain patch from the side).

There are two problems with the algorithm. First, patches which contain at least one rough piece of geometry will not switch to lower LODs, which can mean the algorithm can't give any improvements at all for certain worst-case heightfields. Second, the junctions between patches with different LOD can contain visible "cracks" [48], see fig. 1, unless special measures are taken to prevent this. The solution proposed in [48] is to change index buffer whenever a patch is rendered. Another suggestion, given in [49], is to use "skirts", which are additional triangles that create vertical surfaces to cover the cracks. This saves the cost of computing different index buffers depending on the LOD of surrounding patches whenever a patch is rendered. A third method is geomorphing, which replaces the instant LOD switches by a smooth interpolation between two LODs [50].


(a) Heightmap patches of different LOD generated in geomipmapping. Notice the T junctions between the patch on the left, and the lower LOD patch to the right. If the vertex marked in red hasn't got exactly the same height as the average height of the blue marked vertices, cracks will appear

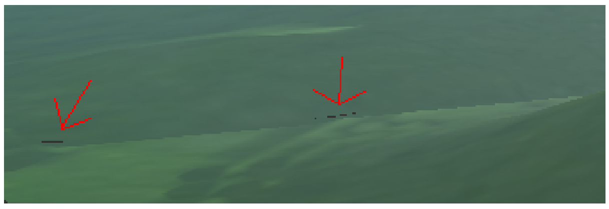

(b) Rendering of "cracks" artifact in a practical implementation of geomipmapping. Notice the holes, through which the gray background can be seen

Figure 1: *Demonstration of the problems that arise in geomipmapping*

Vertex texture fetch terrain Further refinements of the geomipmapping algorithm have been proposed. In [51], an algorithm that takes advantage of the recently introduced vertex texture fetch feature in modern GPUs is proposed. Previously, only pixel shaders were capable of reading from textures, but vertex texture fetch now allows looking up heightfield values in a vertex shader. This makes it possible to use a single vertex buffer for terrain geometry patches that build up the terrain, decreasing GPU memory requirements dramatically. The paper also proposes that terrain normals be calculated with vertex texture fetch in realtime in the vertex shader, using the simplified heightfield normal calculation method described in [52] (however, a normal map can also be generated offline, requiring only a single vertex texture fetch for acquiring the normals). The paper also uses geomorphing, not only to avoid cracks, but also to avoid "popping" artifacts which may occur when instant LOD switches are made [48] as an object passes the distance where LOD switch is scheduled to occur.

Geometry clipmaps Geometry clipmaps, as described in [53], switch between different detail levels purely based on distance, using vertex texture fetch to look up height values. The mesh is precomputed, centered around the camera, and more sparse further away from the camera. The texture lookup coordinates are calculated in the vertex shader, and the terrain can be drawn in a single Draw call. While it gives more visible errors than geomipmapping to have LOD choices based on distance rather than maximum error, the almost uniform rendering cost and worst-case performance guarantees may be preferable in many cases.

The implementation uses geomorphing to avoid major popping artifacts, but the distance based LOD selection means that vertices will be moving in a way somewhat reminiscent of popping, but much more slowly.

The geometry clipmap algorithm can be performed entirely on the GPU, as shown in [56]. Similar techniques, but with a spherical instead of square grid have been used in water rendering in [54]. Spherical grids have also found their way to terrain rendering in [55] because they're better suited to performing view frustum culling.

3.1.4 Results

In the TA implementation, a heightmap was used to generate the terrain. A heightmap size of 1024*1024 provided more than sufficient horizontal precision.

To overcome the problem of lacking vertical precision, a custom file format storing each height sample as a floating point value was used. The grayscale bitmap used as data source was filtered through a Gaussian filter in order to smoothen it to take advantage of the potential of the floating point representation. Although a Gaussian filter reduces overall sharpness, using a narrow Gaussian curve was found to preserve the shape of the terrain well, while removing local sharpness caused by too high degree of quantization in the original 256 level heightmap.

Gaussian filters are symmetric and as a result can be applied in two passes, one vertical and one horizontal [57], thus making it possible to reduce its complexity (where $n * p$ denotes the image resolution) from the $O(n \log n * p \log p)$ of general convolution for two-dimensional signals (using Fast-Fourier transforms [58]), to the much faster $O(max(nlogn, plogp))$. For a narrow Gaussian curve, values far from the center of the kernel are so small that they can be ignored without major differences in the result. Cutting off such values that contribute little to the end result can reduce complexity to $O(max(n, p))$. Despite this, the filtering was considered too time consuming and the processing had to be performed beforehand to decrease loading times.

The TA implementation found that sufficient performance for rendering a 1024*1024 heightfield could be achieved without any geometrical optimization techniques. Rendering the entire heightfield by brute force in a single Draw call and without any view frustum culling was one of the most expensive parts of the rendering of the scene, however.

Texturing was done by blending four textures together. Two of these were appearance textures, one large texture stretched over the entire terrain, blended together with a detail texture repeated 10 times over the terrain. The other two textures were normal maps applied with the same texture coordinates. The detail normal map had to be gradually blended out in the distance in order to avoid high-frequency noise, especially when applying lighting.

The terrain was given a more convincing look by the addition of waving grass as described in [59], and by inserting various objects on top of the terrain. Grass was planted in the terrain according to a special grassmap, a bitmap where black indicated grass and other colors were interpreted as no grass. This grassmap could be of a different size than the heightmap to make the grass density independent of the heightmap density. The grass was blended out gradually in the distance, and divided into batches which were culled when far away enough to be fully blended out.

3.1.5 Discussion

A uniformly applied Gaussian filter may not be the best way of generating floating point heightmaps from grayscale bitmaps. A better option could have been to use a level editor, since image processing programs don't provide support for more precision than 256 grayscale levels, and the Gaussian filter doesn't give full control over the smoothing process.

Level editors are however difficult and time-consuming to implement. A faster way to allow generation of local sharpness may be to generate the heightmap from two bitmaps - one for the actual heightfield, and another containing sharpness values. The heightmap offline processor could interpolate between the filtered and unfiltered height value for each vertex, with interpolation weights determined by the degree of sharpness.

If there had been more time, the TA implementation would also have implemented one of the more recently developed algorithms for geometrical level of detail, for example geomipmapping or radial geometry clipmaps. As more and more features were implemented in the terrain shaders, the performance costs of the overdraw and unnecessarily high detail level even for distant terrain became one of the bottlenecks in the engine.

Since the geometrical LOD algorithms require the terrain to be divided in several batches, view frustum culling could have been easy to add as well. A radial grid clipmap approach could have achieved all of this without increasing the number of Draw calls, however at the cost of a distance-based instead of maximum-error based LOD switch.

3.2 Lighting and shadowing

3.2.1 Background

Adding lights and shadows makes a dramatic difference to how convincing a scene looks.

Physically accurate lighting through methods such as ray-tracing are still considered too performance-consuming to be feasible in real-time, even though promising progress has been made in the field recently in for instance [60]. As a result, real-time applications are still restricted to simpler lighting systems.

Some of these lighting techniques incorrectly calculate light in some of the spots that should be shadowed. This has to be compensated by combining the results of the lighting algorithm with the results of a separate shadow algorithm.

3.2.2 Lighting techniques

Below, several lighting systems for real-time rendering are described.

Lightmapping Lightmapping essentially means blending precalculated light textures onto surfaces. The lighting precision is limited by the skill of the artist, and the amount of texture memory available. The technique is further limited by its poor support for dynamic scenes with moving light sources and moving shadow casting objects.

Although it's possible to enable support for dynamic scenes with moving light sources according to [61], the accuracy is low. Shadows are also inaccurately represented by this technique, and the technique may need to be combined with separate shadowing techniques. The enormous inaccuracy in practice, and the lack of support for fully dynamic environments, limits the usefulness of this technique today.

Phong lighting model The Phong lighting model is an empirical, non-physical model of lighting. It calculates light based on the normal of surfaces compared to the location of the light source. Thus, objects not facing the light source are correctly shadowed, but objects occluded from the light source but facing it are not (fig. 2). This means that in order to achieve accurate shadowing, a separate shadow algorithm must be used and its result combined with the light calculated by the Phong model. See [62] for a complete description of the Phong model lighting equations. The Phong lighting model has good support in current hardware and together with a shadow algorithm is accurate in most aspects except for its lack of secondary reflections.

Bump mapping [63], used to make the objects look more detailed without making the geometry more complex, can easily be combined with the Phong lighting model. Bump mapping is less accurate than other suggested techniques, for instance parallax mapping [64]. The original bump mapping paper [63] also lacks support for bumps that shadow other bumps, a problem whose solution is discussed in [65].

Ray-tracing Ray-tracing is physically accurate since it traces photons as they collide on surfaces, but as mentioned above, the technique can still not be performed in real-time. It can be combined with bump mapping and similar techniques.

As of now, the Phong lighting model seems to be the most commonly used lighting model. Lightmapping is too inaccurate (and not possible to combine with bump maps) and doesn't give enough performance gains compared to Phong lighting to be justifiable today, and ray-tracing is still intractable for real-time applications. However, there's room for using slightly different light equations for combining the diffuse, specular and ambient terms in the Phong lighting model; this was done in the TA implementation.

3.2.3 High-dynamic range

As mentioned in 3.1.1, there is a significant difference in light strength in outdoor landscapes. An accurate lighting model must account for this high-dynamic range of light strengths.

The introduction of HDR lighting techniques in computer graphics [66] allows internal rendering the scene to render targets with a significantly wider allowed value interval for colors (the entire range of floating point numbers, instead of only [0..1]). The colors are then scaled by an exponential function with a base smaller than 1.0 to compress the color values non-linearly, a process known as tone mapping. The resulting scene is usually perceived as more realistic, and the technique is also easily combined with bloom filters [67], which make strong lights in the scene "bleed out" light into surrounding pixels, see fig. 3.

Although HDR lighting makes it more difficult to predict how the colors of the final image will look - which may be a problem for content-creating artists - many games today have considered the benefits stronger than the disadvantages.

3.2.4 Shadowing techniques

There are two major shadowing techniques in use today: shadow volumes [68], and shadow maps [69].

The shadow volume algorithm processes individual triangles in the scene to create polyhedra which contain the entire volume behind each shadow-casting object, then render these polyhedra with a stencil buffer to mask regions that will be shadowed.

The Shadow mapping algorithm creates shadows in two render passes. The first pass renders the scene from the light source and stores the depth values of the geometry in a texture render target (called the shadow map). The second pass renders the geometry from the viewer. Each point is transformed into the space of the light source, and compared to the depth value stored in the shadow map. If the object was further from the light source than the depth value generated in the first pass, the object is shadowed.

Shadow volumes suffer from being CPU intensive, thus competing with AI and game dynamics in game engines, and they aren't batching-friendly because they process individual triangles. They do however result in very exact shadow edges and suffer from no aliasing problems. The shadow volume algorithm can also be extended to allow rendering of soft shadows as described in [70].

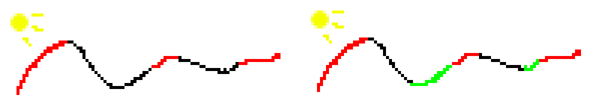


Figure 2: *Phong lighting compared to physically accurate lighting. The left picture shows the desired result, with red indicating lit areas and black indicated shadowed areas. In the picture to the right, the results of Phong lighting are shown. Notice the green areas, which will incorrectly become lit even though they are shadowed by the hill.*

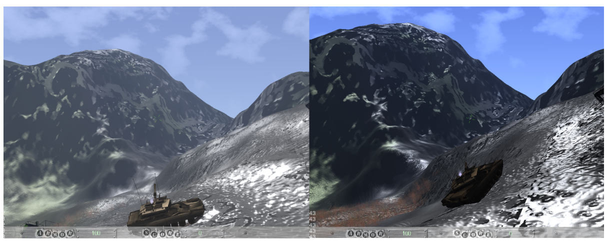


Figure 3: *Left: scene from TA rendered with HDR and bloom. Right: scene from TA without HDR and bloom. Notice the feeling of a greater light intensity of the specular highlight to the left, due to the bloom filter. Also notice the different scaling of color values achieved by HDR tone mapping. The image without HDR has sharper contrasts between lit and unlit regions, and looks more artificial.*

Shadow maps work well with batching, but suffer from aliasing artifacts because the shadow map has limited resolution. An especially problematic situation is the "dueling frusta" case as described in [71], when the light is shining towards the viewer (because the shadow map has the least precision where the second render pass has the maximum precision), but there are also aliasing artifacts when the light source is nearly parallel to the geometry to be lit [71], see fig. 4.

Several solutions have been proposed to deal with the shadow map aliasing problems, for example PCF filtering [72] in which the results of several depth comparisons are combined, which also results in soft shadow edges as a bonus. Another method is to render multiple shadow maps next to each other [73] in the first pass, but this is performance-consuming. An improved version with adaptive resolution is proposed in [74]. A different approach, which doesn't decrease performance compared to the original shadow mapping algorithm, is described in [71].

Shadow maps can be used to generate soft shadows by using Percentage-closer filtering (PCF), or blurring the shadow map with a Gaussian filter in additional render passes [75].

3.2.5 Results

In the TA implementation, the Phong lighting model was used, however simplified for a single light source (the sun), and without support for colored lights. The objects in the world were given a rougher surface by bump maps, which despite their inaccuracies were found to be a good compromise between effect and performance. Specular lighting was improved by HDR lighting with bloom effects implemented by two Gaussian passes, one horizontal and one vertical [76].

Shadows were implemented using the basic shadow mapping algorithm [69], but to simulate a directional light such as the sun, the first pass (from the viewpoint of the light) was rendered with an orthogonal projection instead of a perspective projection. This didn't add any additional complexity to the algorithm, because the second pass could still be performed by multiplying the world space position with a matrix $M = V_l P_l$, where V_l denotes view matrix for the light, and P_l denotes the projection matrix for the light, as in the original algorithm. PCF and a Gaussian filter were applied to achieve softer shadow edges.

Because the depth comparisons between the shadowmap and the point calculated in the second pass are floating point comparisons, a small hardcoded bias had to be applied to the comparison to avoid artifacts. These artifacts were caused because the shadow-casting object will have approximately the same depth value as the value in the shadow map, and the floating point precision sometimes makes the shadow-caster depth in the second pass slightly greater than the depth it wrote in the first pass, see fig. 6.

Using the maximum texture size capability of modern graphics cards barely gave sufficient shadow map accuracy for the first pass, even when using PCF and Gaussian blur in the second pass, because the terrain was too large and perspective shadow maps and similar aliasing-reducing techniques weren't used. The Gaussian filter addition slowed down the execution dra-

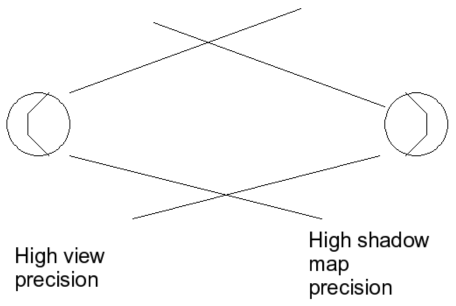


Figure 4: *The dueling frusta case in shadow mapping. When the frusta are facing each other, the shadow map precision is minimal where the view precision is maximal, resulting in the worst possible result.*

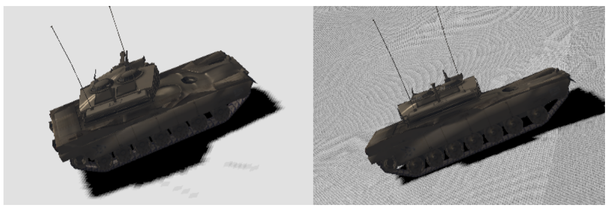


Figure 5: *An illustration of the problem of precision in shadow map depth comparisons. The right image shows a pure implementation. Notice how areas that should not be shadowed, due to numerical precision limits, sometimes incorrectly are calculated as shadowed. These numerical imprecision errors are more or less randomly located in the scene, giving a visually unpleasant appearance, and may also cause very visible flickering by appearing in different spots at different times as the user moves through the scene. The left image shows the result of applying a small bias for unshadowed to the comparison. The depth comparison artifacts are gone (but horizontal lack of precision due to limited shadow map resolution remains).*

matically because the addition of a Gaussian filter meant that the shadow map couldn't be applied to the final scene in the second pass. Instead, the second pass generated a "shadow mask" containing the shadows of the scene in post-projective space. A third and fourth pass performed Gaussian blur on this mask, while a fifth pass blended the blurred shadow mask onto the image when the normal geometry was rendered in its final pass.

3.2.6 Discussion

The aliasing of the shadow maps was a much bigger problem than perceived at first. With more time, an implementation of perspective shadow maps could have been added. Future hardware may also provide support for textures with higher resolutions. It was also found that shadow mapping in combination to HDR led to an extensive usage of GPU texture memory. Many of these render targets could have been reused, resulting in less memory usage and better cache behavior for the GPU.

It is difficult to assess the possible performance gains that could be made from this. Moreover, in order to be supported on the hardware and software platform on which the game was developed, unnecessary memory waste was caused by using floating point RGBA render targets instead of a single floating point output value. Explicit planning of render target surface usage and sharing could become an important optimization technique in modern graphics engines, now that an increasing amount of algorithms use several passes with offscreen render targets to achieve interesting effects.

Additional problems were caused by the sky dome (see 3.3.1). Depth precision is an issue in rendering large outdoor terrains with close up detail because the quotient between the view frustum far plane distance and the near plane distance becomes large, making numerical precision lower. To have a fixed sky dome position in relation to the terrain would require a further away far plane. On the other hand, sky dome moving with the viewer would require the sun lightmap projected on the sky to move with the sky dome in order to avoid artifacts giving an appearance of the sun moving back and forth across the sky. With the sun and sky dome moving, it felt natural to move the position of the light source used in the shadow map passes accordingly. However, this resulted in problems since different regions would switch between being shadowed and lit as the viewer moved around, rather than being based on the day-night cycle and sun movement alone. It was eventually considered accurate to let the shadow map light source position be fixed in relation to the landscape, while moving the sun texture and sky dome. For a directional light source such as the sun, this is also the most logical choice.

3.3 Sky rendering

3.3.1 Background

Rendering a realistic sky is at least as important as an accurately rendered terrain, because the sky covers a very large percentage of the screen surface, and the appearance of the sky affects the overall perception of a scene dramatically. Accurate atmospheric scattering simulations for sky rendering are expensive and difficult to perform on the GPU. As a result, similar to general lighting techniques, physically accurate models have often been ignored in favor of empirical models.

Sky is usually rendered by a textured or colored box, plane or half-sphere. Rendering with a half-sphere has largely replaced the sky box method, and is also the method into which most research has gone lately. Rendering with half-spheres, a.k.a. sky domes, has several advantages:

- The triangle processing speed of modern hardware is so high that a dome requiring more vertices than a box doesn't cause a significant reduction in performance
- The atmosphere is reminiscent of a half-sphere, not of a box. The square shape of a sky box may become visible when the user rotates in the scene unless special more advanced texture generation techniques are used, in which case using a sky box may cost more performance than using a sky dome.
- A sky dome gives simple, intuitive equations for calculating color and texture attributes. For instance, texture coordinates for a cloud texture can be generated by simply scaling the x and z coordinates of the sky dome vertices to the [0..1] interval.

3.3.2 Skydome coloring techniques

Nishita *et al* [77] presents equations for coloring a skydome based on physically accurate atmospheric scattering models. However, the method has been considered to be too costly in terms of performance to become the method of choice used in games. O'Neil in [78] presents a version in which the equations have been simplified greatly by replacing costly computations by approximately identical, simpler functions, ignoring insignificant terms. Still, the method may be too performance-demanding for average games, compared to simpler empirical sky rendering systems which can be implemented more quickly and produce sufficiently accurate results. Another problem with accurate scattering models is that there is little theory on how to combine them with clouds in a realistic manner.

The empirical sky coloring models aren't well documented in any scientific papers, but are used extensively in contemporary games. If the game engine doesn't need dynamic environments and day-night cycles, the problem of sky dome coloring can be solved by simply letting an artist generate a texture which is projected onto the sky dome. This can give higher precision than per-vertex calculated atmospheric scattering simulations at minimal performance cost.

For dynamic environments, more complex systems have to be devised. One implementation suggestion, see [79], is based on a two-dimensional LUT texture containing sky dome colors which have been acquired from a real sky through the use of a digital camera. Interpolation is then performed between these color values, based on time of the day and sky dome height.

3.3.3 Cloud rendering

Adding clouds makes a significant difference to sky rendering techniques, and makes for much more realistic scenes.

There are many ways of generating clouds. One is to let an artist model cloud meshes, but such clouds are difficult to animate, may look repetitive, and takes a lot of time from the artist, but in return can be lit accurately. Another is to model volumetric clouds as particle systems. These can be animated to any degree of complexity that may be desired and it doesn't look repetitive if done well, but particle system clouds are difficult to light accurately, and can be quite performance intensive. A third method is to use procedural, dynamically generated cloud textures to project onto the sky dome. Some theory behind procedural textures is given in [80] and [81]. A procedural approach to rendering clouds is described in [82], and can easily be extended to animated clouds.

With procedural clouds it is also easy to avoid a repetitive look, the rendering is inexpensive, and no work at all is required by the artist. Simple tweaking of parameters can adjust cloud thickness and whether the clouds will look more like cumulus or cirrus, for instance. The main drawbacks are the lack of complete control and difficulty of lighting the clouds accurately.

3.3.4 Results

The TA implementation used Perlin noise clouds projected onto a sky dome. The sky dome was colored with a simple interpolation between a top color and a bottom color. A LUT stored the top and bottom color for different times of the day. A simple linear interpolation between the LUT values was used to make the changes during the day-night cycle smooth. A sun lightmap was rendered by drawing a simple billboard onto the sky dome with additive alpha blending. The effects of atmospheric scattering on the world were approximated by blending far way terrain and other objects with the sky dome bottom color.

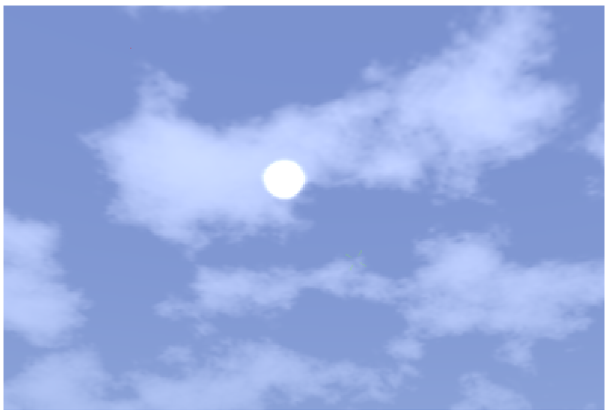


Figure 6: *The final result of rendering the sky in TA with Perlin noise clouds, top and bottom color interpolation, additive sun lightmap, and HDR tone mapping.*

3.3.5 Discussion

The simple implementation used in TA provided decent results, but there were some aspects which could have been improved without switching to more complex scattering simulation implementations. One such improvement would have been to calculate an aura of light around the sun. Looking at a real sky, it is apparent that the sky is significantly brighter in the third of the sky that is immediately adjacent to the sun. Another improvement could have been to adapt the HDR system to the angle of the viewer, so that if the sun ends up in the user's line of sight (a simple view frustum culling test), tone mapping exposure would increase and give the viewer a feeling of being dazzled.

The next step of improvement may very well be to adopt one of the scattering simulation methods. Since the approach in [78] is performed entirely on the GPU, there are no batching problems. Since the sky dome is usually spherical and follows the viewer, and there is a maximum pitch angle for the viewer, only a sector of the sky dome will need to be rendered each frame. This means a clever memory layout can allow view frustum culling to be applied as well without any notable overhead or batching problems.

3.4 Effects

3.4.1 Background

Graphical effects such as fire and smoke are important to make a scene look more alive and realistic. However, they are difficult to implement realistically for several reasons:

- The effects are often caused by incredibly small particles
- The number of particles is huge
- Semi-transparency is difficult to simulate
- Smoke and fire may cause refraction of light that passes through water vapor

The de facto standard method for rendering special effects particles is the Particle Systems method as introduced in [83]. Particle systems consist of an emitter from which particles are created and sent out in the world. Once emitted, the particles describe a movement defined by attributes specified at the time of their emission.

3.4.2 Techniques

Real smoke contains too many particles to be feasible to simulate in real-time with completely physical models. Instead, the particle systems are often abstracted with a smaller number of larger particles. These larger particles can either be simulated by point sprites, which are fixed-size screen-aligned quads, or by billboards, which are large, textured screen-aligned quads.

Nguyen [84] for instance uses the billboarding approach, with animated textures to make up for the too small amount of particles. This method also solves the problems caused by the limited size of particles. The main problem with billboards is that they are difficult to light.

Semi-transparency is typically implemented by using alpha blending. Alpha blending may give rise to artifacts if the objects aren't rendered in back to front order, see [85]. Therefore, it may be necessary to sort objects depending on view depth each frame. This requires an efficient memory management strategy, and may also prevent buffering since the vertex data must be sorted each frame. Other approaches than sorting have been suggested to avoid artifacts, see [85], however they require additional render passes.

Refraction is usually ignored in modern special effects implementations because it may require additional render passes, see [86], or ray-tracing approaches.

3.4.3 Results

The TA implementation implemented a particle manager reminiscent of the system described in [87]. A particle system class

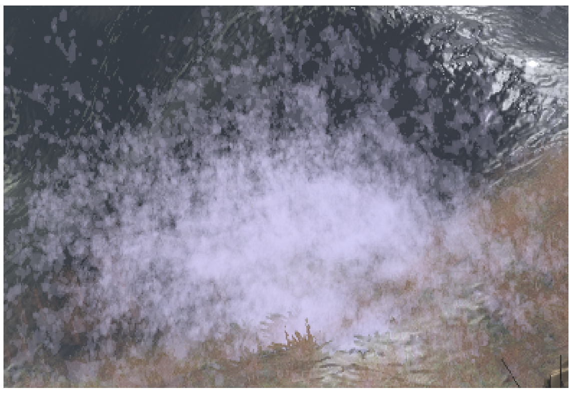


Figure 7: *Rendering of a smoke shell particle system in TA. The particles are implemented by only around 200 billboard particles.*

manages the creation and deallocation of particles for an individual effect, a particle class manages movement of already existing particles and signals to the particle system class when it has faded out completely and can be removed. Finally, a particle manager class is used to manage particle systems. Effects that die after a pre-specified amount of time, such as an explosion, are automatically deallocated by the particle manager, while other effects can be removed by explicit calls to the particle manager.

Sorting of particles was carried out each frame by the sort() function implemented in the standard template library (STL) in the C++ standard. STL sort() uses the introsort algorithm [88], which is a combination of heap sort and quick sort, see [89].

Billboarding was implemented on the GPU in a vertex shader, in order to offload the transformation to special-purpose hardware, however one transformation had to be done on the CPU in order to calculate the depth of each particle to be able to perform the sorting. In order to be able to draw all particles in a single Draw call, all particle textures were stored in a single texture atlas.

In order to synchronize the particles with the rest of the day-night cycle, the output color was multiplied by a factor approximately depending on sun height.

3.4.4 Discussion

The implementation used in TA unnecessarily restricted the number of textures available to rendering by being able to use only a single texture atlas. However, this approach made it possible to render all particles in the scene in a single Draw call. If this approach had been replaced by an approach allowing more textures, a general and correct sorting would give no guarantees that the number of Draw calls wouldn't increase to the same as the number of particles in the scene. To allow for more textures then, it would be necessary to either use an array of several textures, or simply restrict the accuracy of the sorting. One possibility could be to use exact sorting within each particle system, and sort the particle systems after emitter position.

4 Design

4.1 Background

When designing a game, there is certainly no silver bullet. The larger the game, the more *intricate* details proliferate, and one realizes that there is a need for organizational structure. With this in mind, it is still very important not to over-design in the early stage of development, as not to restrict nor cripple the development itself. The design of TA was meant to be very clean and adaptable and does not incorporate a scene-graph, as is usual when designing a game of larger proportions. Scene-graphs are elaborated on in 4.3.3 but for a more elementary introduction, see [4, "Scene Graph Basics", Chapter 19].

Instead of relying on a scene-graph, most components of the game were rather designed with general design patterns. This allows for greater flexibility and adaptability, and will also make understanding the concepts of the design for experienced programmers easier.

However, due to the lack of a common rending interface, this choice of design made the merging of components difficult at times. A scene-graph is a powerful alternative, which will be explained in more detail as we proceed.

4.2 Method

In the initial design of TA, much of the design and planning was done parallel to the development. This allowed patterns and organization of software objects to be incrementally established.

As such, it eased the process of adding new code and new components to the game. When newly programmed components reached maturity, they were gradually integrated into the system. This allowed for a testing period in which components could be properly structured and conceived.

Moreover, by following this paradigm, one does not need to make critical nor final decisions in the early design phase, but instead encourage frequent and iterative updates to the design architecture as work progress. Much effort was also put into creating a robust and flexible framework in the system. This provided programmers an interface early on, which they could use to dynamically load concrete objects such as textures, 3D-models and shader source code.

Furthermore, by ensuring a static interface for programmers to work with in the early design phase, the underlying implementation could dynamically be changed. This proved very helpful at times, such as when there was a need to replace the underlying 3D-model loading library.

4.3 Techniques

4.3.1 The Managers

By investing time and effort early on in creating a robust framework, coding of new components proved to be an efficient task. The most noteworthy fundamental components which comprise the framework of TA are as follows:

- Texture Manager
- Shader Manager
- Mesh Manager
- Sound Manager

These managers comprise the interface from which any software object in the system may request resources.

Essentially, they all work in the same way, except that they provide different resources. The motivation behind having 'managers' to provide resources derives from the need to share resources. In particular, sharing the memory needed for example vertex- and texture-coordinates can be *crucial* in order not to exhaust the **video memory** on the graphics card.

To elaborate further on how these managers work. Let's consider that a particular software object requests a 'mypicture.jpg' from the Texture Manager. The Texture Manager would first match the string 'mypicture.jpg' in a **map data structure**. Them, in the case of a mismatch, the picture would be allocated from scratch into video memory.

Otherwise, a **shared pointer**[1] to one which has already been loaded into video memory is returned. By making use of shared pointers, TA ensures deallocation of memory automatically. The program in fig. 8 demonstrates an example request to the Texture Manager.

The benefits of having clever managers are many, and do not strictly only surface when it comes to the creation and handling of video memory on the graphics card. In fact, when intuitively used as **singletons**[2], they provide a central point of resource allocation for all software objects, regardless of coupling.

```
initialize a map m
        from string to TXHandle;
TXHandle LoadTexture(string fn) {
if(fn was present in m) {
      m[filename] = a new OpenGL
      texture generated from fn;
}
return m[filename];
}
```

Figure 8: *The function used to register textures in the Texture Manager. A TXhandle is a shared pointer to a OpenGL texture.*

One useful aspect of these managers is that one can allocate *all* needed resources at initial runtime so that later dynamic creation will only involve shared pointers.

An example of this is the creation of tanks in the game. TA provides a **TankType** structure which holds pointers to resources used by a tank.

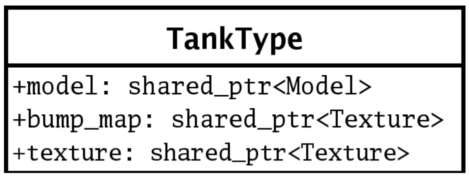

TankType
+model: shared_ptr<Model>
+bump_map: shared_ptr<Texture>
+texture: shared_ptr<Texture>

Figure 9: *A simple TankType containing shared pointers to shared memory.*

Fig. 9 depicts a simplified TankType structure of the one which is used in the game. Other variables such as hit points, weapon information and maximum velocity are hidden from view, but are naturally needed to represent a tank.

Moreover, by compiling all necessary shared memory objects into a well defined structure, one allows allocation of all tank

[1]A type of smart pointer which holds a counter for every listener and self-destructs if that counter becomes zero

[2]For more information about design patterns, see [11].

types to be loaded initially - when the game is started. Furthermore, if two tanks utilize the same texture or 3D-Model for instance, this would only be loaded when first requested. Therefore, the programmer using the TA framework does not need to bother to check specifics such as if a texture, shader, or 3D-model has already been requested.

Since this framework spans across all aspects of the game, a tank model which is used to represent the player can also be used, without extra memory usage, as a decorative static object[3] in the terrain.

4.3.2 The Object System

By further embracing shared memory and its advantages, more complex object structures can be designed. For instance, a general loading scheme for loading a 3D-model, textures and shader parameters could be easily realized.

TA uses a scheme called **The Object System** to gather common procedures and make use of inheritance to generalize common concerns of rendering an object in the game.

An **object** in the game is simply something which is transformed and rendered. An object of this sort will simply be denoted as an *object* in this section, as to not confuse the reader. When trying to understand the Object System's way of loading objects, it's helpful to think of it as a hidden class - of which implementation is not of important.

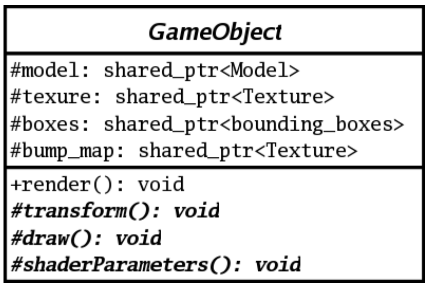
To this hidden class's constructor, one could, for example, supply filenames for the 3D-model, texture and bump-map. Given this information, the Object System would generate *k*d-trees[14], OBBs[15], OpenGL VBOs[4] and OpenGL textures.

To provide this usability, The Object System gathers common procedures and separates concerns on a level chosen by the programmer. What this actually means is easiest to demonstrate with a figure. Fig. 10 depicts how inheritance can separate different concerns of the process of loading and rendering an object. Fig. 10(a) depicts the members of the base class GameObject, which are initialized on creation. Note that this class is abstract and is not possible to instantiate.

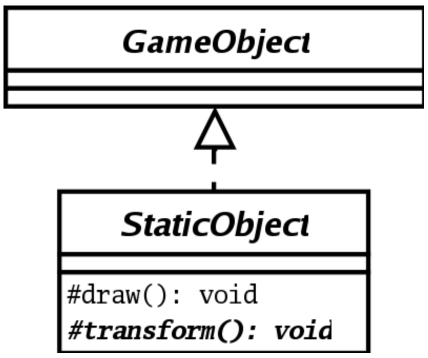
Generally, TankObject's responsibility is to initialize its members, though this alone does not justify its existence. More importantly, what the class GameObject generalizes is actually the common steps of what one really does when rendering. GameObject disregards the implementation of the helper functions it uses when rendering, and instead reserves these functions as abstract for the programmer to implement by extending GameObject. Note that this is a simplified version of the Object System; in the real implementation we need three different types of render procedures because of the shadow mapping. This further justifies the need for abstraction of common functionality. Moreover, describing just **render()** is sufficient. The two other render passes, which shadowmapping requires have similar functionality. What these two are, is clarified in 3.2.4.

[3]A 3D-model which is unmovable and placed arbitrary in the terrain.

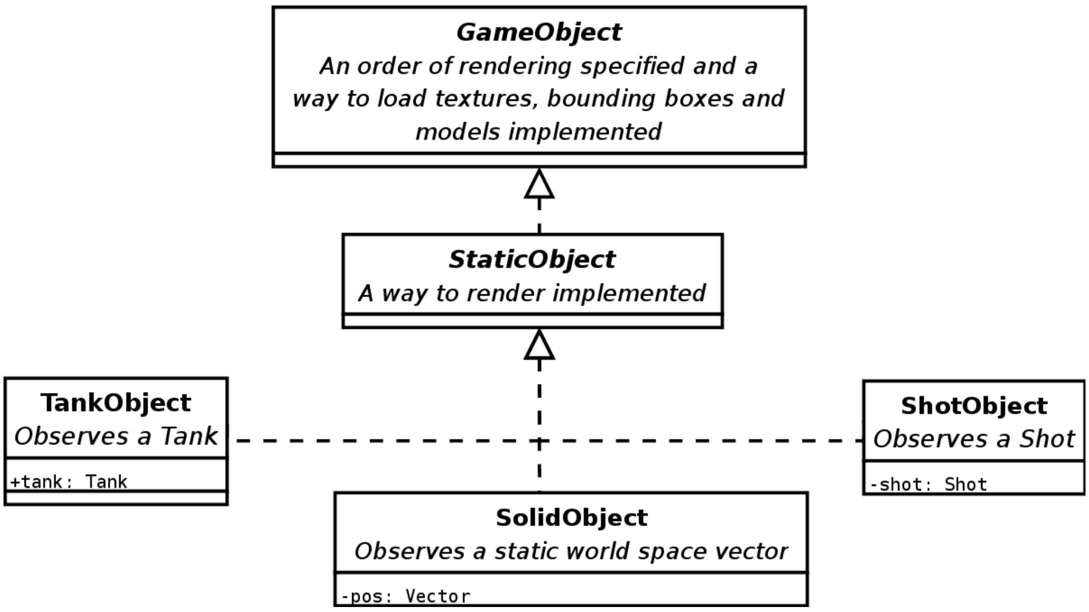
[4]Vertex Buffer Object, for a more detailed description - see [4, "3D-Modeling and Object Composition", Chapter 9].


GameObject
#model: shared_ptr<Model>
#texure: shared_ptr<Texture>
#boxes: shared_ptr<bounding_boxes>
#bump_map: shared_ptr<Texture>
+render(): void
#transform(): void
#draw(): void
#shaderParameters(): void

(a) A GameObject which stores various shared pointers. The render() operation, provides an order of which parameters are set. The transform(), draw() and shaderParameters() are left abstract.



(b) A StaticObject provides a way of rendering the GameObject. The transform() functions is left abstract so a StaticObject can not be instantiated. The draw() function on the other hand, is implemented and outlines the draw calls to 3D-model's meshes.


GameObject
An order of rendering specified and a way to load textures, bounding boxes and models implemented
StaticObject
A way to render implemented
TankObject
Observes a Tank
+tank: Tank
ShotObject
Observes a Shot
-shot: Shot
SolidObject
Observes a static world space vector
-pos: Vector

(c) Concrete classes implement the StaticObject and provide a way of transformation. These three classes are ones which extend from StaticObject. They have implemented the transform() function and are eligible for instantiation. SolidObject in the middle, defines transform() reflecting it's transformation in world-space. The TankObject to the left works a bit different. TankObject's transform() function differs depending on it's member field, **tank**. This tank member field, is of the class Tank. The Tank class includes quaternions and a vector which in conjunction define transformation. Lastly to the right, The ShotObject class. The ShotObject class listens to an instance of a Shot class. The Shot class comprises a quaternion for it's rotation, and a vector for translation. These two outline the transformation of a ShotObject.

Figure 10: *The hierarchy of the GameObject system.*

Refer to fig. 11 which illustrates the basic outline of how render() defines the order of how a rendering proceeds.

```
void render() {
setShaderParameters();
// Iterate over and render all meshes
for i = 0; i < model.numMeshes; i++) {
    glPushMatrix();
    transform(i);
    draw(i);
    glPopMatrix();
}
}
```

Figure 11: *The render() function in GameObject.*

As a first step, set the shader parameters which are the same for every model to be rendered. Thereafter, since a model may consist of many meshes, loop through each mesh and apply their unique transformation and draw call. In this case, because transform(i) modifies the OpenGL modelview matrix, we need to push and pop the stack in order not to apply the same world space transformation equally on each mesh. How matrices work, and how the OpenGL modelview matrix stack works is well described in [5, "Page 83", Chapter 3].

It could be questioned which shader the parameters render() actually is giving arguments to with setShaderParameters(). GameObject does not provide an answer to this. What is necessary, though, is that *some* shader has been set in the OpenGL context before any GameObject is rendered.

The reason for not applying a **shader switch**[5] before setting the parameters for each GameObject is related to cost. The cost of switching shader context is one of the most expensive state changes one can do in OpenGL, as these kinds of state changes should always be done in batches [6, "Shader Management", Chapter 6].

GameObject instead relies on the **creator of the GameObject** to do a shader switch before calling render(). This implies that this so called creator should try and keep as many GameObjects with the same shader as possible in grouped batches. Hence, minimizing the amount of shader switching needed per frame.

As illustrated in Fig. 10(a), GameObject also supplies a few abstract functions which need to be implemented. These functions are used obliviously by GameObject, in hopes of an object supplying them by inheritance. Now, let's move on to what Fig. 10(b) illustrates. Here we have something called StaticObject implementing GameObject. This stimulates the question of which functions StaticObject actually implements. The answer is, in fact - only draw(), and by doing so one could conclude that StaticObject provides **a *way* of rendering**. It could be asserted that one could have embedded this procedure in GameObject instead of StaticObject. Although not illustrated in Fig. 10(c), there may almost certainly be some kind of object which does not use the standard draw() procedure such as the one StaticObject provides.

While a basic version of the draw() function of StaticObject was being developed

[5]What a shader switch is and how this affects the OpenGL context is delineated in [10].

```
class TankObject {
public:
    ...
private:
    ...
    Tank tank;
};
void TankObject::transform(int i) {
glTranslatef(tank->getXPos(),
             tank->getYPos(),
             tank->getZPos());
pair<Vec3, float> meshRot
             = tank->getRotation(i);
glRotatef(meshRot.second,
          meshRot.first.x,
          meshRot.first.y,
          meshRot.first.z);
}
```

Figure 12: *The transform() function in TankObject. The tank->getRotation(*i*) method returns the per-mesh specific rotation (depending on* i*) in form of a pair {axis being rotated, angle in degrees}. This pair is calculated by the tank's quaternions. An example is if* i = *0 then getRotation(*i*) might return the pair representing the rotation of the turret.*

early on, simultaneous development involving bump mapping and HDR was being experimented with on the TankObject. To be able to do this, TankObject initially extended GameObject directly - to relax constraints put upon by StaticObject. Thereafter, when the TankObject draw() function was perfected, it served as a replacement for the draw() function of StaticObject. As such, all objects which extended StaticObject could gain from newly created features such as bump mapping.

Finally, fig. 10(c) illustrates an example of some classes extending StaticObject. By doing so, they commit to using the same draw() procedure provided by StaticObject. In order to be instantiated, these three objects also provide a way of transforming an object. How this is done by the TankObject in fig. 10(c) is of most interest because of it's complexity. The C++ code in fig. 12 demonstrates the functionality of the transform() function.

The Tank member in TankObject supplies getter functions for the transformation of a tank in the game. These getters return the necessary quaternions[6], matrices and vectors which TankObject uses to supply an implementation of transform().

Thereby, these three subclasses to StaticObject are in a sense listening or *observing*[7] an object in order to decide on what transformation is to be applied on the 3D-model and its mesh.

4.3.3 Scene-graph

A scene-graph on the other hand, requires fundamental rendering directives to be known in advance in order to decrease the amount of context switches in the **OpenGL context**[8]. These rendering directives may consist of **context properties** e.g. shader switches, transformation matrices, bound textures and coloring.

These properties change the state of the OpenGL context before polygonal primitives are rendered. Large scenes often require large amounts of geometry to be rendered, much of which if rendered indepen-

[6]Quaternions form a 4-dimensional normed division algebra over real numbers. Its uses in computer game development is elaborated in Pipho *et al.* [9, "Introduction to Quaternions", Chapter 2],

[7]The Observer pattern, outlined in [11].

[8]OpenGL context is well documented in [5].

dently may store a great deal of redundant context properties.

Scene graphs can not only be used when rendering, they can also be used in frustum- and occlusion culling[12]. Occlusion culling can be done in the sense of the painters algorithm[8], where one *paints* every object in the order of which it's draw calls are made. Then by calculating the projected size of the AABB[9], one can examine whether an object is occluded or not.

Definitions Intuitively, one would like to group the use of redundant properties and recurring state changes. This can be dealt with in many ways, but usually redundant properties are identified and used as 'switches' when creating a state hierarchy. This state hierarchy is usually implemented with a **tree data structure**, with these switches as balancing agents. This kind of tree will hereby simply be known as a **scene-graph**. The following categories of node classes are usually identified[7]:

Shape Nodes Represents geometric objects, such as cubes, spheres and meshes. These so called shapes are basically draw calls to OpenGL. Note that a 3D-model and its meshes will usually be split into many shapes and have model-space[10] transformations extracted and created as property nodes.

[9]Axis-aligned bounding box

[10]Every 3D-model may have a world-space transformation and many local 'model-space' offsets locally. Therefore the coordinate of one mesh is the 3D-model's world-space transformation matrix multiplied with the mesh's model-space matrix. See [5].

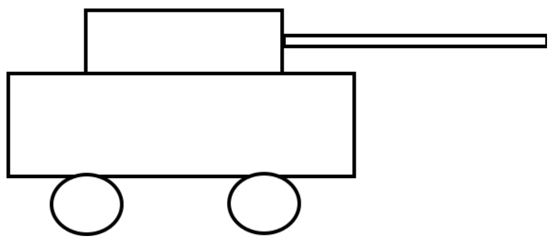
Group Nodes These may have any amount of children attached to it, and are used to collect nodes into hierarchies. When grouping different Property Nodes for instance, a group node may be used as a **separator** when traversing the graph. This separator stores the state which the Property Nodes together define and provides a way of traversing the graph with as few state changes as possible.

Property Nodes Attributes of a Shape Node, such as shader used, texture, transformation. Most importantly a property node consists of information which might change the OpenGL context.

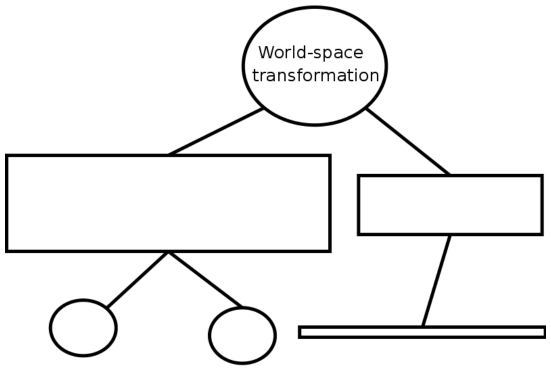
By classifying of nodes in the above manner, one can use the **Composite Design Pattern**[11] in order to create a hierarchical representation of group nodes and leaf nodes. Take notice that these categories are not restrictive. There could be a class of node that incorporates shape, properties and group characteristics[7].

In fact, when creating the scene-graph, nodes are created according to their classification. Much of this exhaustive work can usually be done at initial runtime. This also involved the creation of the nodes themselves and balancing of the scene-graph tree data structure. Other property nodes which are usually static - such as type of shader and texture, may also be created at this time.

Static Property Nodes such as those switching by transformation can also be created at runtime. This is done usually from sources such as a 3D-model. Since a 3D-model may consist of many meshes, but



(a) A Tank consisting of five meshes is transformed and rendered. These meshes are: the base, the turret, the gun and two wheels.



(b) A scene-graph depicting **transformation property nodes**. The turret and pipe derive from the same transformation, only the pitch differs. The wheels and the base of the tank share the same yaw and roll but not pitch.

Figure 13: *illustrates in a scene-graph created from a 3D-model.*

have the same world-space transformation, the world-space transformation could be used as root. Fig. 13 illustrates how **transformation Property Nodes** can be extracted from a model and be organized in a scene-graph. The children Property Nodes of this root will naturally be created from the individual meshes' model-space transformation offset. In this way, properties may be extracted from a 3D-model to create local scene-graphs, which will later be inserted into the **root scene-graph**. Other properties such as which shader or texture a 3D-model would like to be rendered with can also be extracted, and organized in this way.

Moreover, by defining *which* properties a **Shape Node** may be characterized by, one can **group recurring state changes with the help of Group Nodes**. However, before doing this, certain inter-object relations which group nodes in the scene-graph must be understood by the programmer[7]. These inter-object relations must be defined in beforehand in order for the entire scene to make use of the scene-graph as much as possible. Adding new properties to a scene-graph involves taking all inter-object relations into account when structuring the order of which the scene-graph is to be traversed.

Traversal When traversing the scene-graph, the graph must first be sorted and structured. The way in which a scene-graph is structured varies dependent on *which* and *how* many different properties there are. These properties are assessed in how they dictate rendering. Let's say that the scene graph consist of these different types of

Property Nodes:

- Shader Property
- Transformation Property
- Texture Property
- Color Property

In this case, one should consider the *cost* for doing each state change which these properties convey. A shader switch might be considered more expensive than binding a texture for instance, and would therefore be higher up in the hierarchy of the scene-graph.

Certain properties can also be nested and should therefore be considered in a different way. Transformations for instance, which are pushed on the OpenGL matrix stack; are not very expensive to do but can create many recurrent state changes. Since transformations are pushed and popped off and on the matrix stack, one should consider inter-object transformation dependencies when balancing the scene-graph.

Properties such as Color or Texture switches are on the contrary not nestable. The cost of these state switches in the OpenGL context are however not significant compared to switching a shader for instance. This implies that the scene-graph constructed from the above relations should consider having shader switches high up in the scene-graph. An example of how a few Shape Nodes' properties can be modeled is illustrated in fig. 14.

Note, this does not demonstrate a scene-graph, it just depicts the relationships between certain objects. The so called *separators* are meant to illustrate where a OpenGL context change should be made. To create Group Nodes from separators, first collect all Property Nodes which are extended. Record these properties in Group Nodes, then add extended Group and Shape Nodes as children. Moreover, when later rendering and traversing the Group Nodes, set their respective properties and loop over all it's children.

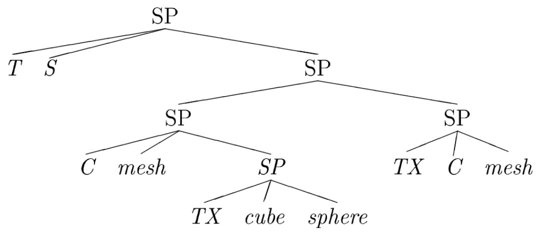


Figure 14: *Grouping of properties with separators. These separators create Group Nodes and comprise Property, Separator and Shape children. legend: SP = separator, TX = texture, T = transformation, S = shader, C = color.*

Keep in mind that separators are not the same as Group Nodes. This is because separators are used to model the current **inter-object relations** in the scene. With this information, and the cost for each Property Node defined; a grouping in form of Group Nodes can be organized in order to minimize cost. Note that the graph depicted in fig. 14 is not actually used by the scene-graph creation algorithm, it is just a way of illustrating from what information one would create it.

When the scene-graph has been built up, traversing can be done according to the pseudo code in fig. 15.

```
void Traverse(GroupNode n) {
for all property nodes p in n {
    push property modified by p;
    p.setProperty();
}
for all shape nodes s in n
    s.render();
for all group nodes g in n
    Traverse(g);
for all property nodes p in n
    pop all properties modified by p;
}
```

Figure 15: *Pseudo code for the traversal of a scene-graph.*

The steps of the algorithm in fig. 15, are elaborated as:

- First of all, loop over all of the Property Nodes in the Group Node. Note that all properties which change the OpenGL context are also pushed in beforehand.
- Then, all objects which do not require any more state changes are drawn with the help of the second loop.
- Next, traverse over all submodes, these nodes are affected by the Property Nodes in the first loop
- Finally, pop all properties which were pushed in the first loop. Pushing and Popping might in this sense be et al. glPushAttrib or glPushMatrix. For more information about what state stacks can be pushed and popped in OpenGL, see [5, "OpenGL State Variables", Appendix B].

4.4 Comparison

Just over a decade ago, games were developed by a handful of people. Game design was still a new notion in the industry and not much practice was standardized. This resulted in that companies such as **ID Software** to a great degree created their own design schemes. The design outlined were to a great degree dependent on the hardware which was used.

Doom for instance, has a very clever rasterizer but not much scene complexity to begin with [16]. This is due to the hardware requirements at the time. Later, with the wave of ID Software's *Quake*, general-purpose retained-mode graphics packages such as Criterion Software's Renderware were introduced. Scene management and culling became widely used as hardware allowed for more complex scenes. This gave way to more organized scene management, much in the form of scene-graphs. Scene-graphs and design however, are generally widely abstract notions and most developers choose to implement their own.

The traditional sense of developing games changed as developing firms began to buy third-party developed game engines licenses and middleware. A concrete success story of this is Grand Theft Auto 3, which uses the Renderware engine. GTA3 not only demonstrated great use of the Renderware engine, but it also serves as an argument that game play has become more important than the game engine itself[17].

The Unreal Engine, developed by Epic Games, delivers in their latest version 3.0, everything developing a game requires. Many developers have chosen to craft their own engine, using the third-party developed game engine as a base. One notable success of this, is Gears of Wars which uses the Unreal Engine 3.0, which was released in 2006.

The current trend also leans towards developers buying third-party middleware along with third-party developed game engines. One example of this is speed-tree[11] which is used by the 2006 released game S.T.A.L.K.E.R.. Middleware also exists in other areas of game development, providing network code, sounds and more. Especially massive-multiplayer games which are far more complex than ordinary single-player games benefit from middleware - most importantly in the area of network code[18].

4.5 Results and Conclusions

Two main branches of techniques have been evaluated in this section. One is that of a scene-graph and the other one includes initiatives taken by the developer team. A scene-graph is a powerful game design alternative to that of the one presented in TA, though it also restricts and delimits.

This however, is only true when developing a small game such as TA. When designing a project of larger proportions, a scene-graph reveals a notable increase in performance which can be vital when creating a real-time game[13].

4.6 Discussion

Due to lack of time, the decision was made to create a framework as fast as possible in order to immediately create contents for the game. Moreover, development was done independently on most components, and creating a scene-graph would have been too time consuming.

Had a change to scene-graph instead been made, it would likely not have contributed much to the performance in speed of TA. Therefore, it is hard to justify why we should have considered it. When the game was finished however, we noticed that components were increasing rapidly in size and therefore if we would have expanded the game further; a scene-graph would have been a must.

It is often satisfactory to rely on common design patterns when designing a simple game. However, when more and more features clutter a scene, it is useful to apply a scene-graph, which provides an increased level of organized structure and not just performance.

For TA, as noted above, performance would probably not have increased, as the Object System does already deal with minimizing shader switches. Moreover, there are no nested transformations in the scene, except those of the 3D-models. These are in The Object System dealt with in the same way as in a scene-graph. Each 3D-model has its own world-space transformation which is pushed on the stack. Then all meshes are iterated over and their per-mesh specific transformation is applied.

[11]This is a middleware which supplies a means of rendering trees and vegetation. Developed by Interactive Data Visualization, Inc.

5 Game Dynamics and Implementation

5.1 Background

One cornerstone of creating a game lies in the game dynamics. Decorative static objects need to interact with the terrain in a realistic way. The physics simulation of objects in the game need to be believable. The tank needs to be able to move on the terrain smoothly and correctly.

These issues and others need to be dealt with in order to tie all the components of the game together. Collision detection in essence does not know when or how it is used. How shots and tanks move in the game also have to be outlined and implemented.

In computer graphics, **Game Dynamics** encompasses different ways of describing how rigid bodies interact with each other. Game Dynamics has been center of much research in recent years and this has led to a new field of middleware.

Most notably, commercial middleware such as Physx[20] and Havok[21] have been included in the development process. Open-source middleware has also been considered by many developers and is growing in popularity. The 2007 hit game S.T.A.L.K.E.R[12] uses a range of open source alternative middleware such as ODE[13] and OpenAL[14]. ODE is a powerful alternative to Physx or Havok but has not yet reached equivalent popularity. ODE as an alternative will be examined in this section, but was not implemented in TA.

Other techniques, such as the camera system will also be elaborated on in this section. Camera systems in general are usually implemented in unique ways, depending on the game. Due to this, there is not much documentation about how one generally would implement a third-person camera.

[12]A FPS game developed by GSC World Games.

[13]Open Dynamics Engine, an open source game dynamics system. Website: http://www.ode.org/

[14]An open source audio library.

5.2 Techniques

5.2.1 The Player Tank

The player tank is arguably the most important entity in the game. It's what the player perceives and interacts with and because of this, it is important to convey a realistic experience.

Inconsistencies may disinterest and annoy the player so a good interface is needed. The game listens to the player's keyboard and mouse input and responds accordingly. For instance, when the player presses *forward*, the tank should move as is expected by the player. The orientation (rotation and translation) of the tank should alter depending on the surface of Terrain.

In the implementation of TA, the player tank is not tightly coupled with the terrain but receives input from it every frame. The player tank does not rely on very complex information from the terrain and requires the following:

***y* position** Given x and z coordinates; a y position on the terrain is needed. This is required in order to translate the tank in Euclidean space, reflecting the given two dimensional coordinate.

Normal Vector Given x and y coordinates; a normal vector for the face[15] on the terrain is requested. This is needed to create the quaternions which define the orientation of the tank.

A mathematical background in understanding quaternions[25] is somewhat assumed in this section. In computer science, quaternions serve as an excellent choice for representing rotations of an object. They are also often used when designing virtual cameras[22], as will be explained in section 5.2.2. See [29, Section 3.3] for a in-depth description of mathematical operators in linear algebra with quaternions. This section deals mostly with quaternion multiplication and SLERP[16]; so an understanding of these along with how quaternions represent rotations might be sufficient. The members of a quaternion q will be defined as

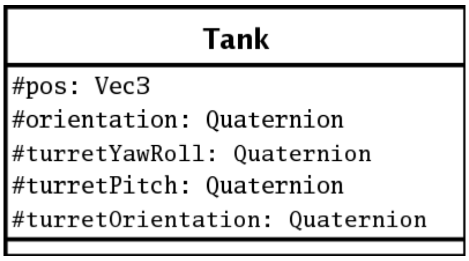
$$q = \{q_x, q_y, q_z, q_w\} \tag{1}$$

In TA, there are two classes of tanks, **PlayerTank** and **Tank**. How these are related in conjunction with TankObject is illustrated in fig. 17(a). The Tank class is a slimmed class which only supplies necessary quaternions and a translation vector. This base class which is used by TankObject on the client side, is described in 4.3.2.

The PlayerTank on the other hand, supplies methods and extra information to represent the tank with which the user interacts.

[15]In geometry, a face of a polyhedron is any of the polygons that make up its boundaries. For example, any of the squares that bound a cube is a face of the cube.

[16]Spherical Linear Interpolation



```
Tank
#pos: Vec3
#orientation: Quaternion
#turretYawRoll: Quaternion
#turretPitch: Quaternion
#turretOrientation: Quaternion
```

Figure 16: *The Tank base class.*

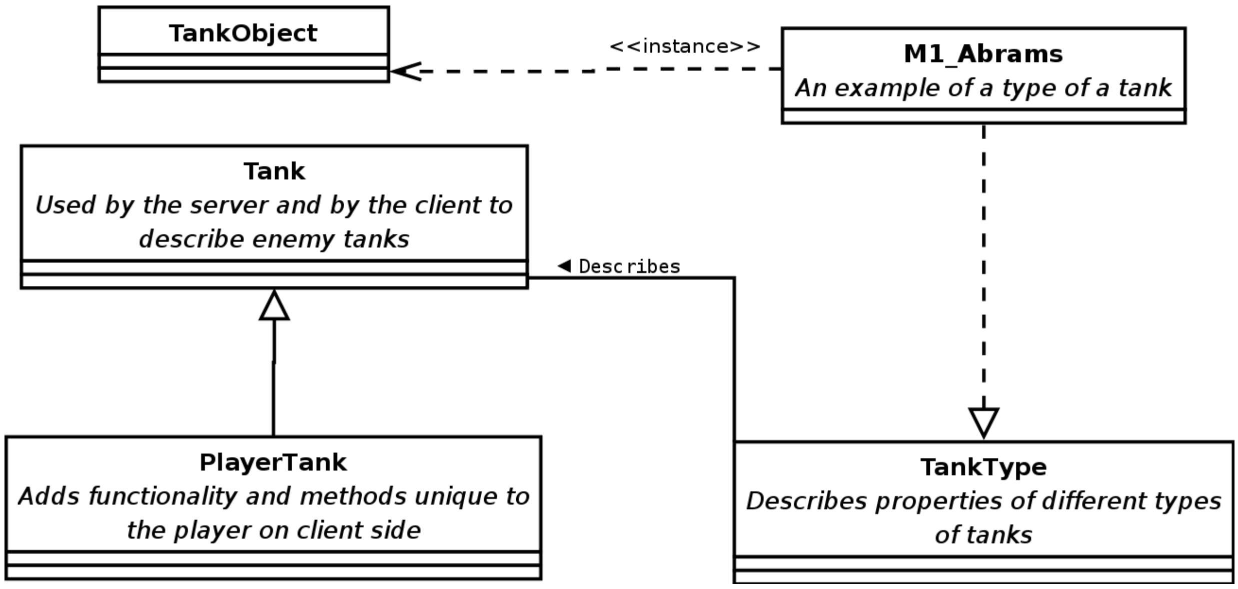
These methods handle how the quaternions and the translational vector of the tank are modified as the user moves around on the terrain.

Refer to fig. 16 for the different quaternions and the translation vector which Tank comprises of. The vector **pos** is the position which the tank has in Euclidean space. The quaternions on the other hand, might be a bit less intuitive to understand.

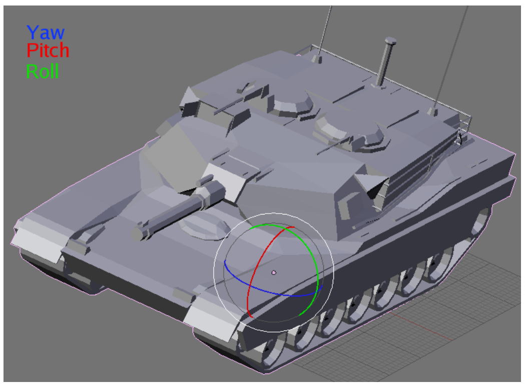
The **orientation** quaternion in Tank describes the orientation of the *base* of the tank. The *base* of the tank refers to the the tracks and engine compartment of the 3D-model. A rendered image of the default 3D-model used in TA is depicted in fig. 17(b) and fig. 17(c). The **turretYawRoll** holds the *yaw* and *roll* rotations of the tank.

The turretOrientation holds the orientation of the gun. And lastly, the **turretPitch** quaternion holds the *pitch* of the turret.

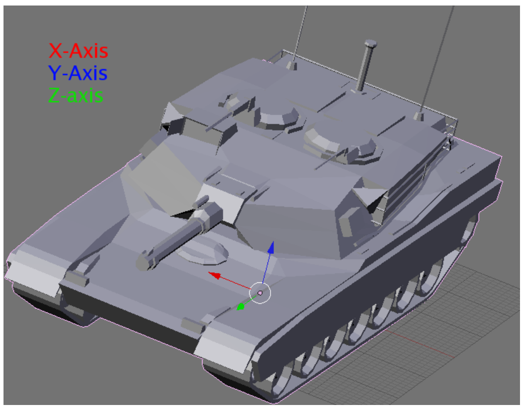
Yaw, pitch and roll are Euler angles and for a basic introduction to them, see e.g. Akenine-Moller and Haines[29, Section 3.2.1]. To see how these Euler angles correspond to the rendering of the tank, see fig. 17(b). To understand the correlation between the Euler angles and the Euclidean components of the tank, see fig. 17(c).


TankObject
<<instance>>
M1_Abrams
An example of a type of a tank
Tank
Used by the server and by the client to describe enemy tanks
◀ Describes
PlayerTank
Adds functionality and methods unique to the player on client side
TankType
Describes properties of different types of tanks

(a) The different types of tanks in relation to TankObject. The TankObject listens to a Tank for its transformation. The PlayerTank on the other hand, is only used on the client side. All the enemy tanks in the game which are rendered, are of the class Tank. The TankType class, is the same as depicted in fig. 9, section 4.3.1. Since TankType holds information necessary to create a TankObject, it may serve as a model for doing so. This is what the top right class depicts, which shows an instant of the type M1_abrams.



(b) The three Euler angles of the tank. Note that the pitch correlates with the arc formed by the z- and y-component in fig. 17(c).



(c) A tank in Euclidean space. The z-component points towards what could be perceived as being the forward direction of the tank. The y-component naturally points upwards and the x-component is the cross product of the two.

Figure 17: *The relationships between a PlayerTank, TankObject and Tank.*

Moving on the Terrain When moving on the terrain, the *base* of the tank should be rotated according to the normals of the terrain. A part from the normal of the terrain, the *yaw* must also be calculated. This *yaw*, is based on the input which the user supplies when he rotates the *base* of the tank with the keyboard. Figure 18 demonstrates how the tank's **orientation** quaternion is recalculated each frame.

Note that the degrees y in fig. 18 is the difference in rotation in which the orientation should be changed. The *yaw* quaternion from this information is later pre-multiplied with the old **orientation**. *Yaw* is pre-multiplied in order to rotate around the **orientation** quaternion's local y-component. This follows from the non-commutative property of quaternions when multiplied.

```
rotateY(y)
yaw = quaternion rotated
      y degrees around
      (0,1,0);
orientation = yaw *
              orientation;
```

Figure 18: *Pseudo code demonstrating a procedure which modifies the orientation quaternion of the PlayerTank.* **Y** *denotes the yaw angle in degrees from the user.*

In more understandable terms, this means that the former rotation of **orientation** is first applied, then the tank is yet rotated around the resulting y-axis in order to create the new **orientation**. The difference in mere visual appearance between the old **orientation** and the new, is that the new tank is rotated around it's y-axis; regardless of positioning on the terrain.

Now the pseudo code in fig. 18 does only demonstrate how a rotation around the tanks y-axis can be done. Let's elaborate on how the tank's quaternion is created from the terrains normal. Let's denote this normal as $\overrightarrow{n}$. First of all, we need the y-axis vector component of the tank's **orientation**. Let's denote this as $\overrightarrow{m}$.

To understand the definition of the conversion from a quaternion to a rotation matrix, refer to equation (3) in fig. 19. As depicted, $\overrightarrow{m}$ is simply the resulting y-component in the rotation matrix. This vector can be extracted from the **orientation** quaternion which for ease will hereon be denoted as simply q. Following this notation, we have

$\overrightarrow{m}_x = 2.0(q_x q_y - q_z q_w)$
$\overrightarrow{m}_y = 1.0 - 2.0(q_x q_x + q_z q_z)$
$\overrightarrow{m}_z = 2.0(q_z q_y + q_x q_w)$

Now we have the normal $\overrightarrow{n}$ from the terrain and the vector $\overrightarrow{m}$ which is the y-component in the matrix of q. The next step is to create a quaternion which represents the rotation from $\overrightarrow{m}$ towards $\overrightarrow{n}$. One could do this statically with no interpolation but since the normals over each frame are not so fine grained, the rotation is SLERPed.

SLERP stands for spherical linear interpolation and is often used when doing interpolation from one quaternion to another[19]. In our case we would like to rotate from q to the quaternion which represents the rotation from the vector $\overrightarrow{m}$ to $\overrightarrow{n}$. Let's denote this rotation as q_2, then we would have

$$\phi = \arccos(\overrightarrow{m} \cdot \overrightarrow{n}) \qquad (6)$$

$$\widehat{R_x}(q) = \begin{pmatrix} 1.0 - 2.0(q_y q_y + q_z q_z) \\ 2.0(q_x q_y + q_z q_w) \\ 2.0(q_x q_z - q_y q_w) \end{pmatrix} \quad (2)$$

$$\widehat{R_y}(q) = \begin{pmatrix} 2.0(q_x q_y - q_z q_w) \\ 1.0 - 2.0(q_x q_x + q_z q_z) \\ 2.0(q_z q_y + q_x q_w) \end{pmatrix} \quad (3)$$

$$\widehat{R_z}(q) = \begin{pmatrix} 2.0(q_x q_z + q_y q_w) \\ 2.0(q_y q_z - q_x q_w) \\ 1.0 - 2.0(q_x q_x + q_y q_y) \end{pmatrix} \quad (4)$$

$$R(q) = \begin{pmatrix} \widehat{R_x} & \widehat{R_y} & \widehat{R_z} \end{pmatrix} \quad (5)$$

Figure 19: *Conversion from the quaternion q to the rotation matrix R[22, "Page 248"]. R is a right-handed transformation matrix consisting of the column vectors $\widehat{R_x}$,$\widehat{R_y}$, and $\widehat{R_z}$. Note that the resulting matrix R is a* 3×3 *rotation matrix..*

$$\overrightarrow{\alpha} = \overrightarrow{n} \times \overrightarrow{m} \quad (7)$$

With the axis $\overrightarrow{\alpha}$ in (7) and the angle ϕ in (6), the q_2 quaternion can be created. *Q2* is simply the quaternion representing a rotation ϕ degrees around $\overrightarrow{\alpha}$.

With q and q_2 defined, SLERP can be done. Figure 20 depicts a mathematical definition of a spherical linear interpolation from q to q_2 using SLERP. The resulting unique quaternion from the $slerp(q, q_2, t)$ (8) function will be a linear interpolation from q to q_2; where $t = 0$ would result in no interpolation (q) and $t = 1$ in full(q_2). This constitutes the shortest arc on a four-dimensional unit sphere from q to q_2 as ex-

$$slerp(q, q_2, t) = \frac{\sin(\phi(1-t))}{\sin\phi} q + \frac{\sin(\phi t)}{\sin(\phi)} q_2 \quad (8)$$
$$t \in [0, 1],\ \phi = \arccos(q + q_2)$$

Figure 20: *The Spherical Linear Interpolation function of quaternions[26].*

plained in Akenine-Moller and Haines[29, Section 3.2.2].

This is hard to visualize, but think of it is as an arc from one point to another on a sphere. This arc spans from one point q to q_2 where $t \in [0, 1]$ defines how far along this arc from q to q_2 a point is traveling. If $t = 0.5$ for instance, the point would have traveled 50% along the way. Moreover, $slerp(q, q_2, t)$ would have returned the quaternion representing the rotation of q animated half the way towards q_2.

With the orientation of the *base* of the tank outlined, the orientation of the *turret* and *gun* remains. These orientations are represented in form of quaternions as **turretYawRoll**, **turretPitch** and their product turretOrientation; as depicted in fig. 16.

The *turret* is simply the part of the tank which is stacked on top of the *base*. The *turret* also comprises the *gun* but let's focus on it's orientation disregarding the *gun* for now.

The *turret* is naturally rotated independently in *yaw*, but *roll* and *pitch* are the same as that of the *base*. This implies that the y-component of the *turret*'s quaternion in matrix-form is equivalent to that of the *base*. Moreover, because the x and z-component of the quaternion in matrix-

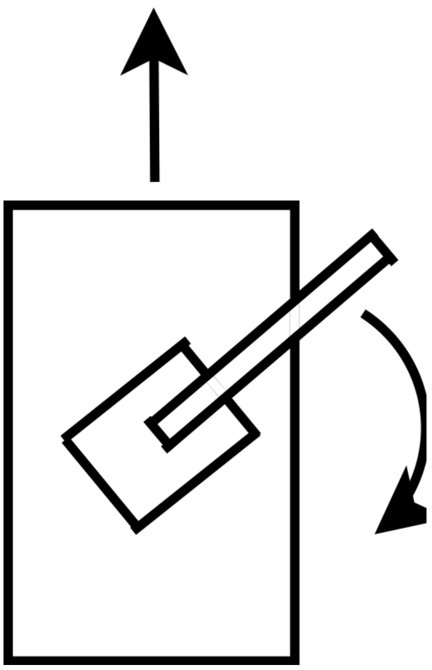


Figure 21: *Top view of the tank and its turret. The turret shares the same y-component as the tank. Due to that the tank does not share the same x- and z-components, rotation is possible as depicted.*

form is different of that of the *base* - the *yaw* will be different. Therefore the orientation of the *base* and the *turret* is essentially the same, with the exception of the *yaw* being different.

Examining fig. 21 may help in understanding this, which depicts the turret's independent rotation in *yaw* contra that of the tank. Moreover, the *yaw* of the turret is calculated in the same way as the with the *base*, as outlined in fig. 18.

The *gun* however, is a different matter. The *gun* conforms to the same orientation as the *turret*, but adds *pitch*. This so that the player is able to raise and lower the *gun* of the tank. A myopic implementation might be to store a quaternion in the Tank class which would represent the *gun* and the *turret* together. However, the *turret* alone, needs no pitch. Therefore, in order to rotate the *turret* correctly, the *pitch* needs to be removed. This brings forth difficulties, which can not be solved in a sensible manner. Therefore, in TA, the *pitch* is stored in a separate quaternion.

When the user moves the mouse, in hopes of rotating the *gun* upwards and downwards; the **turretPitch** quaternion is modified. This works in similar ways as in fig. 18 except that the variable *yaw* would be inter switched by *pitch*. In modifying the pseudo code, this would involve setting the variable $yaw = (0, 0, 1)$. By modifying the variable *yaw* in this way, *pitch* would be invoked instead. Now **turretPitch** alone does not constitute the orientation of the gun. This resulting product does:

```
turretOrientation = turretYawRoll
                    * turretPitch;
```

By post-multiplying **turretPitch** with **turretYawRoll**, the *yaw* and *roll* of the turret is applied with an additional rotation in *pitch* by **turretPitch**. This variable, called **turretOrientation** is memoized and stored in the Tank class because it is often requested.

Moreover, by substituting quaternion multiplication for rotation matrix multiplication, less computational operations are required[24, "Page 425"]. This is also an argument for use of quaternions contra matrices when storing rotational data.

5.2.2 The Camera System

Robust tank cameras are usually implemented with quaternions because of their interpolative abilities and their lack of gimbal locks[22].

In TA on the other hand, a different approach has been taken. Since the player's tank rotations are represented by quaternions and interpolated with the help of SLERP, there is no need to implement a camera based on such.

The camera instead receives input from the tank which it is observing. Every frame this input is received, and since the tank advocates smooth movement through quaternions - the camera benefits as well.

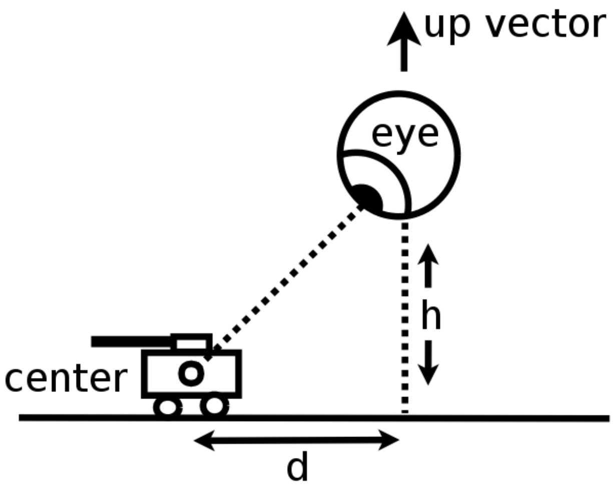


Figure 22: *A camera in two-dimensions.*

Camera Specification The camera works in close relation with gluLookAt() which is defined in Smith and Frazier [23, "Page 5"]. gluLookAt() takes three vectors; *eye, center, up.* These arguments are given to gluLookAt() for the construction of the view matrix. The following specifies the three:

- **Eye** Defines the origin of the eye in world-space, the point of reference.
- **Center** Defines the point at which the eye is looking at. Regardless of where the eye is, it will always point in the direction of the *center.*
- **Up** Defines the *roll* of the eye and specifies the direction of what the eye sees as 'up'. An analogy could be drawn to when you are up-side down; then your 'up' direction would point $\{0, -1, 0\}$, i.e. facing the earth.

Refer to fig. 22 for a two-dimensional description. The **center** defines the position of an eye looking at the **center** which is the center of the tank.

The Tank Camera Basically, the camera is listening on the **turretYawRoll** quaternion, which is noted in fig. 16. From this quaternion, the z-component of the derived matrix is extracted. The z-component, which will be denoted as simply $\overrightarrow{\gamma}$ is extracted from this matrix as in equation (4), fig. 19. That is
$\overrightarrow{\gamma}_x = 2.0 * (q_x * q_z + q_y * q_w)$
$\overrightarrow{\gamma}_y = 2.0 * (q_y * q_z - q_x * q_w)$
$\overrightarrow{\gamma}_z = 1.0 - 2.0 * (q_x * q_x + q_y * q_y)$
This in a way, represents the vector which the *gun* is aligned with. In fact, $\overrightarrow{\gamma}$ also represents the ray which a shot fired from the *gun* employs. This ray is depicted in fig. 23 and is intersected by the crosshair.

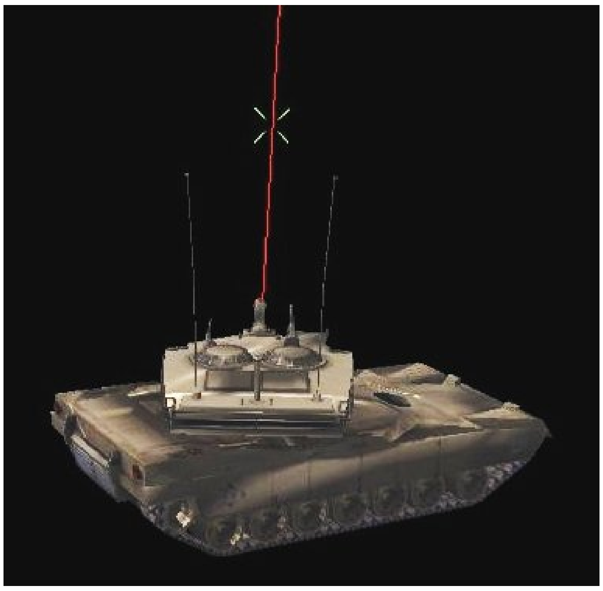


Figure 23: *A tank, crosshair and the ray of the z-component of the matrix from* **gunOrientation**.

In TA the **center** (using the above notation) is in fact the crosshair. This means that the user is always looking towards where he is aiming.

To acquire the point of the crosshair in world-space, one needs to translate first to the tank, then to the position of the ray. Then when on the ray, translate up until the intersection between the ray and the crosshair.

To simplify the mathematical notation, the following definitions are made

- $\overrightarrow{o_r}$ The model-space offset from the tank to the root of the ray.
- $\overrightarrow{o_t}$ The tank world-space position.
- l The length from $\overrightarrow{o_r}$ to the crosshair.

The final point in world-space which is used as the **center** is then:

$$\overrightarrow{center} = (\overrightarrow{o_t} + \overrightarrow{o_r}) + \overrightarrow{\gamma} \cdot l \tag{9}$$

With the **center** defined (9), there remain two vectors. The **up** vector in TA is always $(0, 1, 0)$ which results in the same kind of perspective as one has in real life.

This leaves the **eye** vector which is the reference of the viewer, as depicted in fig. 22. Since the tank camera has access to the PlayerTank, it can request its members. What's of most importance to the camera when computing the **eye**, is the **turretYawRoll** in fig. 16. With this quaternion, the direction of the tank can be calculated. This direction will be denoted as $\overrightarrow{\zeta}$. With the help of $\overrightarrow{\zeta}$, the camera can be positioned behind the tank. As above, this is the z-component of the matrix derived from the quaternion, which equation (4) in fig. 19 demonstrates.

Apart from the direction y, a distance is also needed in order to adjust where on the x-, z-, and y-axis the **eye** should be relative to the tank. These offsets are naturally flexible and interpolation is done whenever the camera **eye** is moved from one point to another.

Some threshold however, need to restrict the camera's **eye**. The **eye** should obviously never be beneath the terrain for instance. This kind of threshold restrict the camera from certain extreme positions but in order not to make the camera **eye** movement jerky, interpolation is always done. What defines the threshold of the camera is denoted by d and h in fig. 22. Refer to fig. 24 for the interpolation from the **eye** to it's new position. In equation (10) and (12), the x- and z- components of the **eye**

$$\overrightarrow{\Delta}_x = \overrightarrow{center}_x - \overrightarrow{\zeta}_x d \qquad (10)$$

$$\overrightarrow{\Delta}_y = \overrightarrow{center}_y + h \qquad (11)$$

$$\overrightarrow{\Delta}_z = \overrightarrow{center}_z - \overrightarrow{\zeta}_z d \qquad (12)$$

$$\overrightarrow{eye}_n = \overrightarrow{eye}_{n-1} + \frac{\overrightarrow{eye_{n-1}\Delta}}{|\overrightarrow{eye_{n-1}\Delta}|}\delta \qquad (13)$$

Figure 24: *Interpolation in world-space from the $\overrightarrow{eye}_{n-1}$ to $\overrightarrow{eye}_n$ (13). $\overrightarrow{eye}_n$ is $\overrightarrow{eye}_{n-1}$ plus the desired position $\overrightarrow{\Delta}$, which is scaled by d and h. The n variable denotes the frame in the game, and δ denotes the interpolation speed. In TA, δ is equivalent to the time it takes to render each frame times a constant.*

are rotated around the **center**. The radius which this circle of rotating creates is in fig. 22 denoted by d. The height of the **eye** (11) however, always strives to be h above the **center**. Also note that $\overrightarrow{\Delta}_y$ in equation (11) is unaffected by $\overrightarrow{\zeta}_y$. This is because **turretYawRoll** does not provide any *pitch*.

5.2.3 Open Dynamics Engine

An alternative to creating one's own game dynamics in a game, is using middleware. There are a lot of commercial middleware available which provides game dynamics to a game, but in this section, the open-source alternative Open Dynamics Engine (ODE) will be evaluated.

ODE emphasizes speed and stability over physical accuracy, which makes it an excellent choice of middleware for games[27]. ODE provides essential collision detection but allows the developer to provide it's own. One of the major features of ODE provides, is the simulation of what is called *articulated* rigid body[17] structures.

An *articulated* rigid body is one which is comprised by many joints and is simulated accordingly. Joints connect rigid bodies and build up a hierarchy. A human for instance, might have five joints which connect to the torso. These are the arms, legs and head. ODE provides a way of simulating this kind of *articulated* rigid body, in a realistically way as possible.

ODE also provides fast and robust collision detection. The most common collision primitives such as sphere, box, ray and triangular meshes can be collision queried. Collision response in the form of physics simulation is also provided.

5.3 Results and Conclusions

By altogether designing and implementing the game dynamics from scratch, an acceptable degree of realism has been achieved. The tank moves smoothly on the terrain and can rotate in all of the expected ways. The tank can on the other hand, not be airborne nor drive over objects. This derives from the fact that the tank is not represented as a rigid object. The tank is basically just represented as a vector in world-space. This is not enough information to conduct a physics simulation regarding how the tank should collide with the terrain and static decorative objects in the scene.

Static Decorative Objects A Static decorative object (hereon shortened to

[17]A rigid body is an idealization of a solid body of finite size in which deformation is neglected.

SDO) refers to an object which is placed in the scene and is only for decorative reasons. These objects could be e.g. trees, houses or rocks.

In TA, SDOs are created from 3D-models and placed arbitrary on the terrain, defined in the level information. Level information in TA is stored in a file which can dynamically be changed if desired. The level file consists of information such as the rotational properties of a SDO and its placement in world-space.

By supplying the player's tank and all SDOs with OBBs, primitive rules of interaction can be enforced. These restrict the tank as to not collide with any SDOs in the scene, i.e. by rejecting movement which would result in the tank clashing with any SDO. This does not imply that there is any collision response. On the contrary, there is none. In TA, this does not irritate the player unacceptably, though it is definitely something which is noticeable. If two tanks ram into each other for instance, they just stop abruptly on collision.

It should be noted that, if ODE would have been considered instead of the player tank technique in section 5.2.1, collision response could have been simulated.

The Camera System A robust camera system was also developed. This system provides satisfactory capabilities and is highly configurable. Since the camera moves according to the orientation of the tank, it can also be used to spectate other users. Due to this, if a player has died, he could be able to spectate others, even see where they are aiming.

The concepts explained in section 5.2.2 do not only apply to creating a camera which strictly listens to a tank. The camera can be modified to listen on any kind of vector. Most usefully one which serves as the character's view in third-person.

5.4 Discussion

As usual, the choices of implementation reflect the amount of time able to be invested. In this project, the team did not want to implement third-party middleware and by doing so have all problems solved. We wanted to design something hands-on, and learn the concepts of Game Dynamics by creating systems by ourselves.

If ODE or some other game dynamics SDK would have been implemented instead, the game would have most certainly been more realistic. By doing so though, we would not have had the chance to learn how to work with quaternions and the fundamental pillars of what actually build up the game dynamics. Our implementation of how the tank moves on the terrain is somewhat primitive, though we would hope the user find it surprisingly agile. Our goal was to reach this basic level of terrain movement, while learning and developing our own system foundation - without the help of third-party middleware

6 Collision Detection

6.1 Background

Collision detection is done in order to determine if two objects collide. Collision detection is an important aspect, as a significant component of the players' interaction in the game is determined by collision queries, and collision response. Performing collision detection is often not possible to do in real-time if certain acceleration algorithms aren't used.

Collision handling involves three separate stages when working with object interaction. These stages are **collision detection**, **collision determination** and **collision response**. Collision detection tells us *if* the objects collide. Collision determination tells us *how* the objects collide. Lastly the collision response determines how objects are affected by each other after a collision has been detected, such as a change of momentum. Most importantly, one should distinguish between collision detection and collision response, whereas the latter dictates the response which occurs after an actual collision has been detected.

In a fast-paced game such as TA, the amount of collision queries are especially frequent, and require rapid real-time response. The response must not only be fast, but also precise. The player often anticipates the consequence of a certain collision before it visually occurs. A noticeable inconsistency could alienate the players' sense of interactivity. Care must be taken when choosing an algorithm for fast collision detection. The choice of algorithm is dependent on the *kind* of objects that are going to be detected for collision, and the constraints we impose on them. The different algorithms and data structures which can be used when mitigating collision detection are often categorized with respect to the type of objects they can be applied to. These algorithms exploit different properties and constraints of the objects they are used on. Certain constraints are implicitly or explicitly imposed on the objects we are accelerating - when using such algorithms and data structures. What it really comes down to is what kind of compromises one is willing to make for higher performance.

Coarse pruning When performing collision detection between objects composed of many smaller primitives, it may be beneficial to calculate some enclosing primitive object. It might then be faster to discard a collision by testing this enclosing volume, also called **bounding volume**, instead of the many primitives. The objective is to quickly discard pairs of objects that does not intersect.

Temporal coherence When objects follow physically accurate motions, their relative change is small between the collision tests. Many objects does not move at all times.

Knowing this, many algorithms exploit the temporal coherence and reuse results from previous collision tests. Information related to how objects are allowed to change between collision tests can also be used to exclude objects as potential colliders, e.g. the maximum velocity for an object. Examples of such algorithms are [32] and [33].

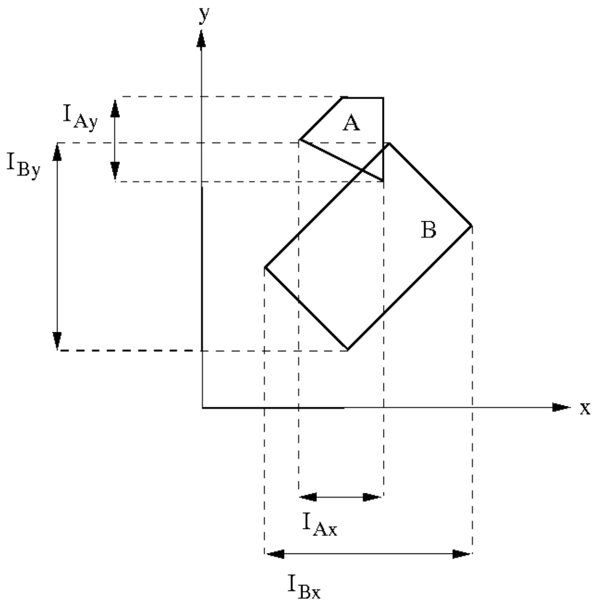
Exact collision detection When two object have been deemed potential colliders they must be examined further. Complex objects are often composed of smaller primitives e.g. triangles. The intuitive approach would be to test all the primitives of one of the objects against the primitives in the other primitive. This naive approach is however not practically viable when objects are composed of a large quantity of primitives. Fast algorithms are available using methods from linear programming for solving these problems. There are methods even more efficient when the simulated system is an iteratively simulated. A method that is used for precise collision detection for convex polyhedral objects is the **Voronoi-clip** algorithm also called **V-clip** [33]. The V-clip algorithm is a **feature-based** algorithm. The features of a polyhedron are the vertices, edges and faces of the object.

Pre-processing When the objects are known to be static, certain algorithms can be used to construct hierarchies which significantly improves the performance for some type of queries. Fired shots and polygonal meshes are essentially the different kinds of objects which can collide in TA. The polygonal meshes are never deformed and only translated and rotated. The only moving mesh in TA, which require collision detection is the tank mesh.

The fired shots are represented as line segments. Shot ray-mesh collisions, from fired shots, are frequently queried so accelerating these queries is important. Collision detection between polygonal meshes are also needed for collision between the player tank and the various decorative static objects in the scene. In TA however, mesh-mesh collisions are approximated with collision between the OBB's for the meshes. This choice was made because of lack of time. When considering algorithms for ray-mesh collision detection for TA, several widely used and well documented datastructures were examined.

6.2 Techniques

6.2.1 Sweep-and-prune



(a)

Figure 25: *This figure illustrates that all axis projections of objects that collide intersect (the intervals), must overlap.*

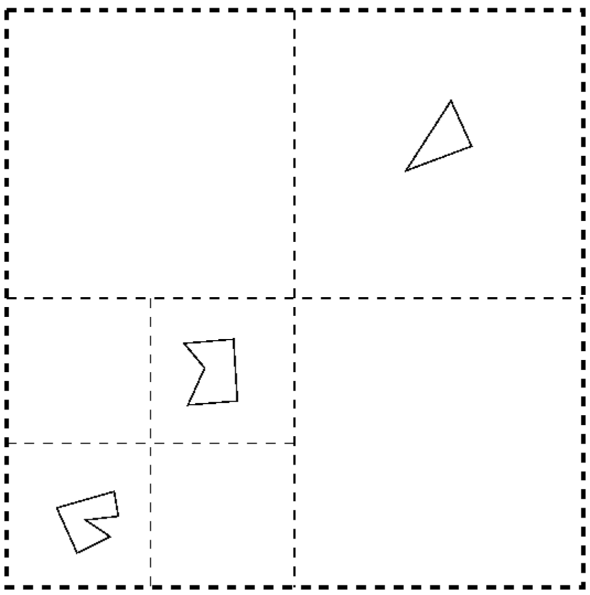
A method that uses both temporal and spatial coherence is the **sweep-and-prune** algorithm, explained in [42]. The sweep-and-prune algorithm is beneficial when there is a large number of independent objects that may be colliding and they only move slightly between the collision tests. This algorithm assumes that the objects that are going to be pruned only are rotated and translated.

A tight fitting "largest bounding box" is computed for each object. The idea is that the object fits inside this bounding box regardless of rotation. When testing for possible collision, the interval created by the largest bounding box for an object is projected to each axis. Each object will have an interval in this list for each axis. This list is then sorted. The criterion for intersection of two bounding boxes and thus, possible intersection, is that and two objects intervals intersect in all the axes. See figure 27. When two objects have been deemed possible colliders, an exact collision detection is performed.

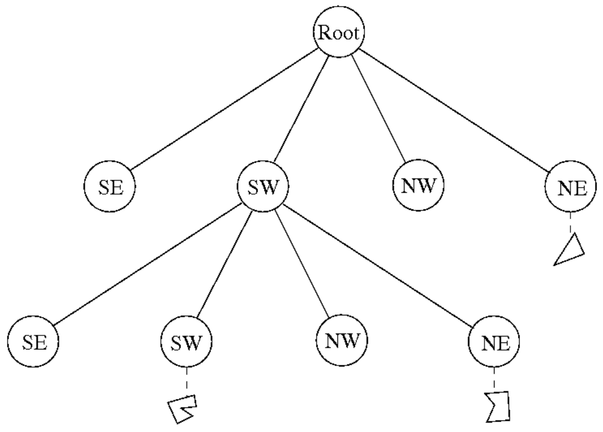
This method is fast because when while the sorting of the interval lists usually takes $O(n^2)$, the relative change of the intervals between collision test are small. This means that if the lists from the last test were to be used, it would be almost sorted already. If using a suitable sorting algorithm like bubble sort or insertion sort, the expected sorting complexity would be $O(n)$.

6.2.2 Quadtree

The quadtree is explained and examined in Finkel and Bentley *et al.* [30]. The quadtree spatial tree data structure is a data structure in which each internal node in the tree can have four children. Each node in the quadtree represents a **square**



(a) This illustrates how the nodes divide the space.



(b) Here we see the node hierarchy for the quadtree.

Figure 26: *This figure illustrates a quadtree.* NE, NW, SW *and* SE *denotes the quadrants of the parent.*

or **rectangular region** of the space. See figure 26. The children of a specific node correspond to the quadrants of the parent space.

As with other spatial data structures, many different kind of primitives can be contained in a quadtree. A simple way of constructing a quadtree for a set of points would be to find a square or rectangle that encloses all of the points, then recursively divide each node with more than one point contained.

Quadtrees can be used for collision detection between rays and terrain. This is possible because of the way the terrain mesh is constructed. No parts typically overlap vertically in a terrain constructed from a grid.

A quadtree may also be used to determine what **LOD** different parts of the terrain should be rendered with. Parts of the terrain close to the camera should then be rendered with high detail, while distant parts with lower detail. This can be done with quadtrees by constructing a quadtree of the terrain. When traversing the quadtree, a check is made for the distance between the camera and the node currently being traversed. If the node is farther away than a setting, it is not traversed further, and a low-quality version is instead used.

The depth of a square quadtree is related to the distance between points and is at most $\log(s/c) + 3/2$, where c is the smallest distance between any two points, and s is side length of the square. The cost of construction for a quadtree depends on the depth of the tree. See Langetepe and Zachmann *et al.* [31, "Quadtrees and Octrees", Chapter 1].

6.2.3 Octree

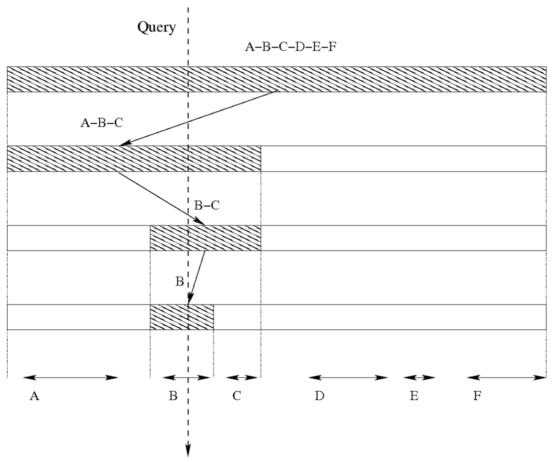
Octrees are the three-dimensional analog to quadtrees. Internal nodes of the octree have eight children commonly called **octants**. These child nodes partition the parent space in eight subspaces. Where along the point which this space subdivision is made categorizes different kind of octrees.

Octrees are like quadtrees generally best suited for rigid bodies where the geometry does not deform in any way. This because of the time it takes to construct the tree. Ray traversal in octrees can be difficult to implement, whereas this is a rather relatively simple task for kd-trees. The generation of octrees is however relatively a computationally cheap task.

Octrees where the subdivision point implicitly is located in the middle of the parent space are called **MX** octrees. The opposite equivalent of MX octrees are when the subdivision point can be arbitrary, and is something that is stored with each node. These trees are denoted **point region octrees** or **PR octrees**. With the exception of a few cases it has been shown that the number of nodes in an octree representation of an object, is proportional to the surface of the object [34].

6.2.4 *k*d-tree

The kd-tree is a type of **binary space partitioning** (**BSP**). A BSP-tree is a tree which recursively subdivides space by arbitrary planes. kd-tree is short for k-dimensional tree and is the special case of the regular BSP-tree. Instead of having arbitrary splitting planes as in a BSP-tree, it uses axis aligned hyperplanes to subdi-



(a)

Figure 27: *This figure illustrates searching in a segment tree.*

vide the space. kd-trees can also be seen as a k-dimensional generalization of the one-dimensional segment search tree explained in Langetepe and Zachmann *et al.* [31, “Segment Trees”, Chapter 2] or as show in figure 27.

The kd-tree accelerates ray tracing significantly because large quantities of potential colliding primitives can be excluded for each recursive step in the tree. The construction of a kd-tree can be very expensive computationally and is therefore suited for rigid bodies that are not deformable.

The cost of constructing a kd-tree depends largely dependent on the algorithm used for construction. The construction of the kd-tree is usually a part of the preprocessing of the rendering and not something that is performed every frame. Some construction schemes are so costly that the kd-tree structure is saved to disk to reduce loading time.

Construction of the *k*d-tree The efficiency of a kd-tree depends to a large degree on the algorithm used when constructing the tree. The algorithm should be chosen depending on which type of data the kd-tree contains and what type of queries that are going to dominate when using the tree. The outline of the construction code for a kd-tree is straightforward.

```
void construct_kdtree(Node & node)
{
    if(termination_criterion_met())
       return

    node.splitpos = find_split_pos()

    for each p in primitives
    {
        if(node.is_on_left_side(p))
            node.left.add_primitive(p)

        if(node.is_on_right_side(p))
            node.right.add_primitive(p)
    }

    construct_kdtree(node.left)
    construct_kdtree(node.right)
}
```

The termination criterion is a criterion that will stop the recursive construction of the kd-tree when some criterion is fulfilled. What differentiates the various construction algorithms is the termination criterion and the splitting rule that actually determine where the split position should be for maximum efficiency. For ray-tracing, a balanced tree is not generally desired.

Splitting rules

Standard split Standard split also known as spatial median splitting is described in [28]. Splitting dimension is chosen along with the axis with maximum spread. maximum spread is the difference between the maximum and minimum values. The splitting location is then determined to be the median. This is the most well known and most used splitting criteria.

Midpoint split The splitting plane is chosen to pass through the center of the cell and splits the longest side of the cube. If the root is a cube the resulting subdivision is similar to a binary version of the quadtree and octree.

Sliding midpoint split Sliding midpoint explained by Mount and Arya *et al* [39] is a variation of the midpoint-split. When doing a split and any of the sub node's are empty, the split is "slid" towards the other node until a data point is encountered and the size of the empty sub-node is maximized. Songrit Maneewongvatana and David M. Mount showed in [40] that the sliding midpoint split satisfy the packing constraint and thereby explaining the split-method's good performance. Their paper "It's okay to be skinny, if your friends are fat." showed that the sliding midpoint split sometimes produces skinny long cells, but that every such cell has a neighbor that is fat along the same direction. The result of this is that a generated tree cannot be composed only of skinny cells since the presence of fat cells then also must exists. The aspect of the skinny cells are still unbounded, but the *amount* of skinny cells is still limited.

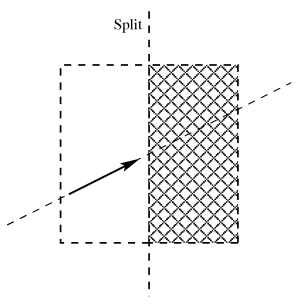
SAH (Surface Area Heuristic) split Described in [38] Idea is to maximize the empty space cells and to do that as close to the root node as possible, this results in an unbalanced tree. The SAH estimates the emphcost of traversing the split cell with regard to the resulting geometry of the split. In order to do so the heuristic makes certain assumptions about the rays that are used in the traversals, and uses these assumptions for the cost function of a split:

- Intersection rays are uniformly distributed.
- Intersection are infinite in length (they don't start or stop in a cell).
- Cost of the intersection test and traversal is known.
- The intersection cost of n intersection tests is directly linear to the cost of a single test.

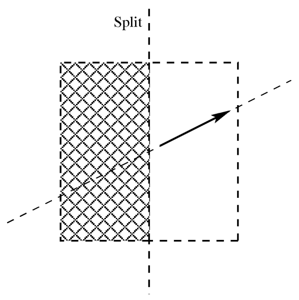
In addition to estimating the cost of a split, the SAH also determines when to terminate the subdivision.

Ray traversal The kd-tree in TA should be able to contain objects that can have rotations and translations performed on them. Hence, the kd-tree itself should be transformable. To do the intersection test, both the ray and the kd-tree should be in a common space. Assume that the ray that is going to be intersected with the kd-tree is given in world space.

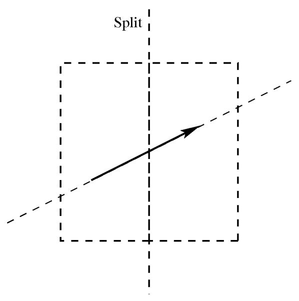
The kd-tree need to be axis-aligned by nature, so the ray is the only thing that can



(a) Ray segment starts on the left side and stops before the split plane. Traversing the left side and culling the right side.



(b) Ray segment starts on the right side and stops before the split plane. Traversing the right side and culling the left side.



(c) Ray segment intersects the split plane. Traversing both sides, beginning with the left side.

Figure 28:

be transformed. To transform the ray into the space of the kd-tree, the inverse of the kd-tree transformation is needed. The ray is transformed with the inverse of the kd-tree transformation and the resulting ray is used when doing the intersection test. Note that the inverse of a transformation consisting of only rotations and translations is trivial.

When traversing a cell, the half-spaces that the ray-segment intersects are checked, and the half-spaces are checked so that the spaces closest to ray origin are checked first. See figure 28.

Performing the kd-tree ray traversal fast is important because one typically perform many more traversal steps than triangle intersections [38, Table 7.5]. When traversing a kd-trees, the ratio of computation to the amount of accessed memory is low. The choice of how to store the kd-tree nodes is therefore key when optimizing traversal.

6.3 Results and Conclusions

TA games used a simple implementation of the kd-tree for shot-mesh intersection testing. The standard splitting rule was used when generating the kd-tree. The loading time for TA was approximately 15 seconds on the target machine and was extensively tested with extreme parameters when generating the tree. The inspiration for using kd-trees came from Jacco Bikker's ray-tracer capable of rendering at interactive rates. His ray-tracer is able to trace millions of rays each second on in static scenes consisting of millions of triangles, [36]. Note however that Jacco Bikker's ray-tracer is specifically optimized with packet tracing as explained in [35]. Packet tracing would not work well with non-coherent rays. Jacco Bikker briefly outlined how he implemented his ray-tracer in [37]. Another reason for choosing kd-tree's was that the ray traversal algorithm is shorter and easier than octrees [38, p. 95] The sample implementation provided is the one used for reference when writing the TA implementation. The sample implementation incorporate certain memory-alignment, and stack optimizations, for the kd-tree, which is briefly explained in [38].

Collisions between polyhedral objects in the game was approximated with bounding volumes. The OBB bounding volume was utilized for this purpose. The OBB-OBB intersection algorithm used is the one explained in [29, p. 602].

The primitives contained in the kd-tree's of TA was triangles. The ray-triangle intersection algorithm used in TA is the one described in [29, p. 573].

6.4 Discussion

The error that the approximated bounding volume created was far to great and noticeable for certain meshes. With more time, we probably would have used V-Clip [33] or GHJ [32]. While these collision schemes are far more complex and demanding to implement from scratch, ready and free implementations exists. Examples of such an implementations is the MERL V-Clip Collision Detection Library[18]. These implementations would presumably be quite easy to implement and would yield more realistic results.

We would also implement a faster and more intelligent split function for the kd-tree when doing ray intersections, presumably a variant of the SAH split rule. The variant would be implemented according to the method mentioned in Ingo Wald, and Vlastimil Havran [41]. They describe a method for creating a SAH kd-tree's with $O(n \log(n))$ complexity, the theoretical lower bound.

Some kind of spatial partitioning that can handle visibility queries, like the polygon aligned BSP-tree's used in Quake3, would be useful to reduce network traffic. The data-structure would be used for deducing which regions the players can see, and only sending game state updates that each player can see.

[18]http://www.merl.com/projects/vclip/

7 Network

7.1 Background

Players tend to have the same performance and consistency expectations of their online multiplayer games as they do of their single player games. Both network latency and a finite network bandwith makes this a big challange for multiplayer game developers. To be able to solve these problems a suitable network platform has to be developed.

7.2 Techniques

Client/Server Every player connects to the same server using a client. Current game state is computed on the server while the client only handles user input, which it sends to the server, and renders the current game state, received from the server. The server could be dedicated remote from any client or reside on the same machine as one of the clients. Because the server is in control of the game state a lot of cheating can be avoided as the server can discard all illegal requests. Clients dont depend on each other and can join or leave a game without restarting the game session.

Peer-to-peer In the peer-to-peer architecture, there is no central repository of the game state and no computer is more important than any other. Each client controls a part of the game state which it sends to every other client. This protocol's primary advantage is reduced network latency. In the client/server protocol each message travels to a potentially distant server and the resulting game state is sent to all the clients. In p2p each client sends directly to all other clients.

7.2.1 Latency and Bandwidth

The term bandwidth in computer networking refers to the data rate supported by a network connection or interface. One most commonly expresses bandwidth in terms of bytes per second. Because network bandwidth is limited, an action game server (such as TA's) can not send an update every time the game state change. Instead, the server takes snapshot of the game state at constant rate which it then sends to all clients. Bandwdth is not the only problem when communicating over a network, one other thing is delay in message delivery called *latency* or *ping*. Latency is the time between the client sending a user command, the server responding to it, and the client receiving the server's response.

It should be noted that if the client only renders the scene with the objects at the positions received from the server, moving objects and animations will appear choppy. There are several ways to prevent this:

Interpolation One solution to this problem is to go back in time when rendering the scene so animations and moving objects can be smoothly interpolated between two recently received snapshots. The amount of time to go back need to be at least as much as the time between server snapshots. Games based on the Source Engine goes back double that time [90] to prevent problems if a packet is lost or delayed. If more than one packet is lost or delayed, linear extrapolation is used to create an approximated snapshot.

Extrapolation Extrapolation is the process of constructing new data points outside a discrete set of known data points. This could be a very simple linear function (i.e. keep moving the object in its current direction), or more advanced, see [94] for an example where several different functions are used depending on the object and purpose.

Client Prediction When a player makes any action such as moves, the client sends a request to the server which updates the position of the player and eventually sends a new game state back to all clients with the next snapshot. This will make a delay between input and visual update. If the client predicts what the server will return with the next snapshot it can start to move the player instantly. Then, if the next snapshot does not match the predicted one, the client will have to correct (preferably using interpolation) its own position because the server has final authority.

Roll-Back Contrary to the above solutions this is a server side refinement and its purpose is not to make visualization smoother but to hide player latency. When the server receives a shoot request (this only concern instant hit weapons) from a client, it will roll back the game state until the time when the player pressed the shoot button on the client (i.e. half the client's latency). This reduces the otherwise significant penalty of incurring high latency, but can allow players feel like they are being "shot around a corner". This can happen if a player hides behind a corner (or anything else), and a player with such a high latency that when the server make the roll-back, the player that hide will appear to be hit even though he is behind protection. However, the player who shot will perceive it as a direct hit.
There are also several optimizations to keep bandwidth utilization as low as possible.

Variable Quantization Variable Quantization is a way to reduce network use by decreasing the number of bytes sent per variable. A float can be converted to an integer, and by doing so lose precision and reduce the bytes needed to represent its value.

For example, the position of a tank is represented by a vector using three floats, four bytes each. Often, it is not necessary to send all those 12 bytes - a player will not notice if the tank is positioned at 325, rather than 325.12. A two bytes integer could be used instead of the four byte float, thus cutting the network utilization in half.

This is a lossy coding, which implies that the lost precision can never be reproduced.

Lossless Data Compression Before sending a packet, a lossless compression algorithm can be applied on the data string in order to reduce the bytes which need to be sent. When the data string is received by the other end, it can be restored to the original string before being processed. One of the most common compression algorithm used in several games, (such as Tribes [91] and Quake 3 [93]) is the Huffman coding [98].

Delta Compression or **Delta Encoding** [19] is a way to reduce the data sent by

[19]The unix commando *diff* is a delta encoding software.

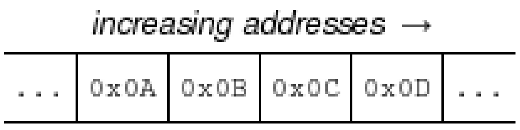
only sending the *differences* between the receivers current data, and the new data [96]. For example, if the client has a string “Hello World?” and the server wants to update it to “Hello World!”. Then, in this case, sending the entire string would be unnecessary. All that actually has to be sent, is the difference between the two strings, i.e. ’?’ to ’!’. This can also be applied to when sending the position and orientations of the tank.

As explained in section 5.2.1, the tank contains three quaternions and a translation vector. If none of the quaternions are changed between two server ticks, then we only need to send the translation vector. This would decrease the amount of floats sent, i.e. by four floats per quaternion × three quaternions. This would result in that only three floats would have been sent, which is what the translation vector requires.

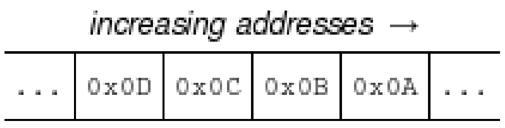
7.2.2 Endianness

Endianness is the byte ordering[20] in memory used to represent data (e.g. an integer).Big-endian and little-endian are the two most commonly used orders but not the only two available. The relationship between the two is depicted in fig. 29. With little-endian the bytes are ordered increasing numeric significance with increasing memory addresses, "little end first". Big-endian is the opposite order. i.e. decreasing numeric significance with increasing memory addresses. All x86 platforms use the little-endian format and Motorola, PowerPC and SPARC platforms use the big-endian format.

[20]Note that endianness can also refer to bit order.



(a) The big-endian byte ordering with 1-byte address increment.



(b) The little-endian byte ordering with 1-byte address increment.

Figure 29: *The difference between big-endian and little-endian.*

Communications between systems which uses different endianness could be a problem without some precautions taken. Networks generally use big-endian order and the **Internet Protocol** [97] defines a standard big-endian network byte order. There is a set of functions defined in the **Berkeley sockets[99] API** which convert 16- and 32-bit integers to and from a network byte order:

htonl() host-to-network-long, converts a 32-bit integer from host byte order to network byte order

htons() host-to-network-short, converts a 16-bit integer from host byte order to network byte order

ntohl() host-to-network-long, converts a 32-bit integer from network byte order to host byte order

ntohs() host-to-network-short, converts a 16-bit integer from network byte order to host byte order

7.2.3 Tribes Networking Model

Tribes [91] uses both reliable and unreliable packets. All data are classified into four different categories:

Non-guaranteed data Data which is never re-transmitted if lost.

Guaranteed data Data which must be retransmitted if lost, and delivered to the client in the order it was sent.

Most Recent State data is volatile data of which only the latest version is of interest.

Guaranteed Quickest data is data that needs to be delivered in the quickest possible manner.

The Tribes network model consist of two different layers. The first layer is the **Connection Layer** which actually delivers the UDP packets. The **Connection Layer** doesn't guarantee that these packets are delivered but it does provide packet delivery status notifications. These notifications can be used by the **Stream Layer** to achive a guaranteed packet delivery. The **Stream Layer** consist of three different managers which send data:

Ghost Stream Manager This manager is responible for creating a "ghost", i.e., a copy, of a local object on a remote host and keeping it up to date. An object's data is classified as Most Recent State data and if a packet for an object is dropped and the position of that object has changed since the packet was sent, the new position will be sent instead of sending the old packet again. The creation of a new copy on a remote host is classified as Guaranteed data and will be resent if lost.

Move Stream Manager Delivers client input moves as Guaranteed Quickest data to the server. This manager is also responsible for sending the clients' movement data from the server to all clients (actually, to all players who can see, hear or in any other way need it). This is done in two separate ways. First it send this move data with every packet to all clients, it also creates a control object which is transmitted through the Ghost Stream Manager. This control object contains data used to validate, and if necessary correct, the clients objects position.

Event Stream Manager The Event Stream manager is responsible for providing guaranteed and non-guaranteed delivery of all other data from one host to another. Guaranteed deliveries will also be processed in order.

All three manager uses separate sliding windows [95] to track packet delivery. When these windows are full, transmission of new data will be halted until acknowledgements arrive and the window can advance.

```
if (newState.sequence < lastState.sequence)
{
  //discard packet
}
else if (newState.sequence > lastState.sequence)
{
   lastState = deltaUncompress(lastState,newState);
   ackServer( lastState.sequence );
}
```

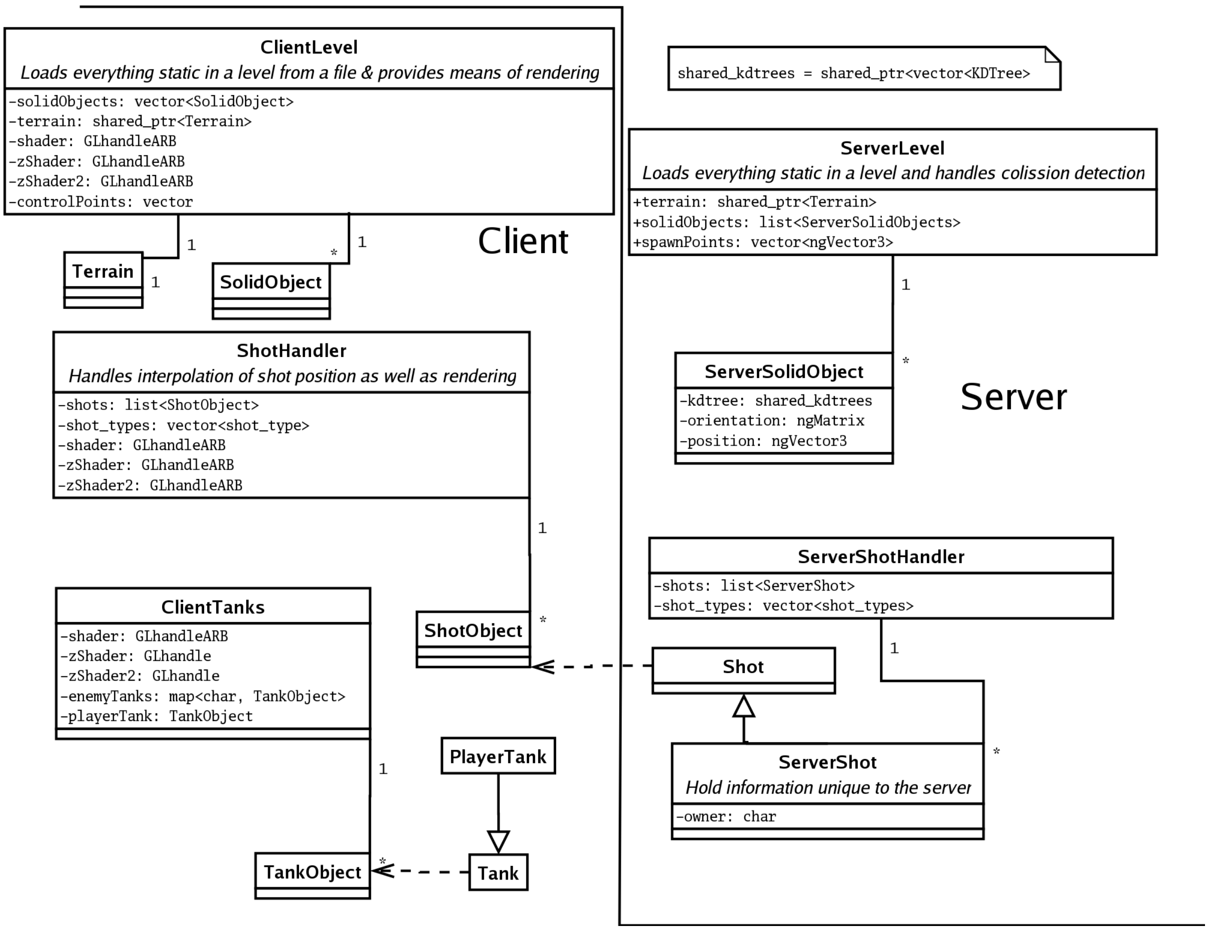
Figure 30: *The client receive logic in Quake 3.*

```
deltaCompressState(client.lastAckState, newState, &compressedState );
sendToClient( client, compressedState );
```

Figure 31: *The server send logic in Quake 3.*

```
update shot positions;
for each colided shot s
      spawn new explosion at s's position;
if any player dies,
      increase the shooter's score;
      respawn player;
Send new game state to all players
```

Figure 32: *Tank Action's main server loop.*



ClientLevel
Loads everything static in a level from a file & provides means of rendering
-solidObjects: vector<SolidObject>
-terrain: shared_ptr<Terrain>
-shader: GLhandleARB
-zShader: GLhandleARB
-zShader2: GLhandleARB
-controlPoints: vector
Terrain
SolidObject
Client
ShotHandler
Handles interpolation of shot position as well as rendering
-shots: list<ShotObject>
-shot_types: vector<shot_type>
-shader: GLhandleARB
-zShader: GLhandleARB
-zShader2: GLhandleARB
ClientTanks
-shader: GLhandleARB
-zShader: GLhandle
-zShader2: GLhandle
-enemyTanks: map<char, TankObject>
-playerTank: TankObject
ShotObject
PlayerTank
TankObject
Tank
shared_kdtrees = shared_ptr<vector<KDTree>
ServerLevel
Loads everything static in a level and handles colission detection
+terrain: shared_ptr<Terrain>
+solidObjects: list<ServerSolidObjects>
+spawnPoints: vector<ngVector3>
ServerSolidObject
-kdtree: shared_kdtrees
-orientation: ngMatrix
-position: ngVector3
Server
ServerShotHandler
-shots: list<ServerShot>
-shot_types: vector<shot_types>
Shot
ServerShot
Hold information unique to the server
-owner: char
1
*

Figure 33: *The client-server relationship. Note that there are* kd*-trees on the server and SolidObjects and TankObjects on the client. This means that only rendering data (such as meshes and textures) is on the client. Some things are of the same type on the server as on the client though. The shot for instance, is the same, but is observed by the ShotObject on the client - to enable rendering.*

7.2.4 Quake 3 Networking Model

The core concept with the Quake 3 Networking Model [93] is that there is only one main packet type, the client's necessary game state. This packet is a sequenced delta compressed [96] state built for each client from the last acknowledged game state and current state. Clients acknowledge entire states and never independent commands.

If a packet is dropped, the server will never resend the same packet but build a new one from the current state and packets recieved out of order will be discarded. There are no reliable packets, all reliable data (e.g. chat messages) will be sent repeatedly with the game state until the server receives acknowledgement for an update containing that data. For psuedo code, see program in fig. 31 and 30.

For example, if a player sends a chat message (reliable) with update 6, he will continually send that chat message on subsequent state updates until he receives notification from the server that it has received an update $>= 6$.

With this approach the server never waits for an acknowledgement. As a result, latencies are much lower but more bandwith is used. Because the server uses old game states when building new packets it may have to buffer a lot of data, especially if latencies are high.

7.2.5 Tank Action

Since the focus of this project was graphics and not gameplay or security, the TA network implementation is very simple. TA uses the client/server model for simplicity and the server is dedicated and console based. All communication uses UDP packets to ensure low latencies and keep bandwidth use as low as possible.

One major difference to most other fast-paced action games (e.g. Quake, Unreal, Half-Life, Tribes) is that TA trusts the client and allows it to just convey its position to the server. This in contrast to instead of making move requests or allowing the server to correct its position.

Refer to fig. 33 for a UML diagram of the client-server components in the game. Collision detection of projectiles (shots) is handled on the server. A ServerLevel object is used to load all data necessary for a game session and for creating all kD-trees used for collision detection. The ServerShotHandler is used for managing shots, adding new, remove old and update positions. kd-trees are on the server only, so the client is not burdend by generating these. Instead, the client holds bounding boxes for culling and box-box collision detection with static decorative objects.

Refer to fig. 32. The server main loop is fairly simple. Every frame, the server iterates over all shots and check whether a collision has occurred using the kD-trees for all objects, including terrain and players. If that is the case, the shot will be removed and an explosion will be spawned at that position and players in its vacinity will be damaged accordingly. The destruction of a tank will lead to that the player who shoot the final blow will get a "frag" and his score will increase. Any updated state is sent from the server to all clients when at least 20 ms has elapsed since the last update.

There are three main messages:

- NEW STATE The main message sent from the server ot all clients, containing positions and rotations for every player, position of every shot and information about new players. This packet will be sent to all players each server tick.
- NEW STATE Sent to server when a client moves, contains the new position and rotations.
- NEW SHOT Sent whenever a player fires a projectile.

7.3 Result and Conclusion

The network platform used in TA is very simple but very effective. Players can focus on their action and ignore the fact that all other tanks in the game are controlled by other players. By allowing each client to be in total control of his own position, no clientside refinement such as Client Prediction is of use.

Since TA does not have many interactive objects which need to be sent over network, a lot of optimization is not required. Huffman compression and quantization for instance, would have decreased the packetsize, but since the packetsize would never have been surpassed in the first place - it would not really matter. If the game were to be expanded on the other hand, this would have been a must.

7.4 Discussion

If we had used a common network model instead of our empirical solution, we think that we could have avoided some of the problems which we had during development. Things that are very simple in a single player game can become a really big problem when trying to keep things synchronized on several clients. Many problems can seem very elementary until actual implementation.

An example of this, are the shots. At our first implementation, the players sent two vectors for each shot. One which represented the position in world-space, and the other which represented the direction. This seemed to work well, but when several players and shots were present in a game, the amount of data being sent could easily surpass the limit of the packetsize. Because of this, we had to re-think how the shots should be handled over the network. Our first solution to this, was to on the client-side, calculate the directional vector of the shots from the difference in position of the position vectors. This seemed to work well, but with many players we had the same problem with the packetsize.

A final solution to this, was to instead send a directional vector and position vector only once to the clients, when a shot is fired. The clients then, would themselves simulate and predict movement of the shots. This works well, but synchronization is hard and is never exact.

8 Results and Conclusions

For the TA project, the use of the RUP model was greatly beneficial for many reasons. Programming early on allowed the team to better assess the time needed for each task, which led to more accurate planning. However, using an agile model without rigid early design also caused problems. An example of this is the network code, which had to be rewritten several times to cope with changing requirements on what data to be sent.

9 Discussion

We believe that the effectiveness of the team and the efficiency with which we conducted our respective tasks was significantly improved by dividing responsibilities amongst the team members. When programming independently, it is however important to have a unified vision of the product. We did experience some synchronization problems. For example, some untested code was committed to the subversion system and was assumed by others to be correct, although it was not tested until several months later. We gained great respect for the fact that independent programming must be balanced and closely coupled with effective means of team communication.

We also experienced that using a component-based software engineering methodology can save a lot of time. Most libraries proved to be valuable - and generally saved a lot of time. However, we also experienced that libraries sometimes can be more time consuming to use than writing the code from scratch. For instance, two 3DS model loader libraries were tried, and both turned out to be incorrect. However, in general, using libraries improved efficiency, allowing us to concentrate on high-level details. In general, the libraries seemed to be very well tested and robust.

We believe that a common style guide could have improved efficiency. This would have made it easier to read and edit each other's code.

Some code was unnecessarily rewritten because of our programming methodology. For example, the network code was rewritten several times. One conclusion is that lack of strict design early on in agile development doesn't necessarily have to mean not committing to solutions to some problems already from start. At the very least, more careful planning of the network packet layout in the early phases of the project could have improved our efficiency. However, our agile methods proved effective for graphics. Rather than being slowed down by introducing a scene graph before anything at all was known about the rendering code for objects, this was postponed until later in the development.

Some of us discovered that reading more about the subject before starting implementing specific features would have improved efficiency. For instance, the tank movement was first implemented with matrices, then implemented with quaternions. A lot of time was also lost in the implementation of particle systems, because it wasn't known at first that particles had to be sorted. We realized that a good source of information is usually to read scientific papers. Books are also good, but in our cases seldom went into depth regarding implementation details, and also were less up to date.

A Abbreviations

- TA - Tank Action
- GPU - Graphics Processing Unit
- API - Application Programming Interface
- LUT - Look-up Table
- LOD - Level of Detail
- FPS - First Person Shooter
- ODE - Open Dynamics Engine
- AABB - Axis-aligned Bounding Box
- OBB - Oriented Bounding Box
- RUP - Rational Unified Process
- SDO - Static Decorative Objects
- SLERP - Spherical Linear Interpolation
- SDL - Simple Direct Media Layer

References

[1] Kruchten P., “The Rational Unified Process: An Introduction”, 3rd ed., 1998.

[2] Beck K., “Extreme Programming Explained: Embrace Change”, Addison-Wesley Professional, US Ed edition, 1999, ISBN: 0-201-61641-6.

[3] http://subversion.tigris.org/, 2007.

[4] Richard S. Wright, Jr. and Michael Sweet, “OpenGL SuperBible”, 2000, ISBN: 1571691642.

[5] Jackie Neider, Tom Davis, Mason Woo, “OpenGL Programming Guide”, Release 1, 1994.

[6] Guennadi Riguer, “Performance Optimization Techniques for ATI Graphics Hardware with DirectX 9.0”, ATI Technologies Inc., 2002.

[7] Paul S. Strauss, “The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor”, ACM Press, 1993.

[8] M. E. Newell, R. G. Newell, T. L. Sancha, “A solution to the hidden surface problem”, ACM Press, 1973.

[9] Evan Pipho, “Focus On 3D Models (Game Development)”, Course Technology PTR, 2002, ISBN: 1592000339.

[10] Randi Rost, “OpenGL Shading Language”, 2nd Edition, 2006.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides., “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994, ISBN: 0-201-63361-2.

[12] Dirk Staneker, “A First Step towards Occlusion Culling in OpenSG PLUS”, WSI/GRIS, University of Tubingen, Germany, 2002.

[13] Sowizral, H., “Scene graphs in the new millennium”, IEEE, Sun Microsystems, USA, 2000, ISSN: 0272-1716.

[14] Jon Louis Bentley, “Multidimensional binary search trees used for associative searching”, ACM Press, Stanford University, 1975, ISSN: 0001-0782.

[15] Stefan Aric Gottschalk, "Collision queries using oriented bounding boxes", ACM Press, 2000, ISBN:0-493-01573-6.

[16] Lars Bishop, Mark Finch, Michael Shantz, “Designing a PC Game Engin”, IEEE Computer Graphics and Applications, Vol 18, Issue 1, pp 46 - 53, January 1998.

[17] Rick Gibson,David MacQueen, “Outsourcing in Next Generation Games Development: Delivering cost and production efficiency”, Screen Digest, 2006.

[18] Shanika Karunasekera, Scott Douglas, Egemen Tanin, and Aaron Harwood, “P2P Middleware for Massively Multi-player Online Games”, 6th Internation Middleware Conference, ACM/IFIP/USENIX, 2005.

[19] Alan H. Barr, Bena Currin, Steven Gabriel, John F. Hughes, “Smooth interpolation of orientations with angular velocity constraints using quaternions”, ACM SIGGRAPH Computer Graphics, 1992.

[20] http://www.ageia.com/, 2007

[21] http://www.havok.com/, 2007

[22] Ken Shoemake, “Animating rotation with quaternion curves”, SIGGRAPH; ACM PRESS, 1985, ISSN:0097-8930.

[23] Kevin P. Smith, Chris Frazier, “The OpenGL Graphics System Utility Library”, Silicon Graphics, 1995.

[24] Russel H. Taylor, “Planning and Executiono f Straight Line Manipulator Trajectories”, IBM J. DEVELOP., VOL.23, NO. 4, July 1979

[25] William R. Hamilton, “On Quaternions; or on a new System of Imaginaries in Algebra”, Philosophical Magazine, 1844-1850.

[26] David Eberly, “Quaternion Algebra and Calculus”, Geometric Tools Inc., 1999 (Updated 2002)

[27] Russel Smith, “Open Dynamics Engine User-Guide”, 2006, http://www.ode.org/ode-latest-userguide.html

[28] Freidman, J. H., Bentley, J. L., and Finkel, R. A. ,“An Algorithm for Finding Best Matches in Logarithmic Expected Time.” ACM Trans. Math. Softw 1977.

[29] 2nd Edition, Tomas Akenine-Moller, Eric Haines, “Real-Time Rendering”, 2002, ISBN: 1568811829.

[30] Raphael Finkel and J.L. Bentley, “Quad Trees: A Data Structure for Retrieval on Composite Keys”, Acta Informatica 4 (1): 1-9. 1974

[31] Elmar Langetepe and Gabriel Zachmann, “Geometric Data Structures for Computer Graphics”, A K Peters Ltd (February 1, 2006), ISBN: 1568812353. 2006

[32] Chong Jin Ong Gilbert, E.G. ,"Fast versions of the Gilbert-Johnson-Keerthi distance algorithm", 2001 Robotics and Automation, IEEE Transactions on Aug 2001.

[33] Brian Mirtich, "V-Clip: Fast and Robust Polyhedral Collision Detection", ACM Transactions on Graphics. 1998

[34] Gregory Michael Hunter, "Efficient computation and data structures for graphics", "PhD Thesis, Order Number: AAI7823520, 1978.

[35] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Ingo Wald, and Peter Shirley, "Packet-based Whitted and Distribution Ray Tracing", 2007.

[36] Jacco Bikker, "Flipcode Image of the day", http://www.flipcode.com/cgi-bin/fcarticles.cgi?show=65091 as seen 2007-06-15.

[37] Jacco Bikker, "Raytracing Topics & Techniques", http://www.flipcode.com/articles/article_raytrace01.shtml as seen 2007-05-21

[38] Ingo Wald, "Realtime Ray Tracing and Interactive Global Illumination", PhD Thesis, Saarland University. 2004

[39] D. M. Mount and S. Arya. , "ANN: A library for approximate nearest neighbor searching.", Center for Geometric Computing 2nd Annual Fall Workshop on Computational Geometry, 1997, http://www.cs.umd.edu/ mount/ANN .

[40] S. Maneewongvatana and D. M. Mount. , "It's okay to be skinny, if your friends are fat.", 4th Annual CGC Workshop on Comptutational Geometry. 1999

[41] Ingo Wald, and Vlastimil Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)", Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pages 61-69. 2006

[42] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. "I-COLLIDE: An interactive and exact collision detection system for large-scale environments." In Pat Hanrahan and Jim Winget, editors, 1995 Symposium on Interactive 3D Graphics, pages 189-196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[43] Ingo Wald, Solomon Boulos, and Peter Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies", 2007.

[44] W Hunt, WR Mark, G Stoll, "Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic", 2006.

[45] Kainz F., Bogart R., Hess D., "The OpenEXR File Format", GPU Gems, 2004.

[46] Duchaineau M., Wolinsky M., Sigeti D. E., Miller M. C., Aldrich C., Mineev-Weinstein M. B. ,“ROAMing Terrain: Real-time Optimally Adapting Meshes”, IEEE Visualization, pp 81-88, 1997.

[47] Lindstrom P., Pascucci V.,“Visualization of Large Terrains Made Easy”, Proceedings of IEEE Visualization 2001, pp. 363-370, 574, October 2001.

[48] de Boer W. H.,“Fast Terrain Rendering Using Geometrical MipMapping”, Fast Terrain Rendering Using Geometrical MipMapping. http://www.flipcode.com/articles/article_geomipmaps.pdf, 2000.

[49] Vistnes H., “GPU Terrain Rendering”, Game Programming Gems 6, pp 461-471, 2006.

[50] Wagner D.,“Terrain Geomorphing in the Vertex Shader” ShaderX-2, Wordware Publishing, 2003.

[51] Vistnes H., “GPU Terrain Rendering”, Game Programming Gems 6, pp 461-471, 2006.

[52] Shankel J., “Fast Heightfield Normal Calculation”. Game Programming Gems 3, Charles River Media, 2002.

[53] Losasso F., Hoppe H.,“Geometry clipmaps: Terrain rendering using nested regular grids.”, ACM SIGGRAPH, pp. 769-776, 2004.

[54] Kryachko Y.,“Using Vertex Texture Displacement for Realistic Water Rendering”, GPU Gems 2, 2005.

[55] Clasen, M., Hege, H.-C. “Terrain rendering using spherical clipmaps,” In Proc. EuroVis, pp 91–98, 2006.

[56] Asirvatham A., Hoppe H., “Terrain Rendering Using GPU-Based Geometry Clipmaps”. 2005, GPU Gems 2 .

[57] Young I.T., Gerbrands J.J., van Vliet L.J., “Image Processing Fundamentals”, http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip.html

[58] Rader C., Brenner N., “A new principle for fast Fourier transformation”, IEEE Acoustics, Speech & Signal Processing 24: pp 264-266, 1976.

[59] Pelzer K., “Rendering Countless Blades of Waving Grass”, GPU Gems, 2004.

[60] Woop S., Schmittler J., Slusallek P., “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing”. Proceedings of ACM SIGGRAPH 2005.

[61] Channa K., "Light Mapping - Theory and Implementation", http://www.flipcode.com/articles/article_lightmapping.shtml, 2003.

[62] Bui-Tuong P., "Illumination for Computer-Generated Images", Phd Thesis, 1973.

[63] Blinn, J. F., "Simulation of Wrinkled Surfaces", Computer Graphics, Vol. 12 3, pp. 286-292 SIGGRAPH-ACM, 1978.

[64] Tomomichi Kanek, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, Susumu Tachi, "Detailed Shape Representation with Parallax Mapping", Proceedings of ICAT 2001, 2001.

[65] Kautz J., Heidrich W., Daubert K., "Bump map shadows for OpenGL rendering", Max-Planck-Institut für Informatik, Saarbrücken, Germany, MPI-I-2000-4-001, 2000. citeseer.ist.psu.edu/kautz02bump.html

[66] Cohen J., Tchou C., Hawkins T., Debevec P., "Real-time High Dynamic Range Texture Mapping", Eurographics Rendering Workshop 2001, London, England, 2001. http://www.debevec.org/Research/HDRTM/egwr-01-cohen.pdf

[67] Shastry A. S., "High Dynamic Range Rendering". http://www.gamedev.net/columns/hardcore/hdrrendering/, 1999.

[68] Crow F. C., "Shadow algorithms for computer graphics", Computer Graphics $Proc.of SIGGRAPH 77$, 112:242-248, 1977.

[69] Williams L., "Casting curved shadows on curved surfaces", 1978.

[70] Assarsson U., "A Real-Time Soft Shadow Volume Algorithm". PhD thesis, Department of Computer Engineering, Chalmers University of Technology. ISBN 91-7291-333-9, 2003.

[71] Stamminger M., Drettakis G., "Perspective Shadow Maps", 2002.

[72] Reeves W. T., Salesin D. H., Cook R. L.. "Rendering antialiased shadows with depth maps." Computer Graphics $SIGGRAPH 87 Proceedings$, pp 283-291, 1987.

[73] Arvo, J., "Tiled shadow maps.", Proceedings of Computer Graphics International 2004, IEEE Computer Society, pp 240-247, 2004.

[74] Zhang F., Sun H., Xu L., Lun L. K., "Parallel-Split Shadow Maps for Large-scale Virtual Environments", ACM VRCIA'06, 2006, http://appsrv.cse.cuhk.edu.hk/ fzhang/pssm_project/shadow_vrcia.pdf.

[75] Shastry A. S., "Soft-Edged Shadows", http://www.gamedev.net/reference/articles/article2193.asp, 2005.

[76] Shastry A. S., “High Dynamic Range Rendering”. http://www.gamedev.net/columns/hardcore/hdrrendering/, 1999.

[77] Nishita T., Dobashi Y., Kaneda K., Yamashita H., “Display Method of the Sky Color Taking into Account Multiple Scattering”, Proc. of Pacific Graphics 1996, pp.117-132, 1996-8.

[78] O’Neil S., “Accurate Atmospheric Scattering”, GPU Gems 2, 2005.

[79] Abad J. A., “A simple model for fast, realistic sky dome color rendering”, http://www.geocities.com/ngdash/whitepapers/skydomecolor.html, 2006.

[80] Perlin K., “Making Noise”, http://www.noisemachine.com/talk1/, based on a talk presented at GDCHardCore, 1999.

[81] Ebert et al. “Texturing and Modeling - A procedural Approach”, 3rd ed., 2002.

[82] Elias H., “Cloud Cover”, http://freespace.virgin.net/hugo.elias/models/m_clouds.htm, 2000.

[83] Reeves W. T., “Particle Systems - a Technique for Modeling a Class of Fuzzy Objects”, ACM Transactions on Graphics *TOG* archive, Volume 2, Issue 2 *April*1983, ISSN:0730-0301, pp 91-108, 1983.

[84] Nguyen H., “Fire in the Vulcan Demo”, GPU Gems, 2004.

[85] Everitt C., “http://developer.nvidia.com/object/order_independent_transparency.html”, 2004.

[86] Sousa T., “Generic Refraction Simulation”, GPU Gems 2, 2005.

[87] van der Burg J., “Building an Advanced Particle System”, http://www.gamasutra.com/features/20000623/vanderburg_01.htm, 2000.

[88] , “Standard Template Library Programmer’s Guide”, 1993-2006, “http://www.sgi.com/tech/stl/”.

[89] Musser D. R., “Introspective Sorting and Selection Algorithms”, Software Practice and Experience 278:983, 1997.

[90] “Source Multiplayer Networking.”, http://developer.valvesoftware.com/wiki/Source_Engine, 21-05-2007

[91] Mark Frohnmayer, Tim Gift, “The TRIBES Engine Networking Model.”, GDC, March, 2000.

[92] Tim Sweeney, “Unreal Networking Architecture.”, http://unreal.epicgames.com/Network.htm, 1999

[93] “The Quake3 Networking Model.”, http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking, 21-05-2007

[94] Jesse Aronson, “Dead Reckoning: Latency Hiding for Networked Games”, http://www.gamasutra.com/features/19970919/aronson_01.htm, 1997

[95] Comer, Douglas E. “Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture”, Prentice Hall, 1995. ISBN 0132169878

[96] James J. Hunt, Kiem-Phong Vo, Walter F. Tichy, “Delta algorithms: an empirical analysis”, ACM Press New York, NY, USA, 1998

[97] “RFC791”, http://tools.ietf.org/html/rfc791, 1981

[98] D.A. Huffman, “A method for the construction of minimum-redundancy codes”, Proceedings of the I.R.E., sept 1952, pp 1098-1102

[99] Joy, William, Robert Fabry, Samuel Leffler, M. Kirk McKusick, and Michael Karels. “Berkeley Software Architecture Manual 4.3BSD Edition”, Department of Electrical Engineering and Computer Science. University of California, Berkeley, California 94720, April, 1986.