CHALMERS

WipeIn - F-ϵ

- A 3D Action Game

Bachelor's Thesis
Computer Science and Engineering Programme

CHRISTOPHER ANDERSSON
JESPER LINDH
MIKAEL MÖLLER
MIKAEL OLAISSON
KARL SCHMIDT
CARL-JOHAN SÖDERSTEN
ALLAN WANG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2011

Bachelor's Thesis DATX11-09 - Rally Sport Racing Game

Abstract

The following thesis describes a case study of 3D game programming. It involves the evaluation of several techniques commonly used in real-time rendering, as well as some associated fields such as modelling, collision handling and sound.

We will investigate which of the many options available are the most efficient, as well as which areas are preferably put aside, in the aim of achieving an entertaining and visually appealing 3D computer game within a short time span.

Contents

1 Introduction **5**
1.1 Background 5
1.2 Purpose 5
1.3 Problem 5
1.4 Limitations 5
1.4.1 Contents 6
1.4.2 Areas of focus 6
1.4.3 Open-source code 6
1.4.4 Computer power 6
1.5 Method 6
1.5.1 Choice of programming language and framework . . . 7
1.5.2 API 7
1.5.3 Development process 8
1.6 Game design 8

2 Graphics **9**
2.1 Pipeline 9
2.2 The application stage 10
2.3 The geometry stage 10
2.4 The rasteriser stage 12
2.4.1 Hidden surface determination 12
2.5 Shading 14
2.5.1 The Phong Shading Model 14
2.5.2 Bidirectional Reflectance Distribution Functions . . . 16
2.5.3 Global illumination 16
2.5.4 Shadows 17
2.5.5 Reflections 18
2.5.6 Deferred shading 18
2.6 Method 20
2.6.1 Shading 21
2.6.2 Global Illumination 22
2.7 Future work and discussion 23

3 Particle system **24**
3.1 Introduction 24
3.2 Method 24
3.3 Results 25
3.4 Discussion 26

4 Modelling **27**
4.1 Introduction 27
4.2 Method 27
4.2.1 Polygonal modelling 27
4.2.2 Splines modelling 28
4.2.3 Sculpt modelling 28
4.3 UVW mapping 29
4.4 Texture mapping 30
4.4.1 Diffuse map 30
4.4.2 Specular map 30
4.4.3 Bump map 31
4.5 Model import 32
4.6 Results and discussion 32

5 Physics **35**
5.1 Method 35
5.1.1 Architecture 36
5.1.2 Algorithms 37
5.1.3 Collision shapes 38
5.1.4 Ship motion 40
5.2 Results and discussion 41

6 Sound and music **43**
6.1 Background 43
6.2 Method 43
6.3 Results 43
6.4 Discussion 44

7 Conclusion **45**
7.1 Results 45
7.2 Discussion 45
7.3 Future work 45

8 Appendix - Game design **52**
8.1 Goal 52
8.2 Controls 52
8.3 Energy 52
8.4 Power-ups 53
8.4.1 General 53
8.4.2 Offensive power-ups 54
8.4.3 Defensive power-ups 55
8.4.4 Combos 55
8.5 Grafical User Interface 56
8.6 Menus 56

1 Introduction

1.1 Background

The computer game industry is a relatively new one, but a rapidly growing one notwithstanding. Although its birth can be traced back as early as the 1950s [36], with the 1961 game *Spacewar!* credited as the first influential computer game, the modern gaming industry is often considered as the period following the 1983 North American Game Crash.

Over a decade ago the industry was just $7 billion [40], while in 2007 about $41.9 billion. This number is expected to grow 9.1% annually to $68 billion in 2012, making it fastest-growing component of the media sector worldwide [14].

Racing games do not figure among the best selling games though. Between 2004 and 2009, none of the ten best-selling computer games in Sweden was a racing game akin to ours [38]. Still, the genre is one of the classical ones, and the game that has been our major source of inspiration, *Wipeout*, is, despite being first released in 1995, still a popular game, with new editions getting developed every other year [61].

1.2 Purpose

The purpose of this project is to design and implement a fully functional 3D computer game within a limited time frame. Project requirements include a specific emphasis on the graphical parts, i.e., creating a visually appealing game.

1.3 Problem

When working on such a vast project, the problems encountered are multiple and diverse. They range from global, conceptual ones to precise, technical ones. Problems can be risen as early as "What is a computer game?" and "What makes a game entertaining?". However, as the purpose of this game lies in the graphics field, the main problem will be to evaluate and decide which rendering techniques will prove to be the most efficient to reach our intended goal. Of course, the problem will broaden out to a few other areas that game programming comprises, such as sounds and music, physics engine and networking. The subsequent results will hopefully be of great value for ourselves and other future programmers who intend to create 3D games with a short time span.

1.4 Limitations

As is hinted in the above purpose, our biggest enemy in this project was time itself. We had but four months to build a three-dimensional, visually

appealing, fully functional and hopefully jolly entertaining computer game, starting from scratch, while being far from experts in the field. It was a challenge, and to achieve our goal we had to confine our ambitions and aim our focus at certain areas.

1.4.1 Contents

One of the early decisions we took was to design only one track. Should we continue working on the game in the future, this will evidently be changed.

The environment detailing of the track is also limited, and we had to agree on choosing just a few of the myriad of suggestions that emerged during the preliminary track designing.

1.4.2 Areas of focus

Again, it is clearly stated in the purpose of this project that focus had to be put on graphics, i.e., rendering techniques. Other potential areas of deeper analysis, such as physics, linear algebra and networking, are therefore only treated briefly in this report.

1.4.3 Open-source code

In order to save a lot of precious time, we made use of open-source code as well as available algorithms at some points, since it was well within the rules of the project (if not encouraged). Of course, this is accompanied by clearly stated references in the actual code.

1.4.4 Computer power

Although this might seem obvious, it can be worth mentioning that the finished game has since the beginning always been intended to be playable on a standard personal computer, such as the ones used to create the game.

1.5 Method

Early in the development process, a series of critical choices had to be made, the very first one regarding the type of racing game we wanted to design. The reason for setting the game in a space environment was purely out of personal preference. It allowed our imaginations to spin wildly, and ideas were not as restricted as they would have been if we had chosen a realistic automobile-based game.

Once the theme had been settled, the name *WipeIn F-ϵ* was agreed upon, the double video-game pun involving enough geeky humour to satisfy the most dedicated programmer.

Eventually, more crucial issues emerged, among which the following had to be treated before starting with the game implementation.

1.5.1 Choice of programming language and framework

One of the earliest decisions we had to make was the choice of computer language and framework (or IDE, Integrated Development Environment) combination. The interesting candidates were C# / XNA and C++ /Microsoft Visual Studio.

Both C# and C++ are high-level programming languages well suited for a project like ours, with lots of available libraries. The reason for choosing C++ was mainly its portability and generally better performance, together with the fact that most of us had more experience of it than of C#.

Regarding the choice of framework, we opted for our own implementation, mainly because it allowed for more flexibility than XNA and required more actual programming, which we saw as a pleasant challenge.

1.5.2 API

The choice of API (Application Programming Interface) stood between Microsoft Direct3D and OpenGL. We opted for Direct3D, since it is nowadays more wide-spread than OpenGL, and we reckoned that being accustomed to it would be of more value than to its counterpart in future endeavours.

Microsoft Direct3D is a part of Microsoft's DirectX application-programming interface. Direct3D came to light in 1992, based on an API used in medical imaging and CAD (Computer-Aided Design) software, but was first released as a DirectX component in 1995. Several versions have since then been developed, with the newest one labelled 11, which was the one we worked with.

Direct3D is used to render three-dimensional graphics in applications where performance is important, and uses hardware acceleration techniques for the *3D rendering pipeline*. It also exposes the advanced graphics capabilities of 3D graphics hardware, including Z-buffering, anti-aliasing, alpha blending, mipmapping, atmospheric effects and perspective-correct texture mapping, some of which useful to us in this very project [62].

Additionally, shader model 4.0 was selected as the compile target for our shader programs, as it reduces the requirement of the application to Direct3D 10 hardware, first launched in early 2007.

We will go through the 3D pipeline as well as explain the different steps in the rendering process in subsequent sections.

1.5.3 Development process

Upon starting the project, the use of a software development methodology, in particular Agile Development [7], was discussed, but eventually dismissed, despite the magnitude of the project. We did use a version control software though, since the nature of the work that awaited us inevitably led to individual code editing. For this purpose, Subversion (SVN) was chosen, simply because it was the one provided to us.

1.6 Game design

The game design is attached as an appendix.

2 Graphics

2.1 Pipeline

When presenting a thesis about 3D programming, a description, albeit brief, of the *graphics rendering pipeline*, or simply *pipeline*, is due. The use of the metaphorical pipeline derives from the analogy to the physical world, where a pipeline consists of several stages [30]. In an oil pipeline, for instance, oil cannot flow from one stage of the pipeline to another stage unless also flowing in all other stages. This implies that no matter how fast the fastest pipeline stage is, the overall speed of the pipeline will always be determined by the slowest part of the pipeline [1].

The task of a graphics renderer is to generate a two-dimensional image from a set of data. There are many possible ways of doing this, from tracing photon paths to algorithms based on fractals, but the most common method in real-time applications is to use meshes of triangles as the base datatype and essentially draw the triangles on a canvas. Each *vertex*, i.e., node, of the mesh is associated to a set of data—position, surface normal, tangent & binormal, colour, reflectivity, etc—for the renderer to use in generating a correct image.

The basic construction of the pipeline consists of three conceptual stages: the *application stage*, the *geometry stage* and the *rasteriser* stage. This basic structure is the nucleus and engine of the rendering pipeline, and each of these structures is a pipeline in itself, as is illustrated in figure 2.1.

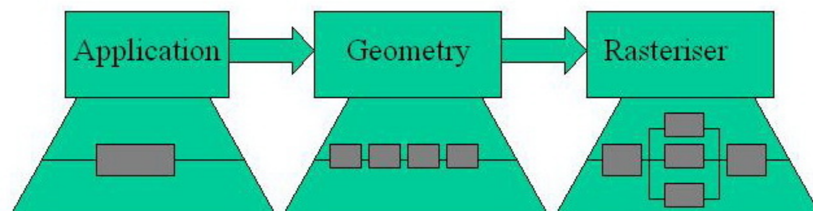


Figure 1: *The rendering pipeline.*

Like the aforementioned oil metaphor, it is the slowest part of the pipeline stages that determines the *rendering speed*, i.e., the update speed of the images, often expressed in frames per second (fps). The maximum rendering speed is thus obtained by tracking down the bottleneck that often emanates at this slowest part.

The first computational technique for rendering solids was ray tracing, which in essence entails casting a ray from a fictive eye through a camera mesh and calculating where it intersects an object present in the scene.[3]

Further ray casts can then be made at the point of intersection, to better determine its colour, such as casting towards a light source to check for occlusion or illumination, and casting towards the angle of reflection. Ray tracing algorithms are, as a rule, slow to compute and unsuitable for real time applications. However, they can be made very accurate and encompass several effects that more advanced techniques do not. They are useful when pre-computing data, such as the precise lighting conditions in a static room, for use in a real-time setting.

Instead of ray tracing, real-time applications and modern graphics cards work by manipulating *meshes* of geometrical primitives, typically triangles, that are transformed down to 2D space, sorted to determine priority, shaded to determine their colour, and finally rasterised into the individual pixels of a 2D image. An overview of each of these will be presented in turn, as well as their hardware implementations.

The following sections offer a brief summary of the theory and research behind real-time rendering, as well as a description of the rendering pipeline as implemented on modern graphics hardware. Later, the techniques made us of will be described in depth.

2.2 The application stage

The application stage relies entirely on the programmer, since it always executes on the software, as opposed to the geometry and rasteriser stages that are also built upon hardware. However, it is possible to implement speed-up techniques that affect the other pipeline stages, for example by decreasing the number of triangles to be rendered, or through acceleration algorithms [1].

The application stage calculates and generates a number of important processes, most notably collision detection, input handling (from keyboard and mouse) and animation of all sorts, such as texture animation, animations via transforms and geometry morphing.

The most important task of this stage is to send rendering primitives (e.g., points, lines, triangles) to the graphics hardware in order to eventually get the desired output on the screen.

2.3 The geometry stage

The second step of the rendering pipeline is the geometry stage, in which the 3D representation of the scene is projected down to 2D space. In general, this is done by applying a matrix transform to the vertex positions. A simple projecting transform can be given as

$$M_p = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{1}$$

M_p will transform a 3D vector $[x, y, z]'$ as $M_p x = [x, y]'$, by simply dropping the z-coordinate, creating an orthogonal projection.

Simply projecting data will not suffice in practice though, and the geometry pipeline must support an array of transforms, e.g., rotations, scaling, translations, shearing, etc. To enable these, homogeneous coordinates are used, in which the vectors are expanded to 4D to support a greater range of linear transforms. A point $[x\,y\,z]'$ is represented in homogeneous coordinates as $[x\,y\,z\,1]'$. This enables the use of matrices for translation [35], as

$$\begin{pmatrix} 1 & 0 & 0 & x' \\ 0 & 1 & 0 & y' \\ 0 & 0 & 1 & z' \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + x' \\ y + y' \\ z + z' \\ 1 \end{pmatrix} \tag{2}$$

A vector that need remain unaffected by a translational transform can be represented as $[x, y, z, 0]'$. This can be practical for items like surface normals and similar directional data. Several transforms chained together follow the associativity of matrix multiplication property, i.e., $M_1(M_2x) = (M_1M_2)x$ [35].

It is typical that one does not provide the rendering pipeline with individual rotations, scalings, etc, but instead precomputes the aggregate transforms in the application and supply these to the renderer. A few of these aggregate transforms used in computer graphics are ubiquitous, and therefore deserve particular mention.

M_w The *m*odel to *w*orld matrix, which transforms the vertices of a model into the main scene coordinate system.

M_v The *w*orld to *v*iew matrix, which transforms from world coordinates to the camera's view. This generally places the camera at the origin, looking down the z-axis.

M_p The *v*iew to *p*rojection matrix, which transforms view coordinates down to 2D space or, more commonly, a *clip space* cube.

M_{vp} The *v*iew*p*ort transform, which is a simple translation and scaling to convert the $[0, 1]$ or $[-1, 1]$ coordinates used in clip space to the [0, width] and [0, height] dimensions used for the image we render to.

Consumer-oriented hardware implementations of the geometry stage started with nVidia's GeForce 256 in 1999, which used a fixed-function vertex pipeline. The user supplied the transforms outlined above and the hardware applied

them to vertex data. This implementation was relatively swiftly supplanted by *vertex shaders*; small, flexible programs that run for each vertex. A vertex shader is able to freely manipulate the data associated with a vertex and is not limited to the transforms above [41]. However, since these matrices offer an efficient and compact description of many common transforms, and graphics processors are highly efficient at vector operations, they remain extremely common [1].

A yet more recent innovation is the addition of *geometry shaders*, programs run on the output from the vertex shader, able to generate additional vertices from this data.

2.4 The rasteriser stage

While rasterisation generally refers to the entire process of taking vector data and converting it to pixel (raster) data, in the context of the rendering pipeline the rasterisation stage refers to the specific substep of taking the projected 2D geometry and generating individual pixels.

Historically this was generally done through scanline rendering [12], where, for each row of the target image, one would maintain a sorted-edge table of the edges of the polygons in that row, sorted from left to right. This is in order to generate the span of each polygon for that row, which can be easily used to determine which polygon applies to each pixel of the row. Scanline rendering is still effective for scenes with a relatively low polygon count, and hardware implementations are used in devices like the Nintendo DS [63].

2.4.1 Hidden surface determination

When projecting a 3D mesh down to a 2D plane, several primitives end up being projected onto the same region on the plane of projection. A central problem of 3D rendering is that for each pixel one needs to determine which of the primitives mapping to that pixel ought to be the displayed one.

The first and simplest solution to this problem is the painter's algorithm [49], in which primitives are drawn in sorted order, with the most distant primitive drawn first and closest primitive last. Subsequent drawings then overwrite the previous data, if any. There are, however, several problems with this approach. First, primitives must be sorted, which is expensive for any dynamic scene. Second, processing time is wasted on calculations of pixels that are not visible in the final image. Third, artefacts may appear with primitives that are not unambiguously ordered, i.e., when drawing two intersecting triangles, one will appear to cover the other.

Coverage buffers, or C-buffers, were common in real-time graphics in the mid-90s and can still be used in some situations today [8]. They can be thought of as a reverse painter's algorithm. The buffer is a matrix with a dimension of equal size to the viewport. Primitives are sorted front-to-back,

and, when drawing, one both draws colour data in the render target and updates the corresponding region in the C-buffer, as covered. Pixels marked as covered are not drawn to in the draw call. This has the advantage of avoiding expensive colour calculations for hidden surfaces, but the sorting requirement and issues with ordering remain.

Z-buffering [15] is the most common algorithm for solving hidden surface determination (HSD) today. As with C-buffering, a buffer of equal size to the render target is created, but where a C-buffer stored a simple boolean, a Z-buffer stores a depth value, typically in a fixed point format (illustrated in Figure 2). When drawing a pixel, its depth is compared to that stored in the buffer. If it is closer, the pixel is drawn; if not, the pixel is discarded. This has many advantages. We are no longer dependent on a strict ordering of primitives, although rendering mostly front-to-back improves performance. Intersecting primitives are handled correctly, although artifacts can appear for surfaces with very similar depths due to the limited precision of the buffer.

Figure 2: *A Z-buffer, storing the depth value of a pixel.*

The Z-buffer is ubiquitous in modern real time applications. In both the DirectX and OpenGL pipelines, geometry is projected not onto a plane but onto a cube. In DirectX, the cube has the xy-dimensions of $[-1, 1]$ and the z-dimension of $[0, 1]$. Any primitive lying outside the cube is clipped by its edges, and the contents of the cube is what is actually projected onto the viewport by the viewport transform, using the z-direction as sorting data for the Z-buffer.

Several other solutions to the hidden surface problem exist, such as the Active Edge List used in the original Quake engine, and the Warnock algorithm[59]. However, they are less relevant today, and sadly describing all of them lies outside the scope of this thesis.

2.5 Shading

Shading is the process of determining the colour of a pixel once it has been rasterised. The main theoretical framework, in which shading can be understood, is Kajiya's Rendering Equation [31]. A slightly simplified version of the equation, where time and spectral dependency are ignored, can be written as

$$L_o(x,\omega) = L_e(x,\omega) + \int_\Omega f_r(x,\omega',\omega)L_i(x,\omega')(\omega' \cdot n)d\omega' \tag{3}$$

This states that the outgoing light L_o at a point x and direction ω is the sum of the emitted light L_e and reflected light. The reflected light is, in turn, the incoming light L_i, multiplied by the cosine of the angle of incidence and the value of the bidirectional reflectance distribution function (BRDF) f_r, summed over all directions ω' in the hemisphere Ω.

The rendering equation is not, in itself, a full description of light transfer in reality, as it is applicable exclusively to geometric optics. It does not account for wave phenomena such as diffraction and polarisation, outgoing light is assumed to be at the same wavelength, time and place as incoming light, etc. It is, however, more than sufficient for most situations.

There is currently no known algorithm that accurately solves the rendering equation in real-time, as integrating over a hemisphere is generally not something one does to any significant degree of accuracy in a few milliseconds. The focus of research is on creating passable approximations to the solution, using the capabilities of modern graphics hardware to be as accurate as possible, while maintaining a high frame rate.

The first issue to tackle in shading is how many colour samples to use for each primitive drawn. The simplest option is flat shading, where one colour is used for each primitive. The next step up from this is Gouraud shading [26], where colour is calculated for each vertex of the primitive and then interpolated. These techniques are rarely used today, and modern techniques are instead based on Phong shading [50], where colour is calculated per pixel of the primitive, using interpolated values for the surface normal of each pixel, instead of linearly interpolating the colour itself. This is significantly more calculation-intensive, but the result is much improved.

2.5.1 The Phong Shading Model

In the original Phong shading model, the radiance of a pixel is calculated as the sum of three different types of reflected light: ambient reflection, diffuse reflection and specular reflection. Among these, the diffuse and specular terms need to be calculated for each light source. The final pixel colour is the sum of all terms. The light intensity I_p at a point p in the Phong model can be calculated as

$$I_p = k_a i_a + \sum_{m \,\in\, \text{lights}} (k_d (L_m \cdot N) i_{m,d} + k_s (R_m \cdot V)^{\alpha} i_{m,s}) \tag{4}$$

Where

i_x Ambient, diffuse and specular light intensities of the source light.

k_x The material's ambient, diffuse and specular reflection constants.

N Surface normal.

L Direction from the surface to the light source.

R Reflection of $-L$ on the surface, using the Householder transformation $R = 2(L \cdot N)N - L$.

V Viewer direction.

α A measure of the material's shininess. A high shininess means smaller, more intense specular highlights.

In some situations it can be more efficient to replace the Phong model's specular term $R \cdot V$ with a term $N \cdot H$, where H is the halfway vector between the viewer and the light source $H = (L + V)/|L + V|$. This alteration is known as the Blinn-Phong model [9], illustrated in Figure 3, and produces similar results to the Phong model. The advantage of this form is that for a directional light at infinite distance, such as the sun in an outdoor scene, H only needs to be calculated once per light. R, in contrast, must be calculated separately for each pixel of the scene.

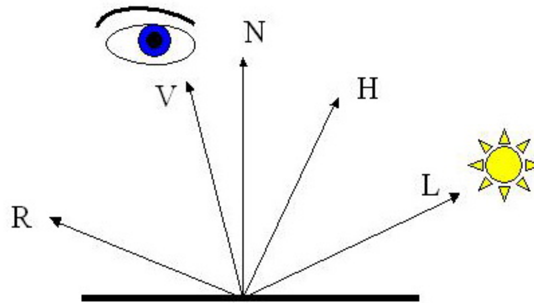


Figure 3: *The vectors used in the Phong and Blinn-Phong shading models.*

2.5.2 Bidirectional Reflectance Distribution Functions

Recall Kajiya's rendering equation 3, where the outgoing light contribution from a light source in a direction ω' was given by

$$f_r(x, \omega', \omega) L_i(x, \omega')(\omega' \cdot n)$$

The Phong and Blinn-Phong models both provide examples of a BRDF f_r in this framework. They are reasonably simple to calculate, but make a number of unrealistic assumptions: all lights are assumed to be point lights, the surface material is assumed to reflect light isotropically (with no concern for direction) and ambient lighting is assumed to be equal everywhere. These are seldom correct, and while the models are reasonable for certain materials, like paint or plastic, they tend to fail when rendering skin, metal and other surfaces that do not adhere to their assumptions.

Whatever its failings, as with the standard geometrical transformation matrices, the model remains a powerful and widely used tool. In initial hardware implementations of the rendering pipeline, the users were limited to setting the material and light constants in Blinn-Phong, in the same way they were limited to setting matrices in the geometry stage. However, much like the introduction of programmable vertex shaders allowed arbitrary transforms in the geometry stage, *pixel shaders* (also called *fragment shaders*) have been introduced to allow for the same in the shading stage. A pixel shader is a short program that takes, as input, the output of a vertex shader, linearly interpolated from the output provided by the three vertices of its triangle, and can run whatever calculations deemed necessary on these, in order to arrive to a final colour value.

Programmable shading allows a much greater range of BRDFs to be implemented in real-time applications. For instance, it is possible to include a tangent and binormal with the vertex data, along with the usual normal, in order to calculate reflectance on anisotropic surfaces, like brushed metal, more accurately. It is also possible to use texturing techniques to map arbitrary constants of a BRDF to points on a surface.

Common general-purpose BRDFs, other than Phong and Blinn-Phong, include Ward [58] and Cook-Torrence [19]. In addition there are several specialised models for certain materials, such as Kajiya-Kay [32] for rendering fur and hair.

2.5.3 Global illumination

One of the more difficult effects to take into account when shading is the lighting effect that diffuse light, emitted from one surface, has on the shading of other surfaces in the scene. In the models previously discussed, this is typically approximated as a single ambient lighting term that is constant for the entire scene. Models that attempt to take the scattering of ambient light into account are referred to as global illumination models.

Several global illumination models, implementable in a real-time application, exist. One approach is *radiosity transfer* [25] [4], where the diffuse light in the scene is allowed to "bounce" in multiple passes. The most common models in games are *ambient occlusion models*. In these, as a point is being shaded, one measures how much of the hemisphere of incoming light is occluded and thence reduces the global ambient light for that point accordingly. There are several models for ambient occlusion, such as the one used by Crytek in the game Crysis [48], and the more approximate unsharp mask method used by Luft et al. [39]

2.5.4 Shadows

An important element of shading is the handling of occluded light sources and the shadows they cast. There are a few major methods for shadows in modern rendering.

For a static scene and a static light source, it is possible to pre-calculate a light map, where the pixels and vertices of the scene are mapped onto their corresponding level of occlusion. This can be made arbitrarily accurate at the expense of texture memory, but as was just mentioned, it cannot handle moving lights or dynamic objects.

There are two major classes of algorithms for producing shadows in real-time for dynamic scenes. First is the shadow-map algorithm, originally by Williams [64], in which a surface is considered in shadow if it is further away from a light source than the closest object in its direction. It works in two passes. First, the scene is rendered from the perspective of the light and the z-values are saved to a buffer. When the scene is rendered normally in the second pass, the world coordinate of each rendered pixel is transformed in the same way it was in the first pass, and the resulting depth value is compared to that stored in the buffer. If the pixel to be shaded has a higher z-value, then it is occluded.

The main problems with shadow maps are the limited resolution and the fact that the texels (i.e., the texture pixels) of the shadow map will not map exactly to pixels in the final scene. Areas where there is a lack of resolution will appear as jagged blocks of pixels instead of the desired smooth-shadow edge. There are two primary approaches to lessen this. First, there are techniques that modify the projection used, such as *perspective shadow maps* [55] and trapezoidal shadow maps. These often require little extra work when sampling the shadow map, but generating the shadow map may be cumbersome, and the altered projection often means that shadow edges flicker when the observer moves. Complementary to projection-based techniques are filtering techniques, like *percentage closer filtering* [51] and *variance shadow maps* [22]. These use multiple samples in the second pass in order to compute some form of average, which may impact performance, but will produce relatively smooth and natural shadows.

The other major class of algorithm is *shadow volumes* [20] (also called *stencil shadows*). Here, a surface is considered in shadow if it is contained within a volume of shadow. The volume is calculated by generating polygons, stretching from the silhouette edges of any occluder as seen from the light source, and out to infinity. The silhouette edges visible to the observer are then counted for each pixel: a front-facing polygon increments the count, a back-facing polygon decreases it. Thus, a pixel is in shadow if its count differs from zero. The case where the observer is in shadow must be handled separately though. An efficient implementation of this technique using a stencil buffer is given by Heidmann [29]. An alternative approach is to count the silhouette edges away from a pixel and towards infinity instead.

Shadow-volume–based shadows are sharp for the entire scene, which is often unrealistic. They may also be quite expensive to rasterise, since in a complex scene a great number of shadow-volume edges may need to be generated and drawn.

2.5.5 Reflections

Reflections are paramount effects that provide the viewer with depth and a sense of space in a realistically rendered scene. Unfortunately, calculating accurate reflections from curved surfaces is a difficult matter.

For a flat surface, it is possible to generate a matrix transform that accomplishes the mirroring. Thus, a possible technique is to simply draw the reflected objects (and associated light sources) twice, first using the reflection transform, and then as normal, using the reflecting surface as a transparent window to the reflected world. Unfortunately, a simple matrix transform generating the reflected geometry for curved surfaces does not exist.

A common solution to reflections on curved surfaces is to use some form of ray tracing, to see what a ray cast in the direction of reflection will hit. It is still unfeasible to perform full ray tracing for the entire scene, and therefore *environment maps* are used to speed up the process. An environment map is a projection of the surrounding geometry, as seen from the reflecting object, onto some texture. The original environment-map algorithm, by Blinn and Newell, used a *sphere map* [11] for its projection, but this and other parabolic surfaces have largely been superceded by the *cube map* [27] in modern applications. Environment-map techniques are approximations, but it is often very difficult for the viewer to determine the difference between an accurate reflection and an approximation for a curved surface.

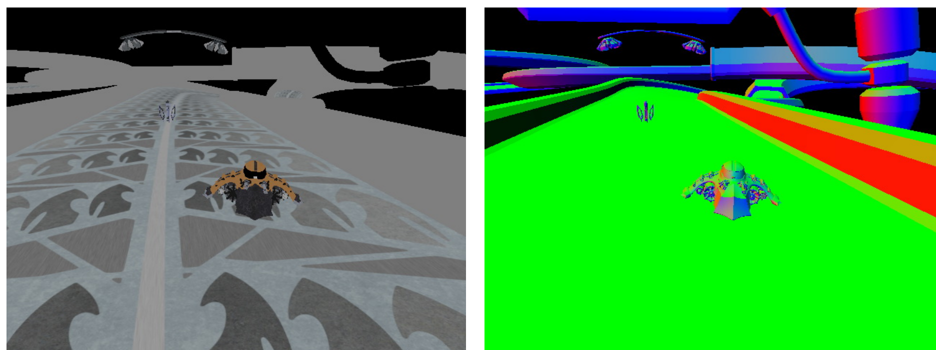
2.5.6 Deferred shading

In traditional forward rendering, one performs the full shading operation on the pixels of a primitive as rasterisation is being done. This can be implemented as either as single pass, in which case the operation is run

once for each pixel of the primitive, and information about all light sources affecting the primitive must be included in the draw call, or as multi-pass, in which case the primitive is drawn once for each light affecting it, and the results are added together.

Both these techniques can be problematic. In a Z-buffer or painter's-based algorithm, the primitive being drawn might later be covered by a different primitive, thus making the previous calculation superfluous. For single-pass shading, it may be quite difficult to determine which lights can be said to affect a specific draw call, and harder still to implement specific shaders for all possible combinations of material and light. For multi-pass shading, one might end up either drawing several primitives for a light source that does not affect them, or doing costly calculations to determine which primitives are to be drawn for each light.

A solution to these issues is deferred shading, first introduced by Deering et al [21] and later expanded by Saito and Takahashi [56]. In a forward shader, shaded-colour data is drawn directly from primitives, while in a deferred-shading pipeline, a middle step, called a *geometry buffer* (or g-buffer), is first calculated. The g-buffer has the same dimensions as the final colour buffer, and for each pixel it stores the information required to calculate shading for that pixel, e.g., normal direction, constants of the Blinn-Phong model, etc (see Figure 4). The exact data stored in the g-buffer depends on the type of shading desired.



(a) A geometry buffer texture storing the diffuse colour of a pixel.

(b) A geometry buffer texture storing the surface normal of a pixel.

Figure 4: *Two applications of the geometry buffer.*

To create the final shaded image, a bounding volume is then drawn for each light. For each pixel in the bounding volume, an appropriate shader for that light type is computed, reading the material data from the g-buffer, calculating the current light's colour contribution and adding this contribution to the colour value in the final image.

Deferred shading has several advantages. Shading calculations for each

light are done precisely for the pixels affected by the light. An arbitrary number of lights can be used in a scene and typically good performance can be maintained with several hundred light sources, somewhat depending on how much of the scene is illuminated by each light.

There are, unfortunately, also a few drawbacks. Deferred shading techniques take a decent chunk of graphics memory to store the g-buffer and especially use a lot of memory bandwidth, as the buffer is continuously read in the draw call for each light. It is also difficult to handle transparency in a deferred shading pipeline; the material data for one pixel is stored in the buffer, but when transparency is involved, the colour of a pixel is determined by more than one material.

2.6 Method

At first, a very simple single-pass forward renderer based on the Phong shading model was developed, in which to build the gameplay systems. Later, the decision was taken to implement a more extensive multiple-pass renderer based on deferred shading. The deferred approach allowed for more complicated lighting to create a lively scene, admittedly at the cost of some effort, but resulting in much nicer effects.

The renderer itself is implemented as a state machine, where each state in this abstract renderer represents a set of states in the underlying DirectX rendering pipeline. The main modes of operation in the renderer, together with their function, are the following:

ShadowMap
: Draw geometry and render to a shadow map texture (illustrated in Figure 6).

Geometry
: Draw geometry and render to the geometry buffer.

NormalMapGeometry
: Draw normal mapped geometry to the geometry buffer (illustrated in Figure 4(b)).

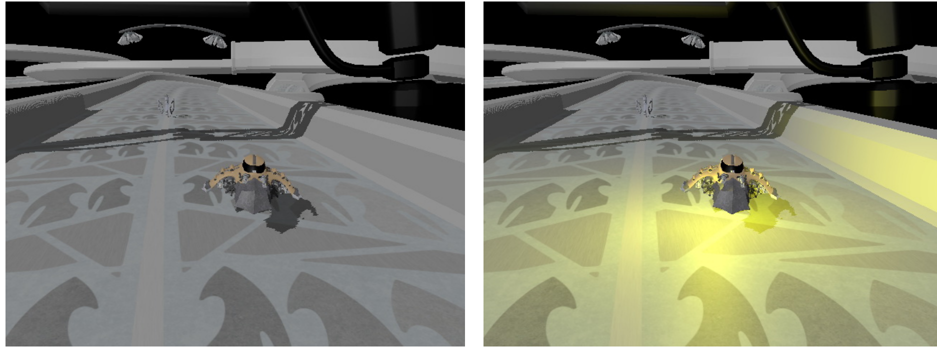
SceneLight
: Draw regions, render ambient and directional light contribution to the back buffer (illustrated in Figure 5(a)).

PointLight, SpotLight
: Draw bounding volumes for lights, render light contribution to back buffer (illustrated in Figure 5(b)).

Particles
: Draw particles to the back buffer.

PostProcessing

Apply a shader drawing from the backbuffer, to the backbuffer. This encompasses effects like motion blur and antialiasing.



(a) The scene after the Ambient and Directional shading pass.

(b) The scene after point lighting contributions have been added to the previous result.

Figure 5: *Lighting passes.*

The renderer contains a simple resource manager for its own resources, which essentially amounts to the geometry buffer and the various shaders used for the rendering modes outlined above.

2.6.1 Shading

Using complicated shading models was considered counterproductive, and the Blinn-Phong model was therefore used for the shading calculations, as it is simple to implement in a short time frame, has a number of easily available defined-constant sets for various materials and looks reasonably realistic while yielding excellent performance.

Initially, no bump- or normal maps were utilised. However, this gave tracks and other less detailed meshes a very unrealistic appearance, and support for normal mapping in the rendering engine was therefore added eventually.

During rendering, the properties that need to be used for shading each pixel in the g-buffer are stored. To minimise memory space and bandwidth used, the following 4 textures, in addition to the back buffer, are maintained:

- A 32 bit R8G8B8A8 2D texture. RGB encodes the pixel's diffuse-reflection colour in the Blinn-Phong model. A encodes a material index
- A 32 bit R10G10B10A2 2D texture. Here, RGB encodes the world space normal direction. The 2 bit alpha channel is unused.

- A 24 bit depth buffer. Full spatial coordinates are not stored and instead reconstructed from the texture coordinate and depth buffer value.
- Two 1D textures containing the material properties allowed to vary in the scene: specular colour, shininess, reflectivity and specular intensity in the Blinn-Phong model.

During pixel shading, relevant values are loaded from the g-buffer textures. The world space position, should it be needed, is created using a transform from the viewport coordinates and depth. This saves memory and bandwidth at each shading operation, at the expense of additional vector operations.

When applying the shading operation for point- and spot lights, only the relevant pixel shader for pixels that have some likelihood of being affected by the light should be run. To accomplish this, bounding volumes for the light volume are rendered, when rendering the contribution from a specific light. A pixel is only shaded by a light if the back-facing polygons of the bounding volume fail the z-test for that pixel and the front-facing polygons pass it. This is implemented with a stencil buffer, and is in essence identical to the method used with shadow volumes.

2.6.2 Global Illumination

Several global illumination techniques were considered for creating a more realistic image. Unfortunately, due to time constraints, only a few of the originally planned features were implemented. Shadows and reflections were considered the most important, as these components are central in creating a sense of depth to a rendered image. More advanced techniques, such as ambient occlusion, were deemed less critical.

For the shadows, a simple shadow-map–based solution (illustrated in Figure 6) was implemented. The shadow map was chosen because it is generally simpler to implement than a shadow-volume–based system. To simplify further, shadows are only drawn for the directional sun lighting in the scene, not for the point lights or spot lights. This can potentially look awkward with some lighting configurations, but the impact of this can be minimised by designing the track to ensure the sun is usually the strongest light present.

A relatively simple static cube map is used for all reflections. A dynamic reflection system was briefly considered, where a map would be periodically drawn for each ship and updated with respect to the current environment. However, the idea was dismissed, once again due to time constraints.

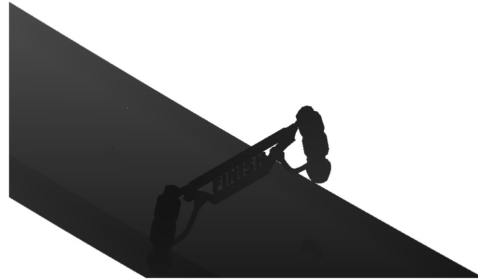


Figure 6: *The depth texture used in the shadow map algorithm for directional lighting.*

2.7 Future work and discussion

The list of techniques we wanted to implement, but had to abstain from due to lack of time, is unfortunately large.

Several features present in the initial plan for the graphics engine were cut due to time constraints and would be useful avenues for further work on the project. Notably, screen-space ambient occlusion would enhance the image quality at reasonably little effort. Improving the shadow-map fidelity by using filtering or cascading shadow maps would also offer a marked improvement, as the current shadows suffer from aliasing issues.

More advanced radiosity techniques were deemed too complicated to be included in the project given the time limit. We initially intended to feature ambient-occlusion techniques, as these are generally suited for implementation in a deferred-shading pipeline, and code was even written in that purpose in the ambient-lighting shader. But again, it was decided that other parts of the project were more critical, and occlusion techniques never made it to the final program.

Other planned features that fell victim to our time- and labour constraints were an edge-detection–based anti-aliasing shader in the post-processing stage as well as using dynamic reflections.

Aliasing issues also crop up in the final renderer, and implementing an anti-aliasing scheme in post-processing would greatly enhance image quality.

3 Particle system

3.1 Introduction

A particle system is a set of separate small objects that are set into motion using some algorithm. Particle systems are used in computer graphics to simulate phenomena otherwise hard to reproduce with conventional rendering techniques [34]. Examples of such phenomena include fire, explosions, smoke, water flows, sparks, fog and abstract visual effects like glowing trails. They can also be used for rendering purposes, for examples to simulate the heterogeneities on the surface of a tree, with more particles being generated and displayed the closer the camera gets.

3.2 Method

The particle system differentiates between two types of particles: emitters and flares. The main difference between these is that only the flare particles are actually rendered. The emitter particles, as the name suggests, do spawn new particles, but these are not displayed; their positions are used to create light sources in the rendering engine. Flare particles are, however, continuously rendered until destroyed by the particle system.

A particle is modelled as a single point, which then is rendered as a single pixel. This might appear to confine the flexibility of the particle system, for instance when it comes to changing size and map texture. There are simple techniques to resolve this, for example by stretching the particles to lines or quads (i.e., quadrilaterals made from two polygons). In the latter option a texture can then be added to the quad, transforming it into a *billboard* and finally turning it to face the camera, as is illustrated in Figure 7. This technique is known as *billboarding* and is commonly used to simulate particle systems and low-detail vegetation [42].

Once the type, position and size of the particles are known, their velocity and life span remain to be defined. Together, these five parameters form the foundation of the particle system.

In order to obtain a sense of authenticity, some sort of randomness in the creation and movement of the particles is needed. The phenomena intended to be reproduced in this specific game, i.e., fire, explosions, smoke and sparks, do not follow any visible patterns in real life. Unfortunately, there are no random number generators inside the shaders, and thus randomness had to be computed. The solution consisted in generating random numbers with the CPU, by using the current game time, combined with offset values, as variables. These values are then put together to form vectors, which in turn form textures that are sent to the shader.

A particle system has two distinct stages, the update / simulation stage and the rendering stage. During the simulation stage, new particles are cre-

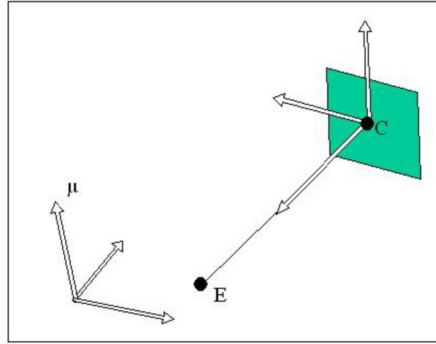


Figure 7: *If we know the world coordinate of the vector μ, the center position C (originally the particle's position) and the eye position E (the camera's position) we can describe the billboard's local frame in world coordinates.*

ated according to the emitter particles' properties and the interval between the updates, and existing particles either have their positions updated or are removed. The rendering stage then displays the appropriate particles.

3.3 Results

A CPU-based particle system is relatively easy to implement but is limited due to the communication between the CPU and the GPU. Since all simulations are done on the CPU, all particle data has to be sent to the GPU at each frame, and on a standard personal computer this limits the amount of particles to approximately 10000 per frame, since the particle system shares the GPU bandwidth with several other rendering tasks.

The alternative was to use a GPU-based implementation, where simulation, as well as particle creation and deletion, are instead performed directly on the graphics card.

In earlier implementations of GPU-based particle systems, stateless systems using vertex shaders were used, but since they did not store the current position of particles it was difficult to have them react with a dynamic environment. These implementations were therefore only suited for small and simple effects. With the introduction of stream programming in D3D 10, GPU-based particle systems can be made more advanced thanks to stream output.

The concept of stream output is to use two buffers. One will serve as an input buffer for the updated particles, and the other as an output buffer for rendering. At each frame the buffers will be interchanged, and the stored particles will be drawn [44]. It is an efficient way to simulate particle systems, and two examples of the results can be seen in Figure 8.

(a) Explosion. (b) Fire.

Figure 8: *Effects produced with particle systems.*

3.4 Discussion

Our particle system is still very primitive. The possibilities are copious, but again, due to time issues, we did not manage to develop more than basic explosions and jet flames. Smoke, magnetic fields, advanced explosions, galaxies and sparks are just a few of the examples we discussed in the very beginning of the project. Moreover, the particle systems we did implement can be substantially improved and embellished, by for example reducing the billboards and adding colours.

4 Modelling

4.1 Introduction

One of the key elements in the creation of a realistic three-dimensional game is the modelling of the graphical components. Along with the constant expansion of computer power come possibilities to handle more and more data, which takes graphical modelling to a whole new level. With an increasing number of polys (areas) per shape, the wealth of details grows, and as a result the credibility of the game as well. The computer development also helps improving the software used for modelling.

An example of this expansion is the "Utah teapot" (Figure 9), first modelled in 1975 by Martin Newell, one Bezier control point at a time (further details in subsequent sections). Since then, the Utah teapot often plays the role of archetypical model, and it still widely used as a reference in a lot of modelling softwares and is well-known in the world of computer graphics [57].

Figure 9: *The Utah Teapot.*

All modelling was done with 3ds Max (formerly known as 3D Studio MAX), a modelling, animation and rendering package developed by Autodesk Media and Entertainment [6].

4.2 Method

Modern 3D graphics modelling can be divided into three main techniques: polygonal modelling, splines and sculpt modelling.

4.2.1 Polygonal modelling

In polygonal modelling, surfaces are approximated with the help of vertices and edges. These form the *polygonal meshes*, and polygonal meshes constituted by four vertices connected by four edges form *quads*, which are the geo-

metrical shapes most commonly used in modelling. Polygonal modelling is a suitable technique when modelling a drawing. Basically, a two-dimensional map is constructed, extracting the edges one at a time and adjusting the vertices conforming to the drawing, as is pictured in Figure 10. When repeating the procedure along every perspective, a very representative model of the original drawing is obtained.

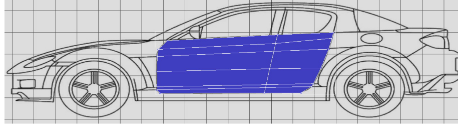


Figure 10: *Polygonal modelling.*

4.2.2 Splines modelling

With the splines technique, surfaces are defined by lines and curves. These are formed by connecting points, and together they constitute the base for further modelling [60]. Each point's characteristics define these curves' behaviour, as well as the amount of control the modeller has over the shapes. Normal points, akin to vertices in the polygonal-modelling technique, are referred to as *corners*. However, to construct smoother connections and to be able to simulate volumes, one makes use of so-called *Bezier points* [23], points with which one can alter with the tangents and thereby enabling modification of the interpolation between the points, as is shown in Figure 11 .

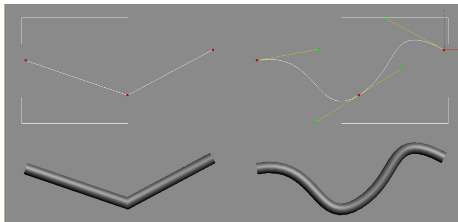


Figure 11: *Splines modelling.*

4.2.3 Sculpt modelling

The sculpt modelling is still a relatively new technique, where the software lets the user "sculpt" the model with the help of different tools, analogously to clay modelling. One works with a surface (a *mesh*) of polygons, that can be modified unrestrainedly. This technique enables the detail perfecting to

the point that the results appear photorealistic, as is illustrated in Figure 12, something that would be extremely strenuous with the aforementioned methods. Sculpt modelling also allows working in layers, starting off with a low-resolution model and continuing building upon it. This is a valuable asset when implementing normal mapping, since one can work on models with relatively few polygons but with still a high level of details. Normal mapping is explained more profoundly in Section 4.4.3.

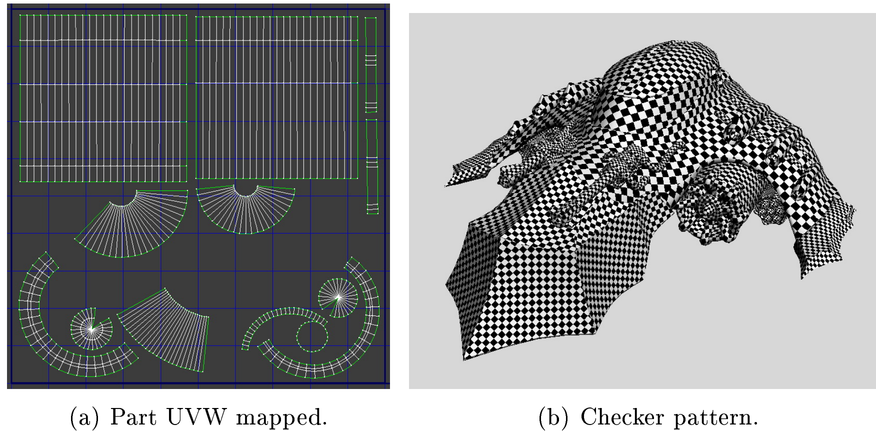
Figure 12: *Sculpt modelling (image by ISEETRUTH, courtesy of Wikipedia http://en.wikipedia.org/wiki/File:Zbrush.PNG).*

4.3 UVW mapping

Once the model is finished, one needs to define how it is to be enveloped by the accompanying texture. In this aim, *UVW mapping* is a suitable technique. It consists in converting a two-dimensional image (a texture) to a three-dimensional object of a given topology.

Each point in a UVW map corresponds to a point on the surface of the object, with coordinates u and v, while the third coordinate w is only used as a height coordinate for complex volumetric textures. The object's surface is decomposed into smaller parts (like on Figure 13(a)), to avoid that the texture spreads out improperly, and the points on the texture are then assigned to XYZ-coordinates on the target surface, according to how the graphical designer decides to implement the map. Once the texture is finished, it is simply wrapped around the object, by applying the correct colour to the corresponding pixel.

To check that the decomposition is correctly done, one can apply a checker pattern to the object, since it clearly shows possible inaccuracies. Figure 13(b) shows an adequately applied texture.



(a) Part UVW mapped. (b) Checker pattern.

Figure 13: *UVW mapping technique.*

4.4 Texture mapping

Having model information limited to the vertices of a mesh and more detailed data interpolated between the vertices is severely limiting. A method for providing additional detail on exactly how the surface properties vary within a polygon is texture mapping, first described by Catmull [15]. Here, each vertex is mapped to a texture coordinate. To obtain the value of a surface property at a given point, the texture coordinate is interpolated from the vertices, and loads the texture data. Texturing is chiefly used to denote the colour of a point, but any surface or volume information can in principle be stored using a texture.

4.4.1 Diffuse map

Diffuse map is the most basic texture. It simply wraps the bitmap image onto the three-dimensional geometry surface, while displaying its original pixel colour. Specific software can also be used to create pre-rendered texture effects such as shadows, in particular the ones engendered by the ambient lighting (the ambient-occlusion method described earlier), an approach that saves a lot of system resources and rendering time, since it spares the renderer some burdensome calculations. An example of standard diffuse mapping can be seen in Figure 14.

4.4.2 Specular map

A specular map is a bitmap that defines the shininess of an object or a part of an object. The lighter the pixel, the shinier it appears when rendered.

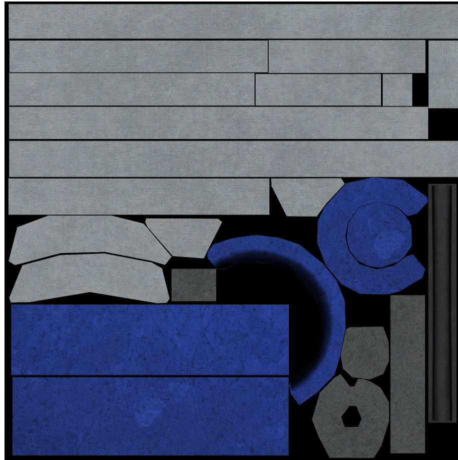


Figure 14: *Diffuse mapping.*

4.4.3 Bump map

One possible use for texturing is to create the appearance of additional geometric detail without adding vertices. The initial method here is known as *bump mapping*, where texture values represent height perturbations from an average, an algorithm first described by Blinn [10]. In modern applications, it is more common to store the surface normal in a texture, known as *normal mapping*. A normal map makes use of the RGB channels to represent the normals, where red, blue and green correspond to the x-, y- and z-coordinates, respectively, of the surface normal. The obtained bitmap (like the one in Figure 15) is then used to calculate the adequate shading, depending on the origin of the light source, the results of which can be seen in Figure 16.

Such mappings are commonly generated from a highly detailed mesh, in order to make a mesh using fewer polygons, a technique first described by Krishnamurthy and Levoy [33], later expanded by Cohen et al [18] and Cignoni et al [17].

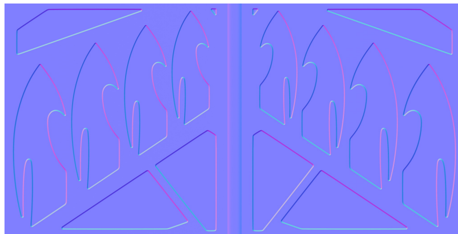
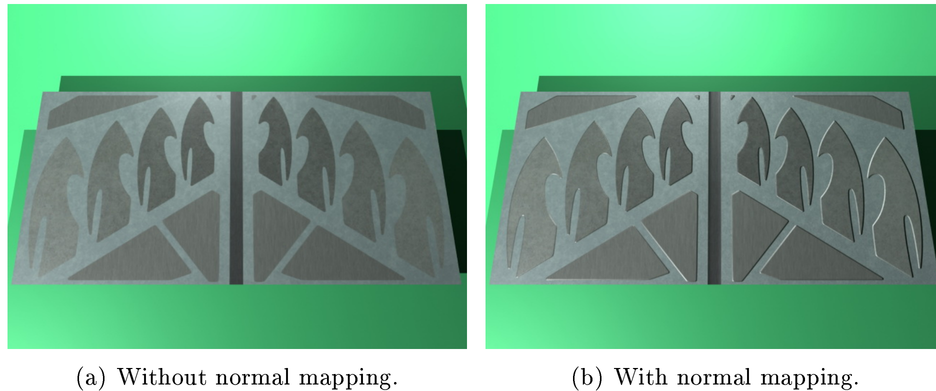


Figure 15: *A normal map.*



(a) Without normal mapping. (b) With normal mapping.

Figure 16: *The effect of normal mapping.*

4.5 Model import

Once the objects are modelled and textured, they need to be imported into the game. It is convenient, if not necessary, to use a model importing software for that purpose.

A model importer loads the object and its associated files and returns an *aiScene structure*, containing the essential information about the model, such as textures, meshes, materials and animations. The IDE then inserts all vertices, indices, UVW-coordinates, etc, into a mesh-list, such as the one depicted in Figure 17, which is then sent to the renderer.

1 meshes found in Media\FlightTest.obj	
Mesh 0 with 24 vertices and 12 triangles	
Diffuse:	<0.5882, 0.6078, 1>
Specular:	<0, 0, 0>
Ambient:	<0.5882, 0.6078, 1>
Emissive:	<0, 0, 0>
Transparent:	<0, 0, 0>
Shininess:	40
Shininess strength:	1
Opacity:	1
Refraction:	1.5
Wireframe enabled:	0
Twosided mesh:	0

Figure 17: *A mesh list.*

4.6 Results and discussion

The model importing software we used, Open Asset Import Library (or simply *Assimp*), did, despite its ability to load over 30 different file types [5] and its user-friendly design (see Figure 18), have a few drawbacks. It was, for instance, unable to handle meshes to which more than one material were

assigned, unless dividing them into submeshes (this due to a lack of memory). The results were that models originally composed of 16 meshes ended up having more than 300, which in turn led to decreasing performance and even occasional crashes.

Also, Assimp seemed very sensitive to the number of vertices contained in meshes, and the time required to import models increased quickly as that number grew.

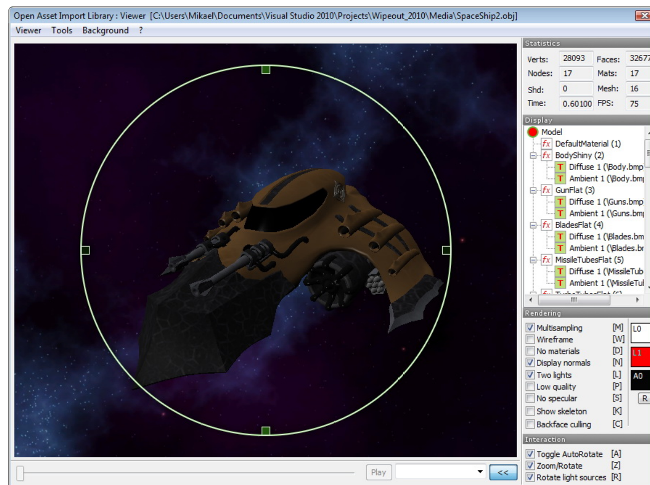


Figure 18: *A snapshot of Assimp Viewer.*

3ds Max turned out to be a suitable modelling software. Its flexible plugin architecture and user-friendly interface, together with the ample availability of tutorials online, made it an effective tool for a Microsoft-oriented programming environment.

Throughout the project, our modelling technique of choice was polygonal modelling, since it suited our primary objects, i.e., space vessels. We did also have recourse to the splines technique, in particular when modelling the track. However, the sculpt technique was not relevant for this specific game, since close-up details were not a priority. In addition, it often requires additional hardware, such as drawing boards, which we neither had access to nor ever even contemplated.

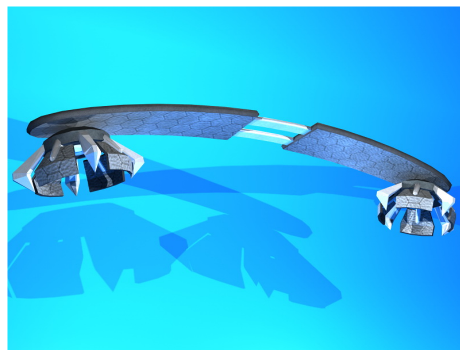
We spent a lot of time in the beginning of the project learning the basics of modelling. None of us had any extensive experience of it, and as a result a lot of the early modelling was done clumsily and even erroneously. We noticed, for instance, that models suited for animations may not at all be suited for computer games, as the settings differ a lot. UVW mapping, size of polygon meshes and texture characteristics are all factors that need to be taken into account when deciding whether or not the model is intended for

a computer game. Which, unfortunately, took us long to realise.

A lot of improvements can be done in the future. We only managed to implement but a fraction of the plausible modelling techniques. Still, the results of these are satisfactory, as can be seen in Figures 19 to 21.

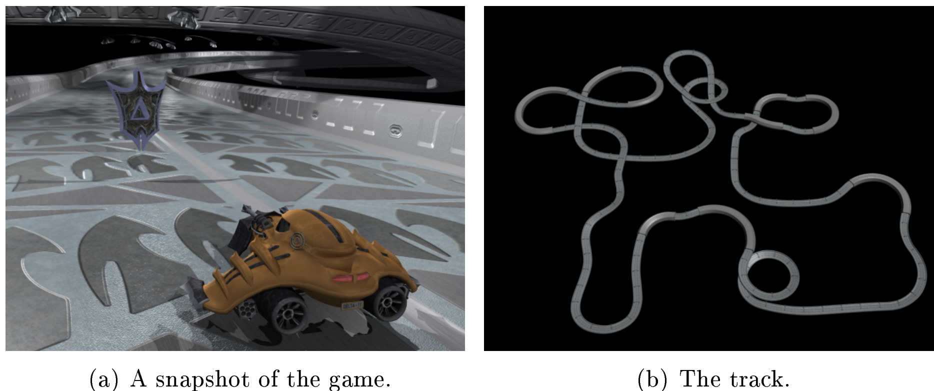
Figure 19: *Our space ship.*

(a) Finish line



(b) Hover light

Figure 20: *Examples of track features.*



(a) A snapshot of the game. (b) The track.

Figure 21: *More models.*

5 Physics

One of the major issues of a computer game simulating real-world scenes is the approximation of physical reality. In order to do so efficiently, one often makes use of a *physics engine* to simulate physical systems, such as rigid-body dynamics, soft-body dynamics and fluid dynamics. Considering the short amount of time we disposed to create the game, we focused only on rigid-body-dynamics systems. It resulted in effects that were realistic enough for a fast-moving racing game like ours.

Rigid-body dynamics allows movements in three dimensions and can simulate important physical properties, such as center of mass and moments of inertia, that are fundamental in the creation of realistic effects. However, rigid-body dynamics does not enable the deformation of objects, which implies that the credibility of our simulated collisions is limited.

5.1 Method

Collision detection is a paramount element in many computer games and in computer-graphics applications in general. It is a part of the more global *collision handling*, comprising *collision detection*, *collision determination* and *collision response*. The three parts' respective aims are quite straightforward. Collision detection returns a boolean telling whether or not two objects collide, collision determination calculates the actual intersections between objects and collision response determines what actions are to be taken as a result of the collision.

It was decided early in the development process that an open-source physics engine would be used for the collision handling, as writing one from scratch would have implied an additional couple of weeks of programming work. After examining a few different ones, *Bullet Physics Engine* (Figure 22) was opted for, since it was well documented and was used in a variety

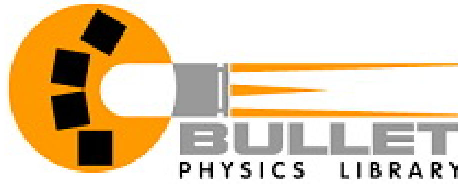


Figure 22: *The Bullet Physics logo.*

of successful movies and games, such as *Toy Story 3*, *Grand Theft Auto IV*, *2012* and *Hancock*, to name a few.

5.1.1 Architecture

Bullet is published under the *zlib* license, a permissive free software license. It features 3D collision detection, soft-body dynamics and rigid-body dynamics.

The collision detection provides algorithms and acceleration structures for closest point queries, as well as ray and convex sweep tests. It is constructed following a hierarchical class architecture, as Figure 23 illustrates.

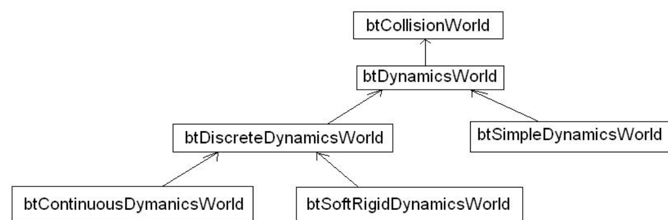


Figure 23: *btCollisionWorld.*

We only made use of *btDiscreteDynamicsWorld*, since it provides more general methods than the basic *btSimpleDynamicsWorld*.

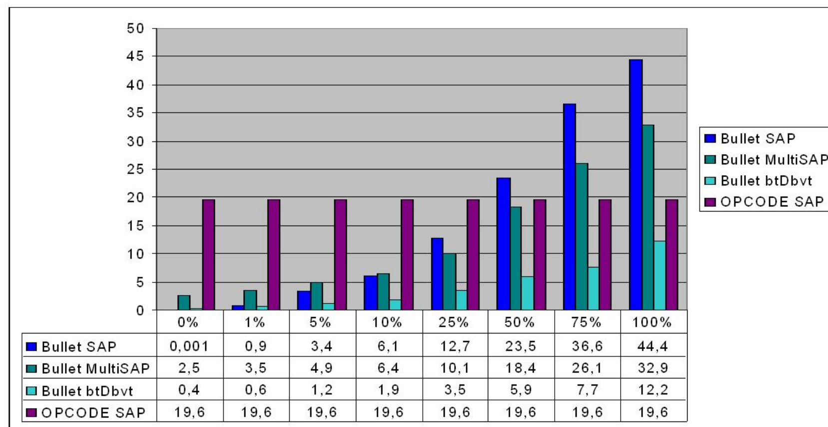
Within that class is a set of crucial elements, that constitute the core of the collision detection algorithm:

- The *broadphase*, a class of algorithms that quickly detects object pairs that might overlap, i.e., collide
- The *collision configuration*, that allows the user to fine-tune the collision detection algorithms

- The *dispatcher*, a customisable mask that instructs Bullet to ignore specific object pairs, deemed unnecessary to flag and cross-test
- The *solver*, that performs the last steps in the collision handling, i.e., computes the collision determination calculations and causes the objects to interact properly, taking into account gravity, game logic supplied forces, collisions and hinge constraints.

5.1.2 Algorithms

One of the features that makes Bullet such an efficient collision-detection software is the first entry in the above list. The broadphase contains algorithms that use the bounding boxes of objects in the world to quickly compute an approximate list of colliding pairs. The list will include every pair of objects that are colliding, but may also include pairs of objects whose bounding boxes intersect but are still not close enough to collide. These extra pairs will then be eliminated before reaching the solver stage, where the exact collisions are calculated, which will improve the overall performance greatly since the associated algorithms are very time-consuming.



	0%	1%	5%	10%	25%	50%	75%	100%
Bullet SAP	0,001	0,9	3,4	6,1	12,7	23,5	36,6	44,4
Bullet MultiSAP	2,5	3,5	4,9	6,4	10,1	18,4	26,1	32,9
Bullet btDbvt	0,4	0,6	1,2	1,9	3,5	5,9	7,7	12,2
OPCODE SAP	19,6	19,6	19,6	19,6	19,6	19,6	19,6	19,6

Figure 24: *Comparison of different broadphase algorithms.*

There are several kinds of broadphase algorithms to choose from when instantiating Bullet [13], all of which substantially improving the $O(n^2)$ quadratic-time complexity implied by cross-testing every object against one another (n representing the number of objects) [24]. The two most commonly used are the dynamic AABB tree algorithm (implemented with *btDbvtBroadphase*) and the sweep-and-prune algorithm (or SAP, implemented with the *btAxisSweep* range of classes).

The AABB tree [52] broadphase adapts dynamically to the dimensions of the world and its contents. It is well optimised and constitutes a good

general-purpose broadphase, particularly suitable for handling dynamic worlds where many objects are in motion, since it has very fast insertion, removal and update-of-nodes operations.

The sweep-and prune algorithm [37] is also a good general-purpose broadphase, with the restraint that it requires a fixed world size, known in advance. This broadphase has the best performance for typical dynamic worlds, where most objects have little or no motion.

The benchmark in Figure 24 shows tests run for different broadphase algorithms, where the abscissas represent the timings of the collision-detection calculations in milliseconds, given a grid of 8192 boxes, and the ordinates are the percentage of the boxes that are moving. We clearly see that the AAPP is preferable for a very dynamic environment, whereas if only a few of the boxes are in motion, SAP is the algorithm of choice. The tests are run by Bullet Physics.

The SAP algorithm was therefore chosen, since its preferences cohered with the game, the latter consisting of only a few dynamic objects, set in a predefined world.

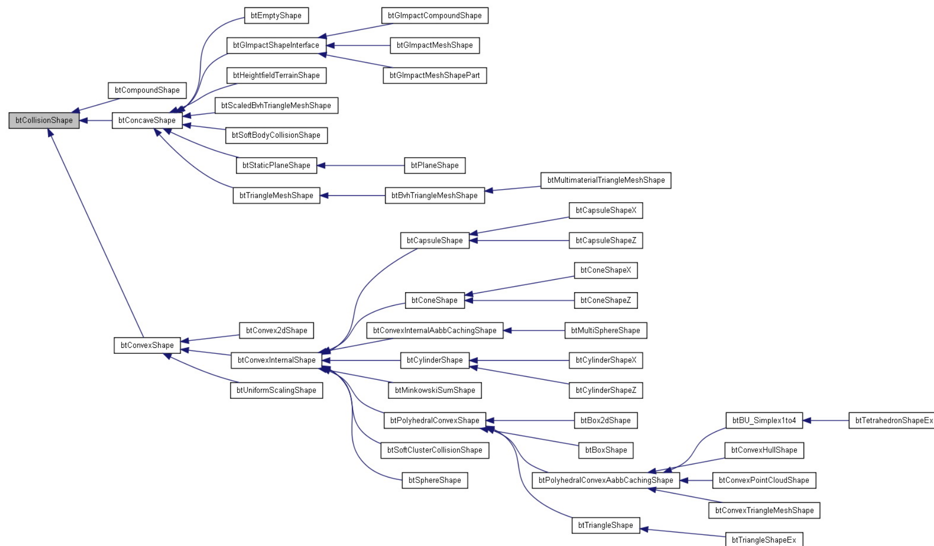


Figure 25: *Bullet's variety of collision shapes (Picture courtesy of Bullet Physics. http://www.bulletphysics.com/Bullet/BulletFull/ classbtCollisionShape.html).*

5.1.3 Collision shapes

Bullet supports a large variety of collision shapes, as can be seen in Figure 25, and it is important for the sake of game performance, accuracy of collision

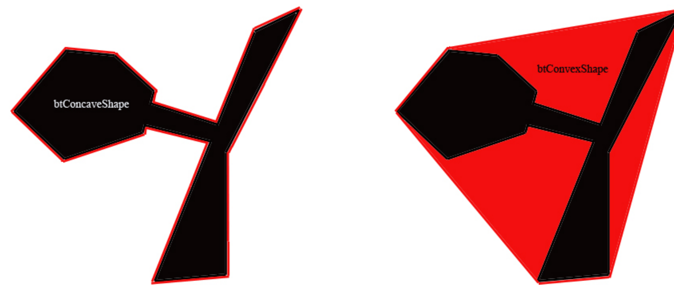
and quality of the simulation, to select the most suitable ones.

There are three categories of shapes: *btConcaveShape*, *btConvexShape* and *btCompoundShape*.

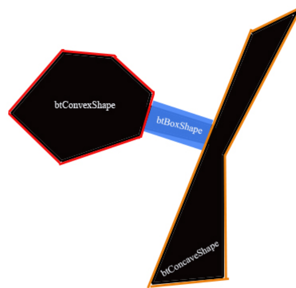
The *btConcaveShape* is loaded as a polygon mesh and fits the object, as is illustrated in Figure 26(a). It is a good alternative for static objects, since they are likely to involve less collisions and therefore can be enclosed with more precision without affecting the performance.

The *btConvexShape* is also loaded as a polygon mesh, but as opposed to *btConcaveShape*, it encloses the outer volume of the object, as is illustrated in Figure 26(b). It is therefore a good alternative for dynamic objects, since it does not require as many calculations as a concave shape. Moreover, Bullet offers a collection of built-in primitive shapes, such as spheres, boxes and rectangles, to simplify the process further.

The more avanced *btCompoundShape* allows to store combinations of convex and concave objects, that, when used properly, results in precise and fast collision handling. It is illustrated in 26(c).



(a) btConcaveShape. (b) btConvexShape.



(c) btCompoundShape.

Figure 26: *Object enclosing with Bullet.*

The Bullet manual provides a useful decision map (see Figure 27) that

helps the novice programmer determine which options to go for.

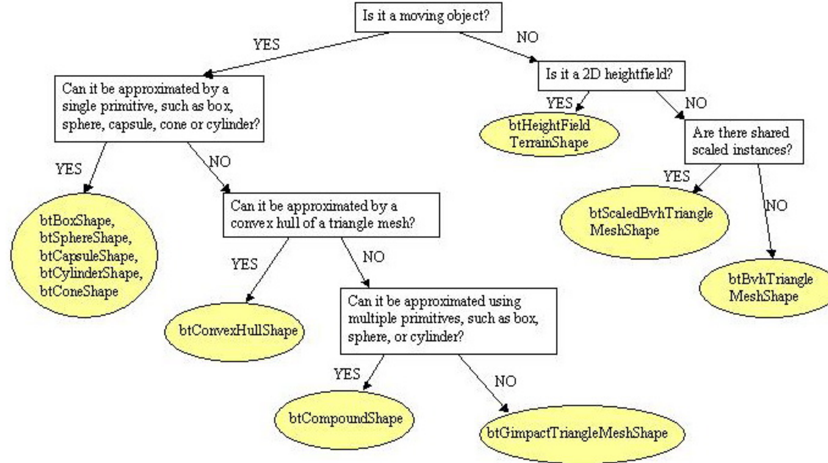


Figure 27: *Bullet's decision map for collision shapes.*

The ship is thus represented as a *btConvexHullShape*, since the simpler option *btCompoundShape* jeopardised the accuracy, while the more advanced option *btConvexTriangleMesh* performed worse.

Regarding the track, *btBvhTriangleMeshShape* emerged as the obvious pick, the height map alternative being conceived for environments with uneven terrain.

Once the collison shapes were defined and loaded, what remained to be done was to attach them to the actual rigid bodies, i.e., the models. They are the real physical units in the dynamics world, and physical concepts such as forces, mass, inertia and velocity will have impact on their position and direction.

Finally, these rigid bodies were added to the world and the instantiating of the dynamic world was completed.

5.1.4 Ship motion

In order to obtain a high level of credibility, it is essential to simulate the ship's behaviour properly, i.e., according to real-world space vessels. In computer graphics involving flying vehicles, the maneuvers are usually arranged into three rotations around perpendicular axes: yaw, roll and pitch. These are illustrated in Figure 28.

There are several ways to determine how each rotation shall be computed, and a lot of factors can be put into consideration, such as velocity, inertia, land relief and vehicle type.

Figure 28: *The yaw, pitch and roll rotations.*

The chosen technique consists in keeping the yaw-axis parallel to the surface normal of the track underneath. The cross product of the yaw axis and the surface normal is computed at every frame, and when it exceeds a predefined threshold value, the angle of the yaw axis is calculated anew and adjusted accordingly. As a result, both the roll and pitch axes are kept steady.

5.2 Results and discussion

The collision detection works flawlessly, but it is unfortunately very basic. The process of simulating a ship floating over a track in a realistic way turned out to be full of unexpected problems, and the method we opted for in the end was not one we were satisfied with, rather the one we were least displeased with.

Our first attempt consisted in approximating the ship as a cube, and have it slide on the track. However, that seemingly logic idea resulted in wobbly behaviour. The second try was based on a Bullet tutorial, involving the simulation of a car driving on a road. That approach did work, but the outcome appeared stiff and unnatural for a space vessel. Besides, modelling a space ship using a car-collision-handling mold felt very, very cheap, no matter how far beyond schedule we were.

Eventually we opted for the technique described above, which was, in fact, but a shortcut of a more advanced idea we had in mind, consisting in reproducing the floating effect of a lunar lander. We experimented with it, by generating rays at each corner of the ship, but eventually, as we added motion to it, the approximation of physical elements, such as forces and inertia, became too complicated. The trick we had recourse to resulted in a collision handling that is decent, but sadly not realistic enough for the space environment the game is set in and the laws of physics derived therefrom.

The lunar-lander idea is definitely an interesting scheme to work further

on in the future.

6 Sound and music

6.1 Background

Modern computer-graphics techniques are able to render scenes from the real world in a very realistic way, but without the help of sound effects and music, it is hard, if not impossible, to bring to pass the proper atmosphere. Small but distinctive sounds are the key to convey the exact feeling one wishes to compose. To spawn a lifelike virtual world, it is paramount to create the illusion that the player actually is inside that world. Sound effects amplify what the user feels when playing the game and thus generates this illusion.

Despite that this game is not the type of game where atmosphere and story are key elements, sound effects are important notwithstanding. Associating, for example, sparks, accelerations and collisions to specific sound effects will make the virtual world seem more real and will thus enhance the playing experience remarkably.

6.2 Method

The implementation was done with Microsoft DirectX 11, which, like earlier versions of DirectX, uses a sound system called DirectSound. However, one remarkable upgrade with DirectX 11 is that it has an integrated high-level library called XACT, which stands for Cross-platform Audio Creation Tool. It can also be used to develop sound systems for XBOX [46].

When implementing the sound engine, a lot of the work consisted in reading and understanding the API, after which it was crucial to implement the classes and functions so that they would be easy to merge with the main program.

XACT supports organising sound and music files in libraries. This is done by creating *wave-banks*. A wave-bank is a way to sort these files and to distribute them in a package. This can be done by either saving all the data on the internal memory (thereby creating a so-called *in-memory wave-bank*), or by streaming, which implies reading the data from a buffer. The primary disadvantage with the latter is that the playback needs to be prepared in advance to avoid delays, since the sound does not exist in the primary memory. The obvious advantage is that one saves a lot of memory.

6.3 Results

We opted for XACT, since it seemed relatively easy. Moreover, the cross-platform property allows us to use the same implementation, should we want to implement the game on other platforms than DirectX in the future.

For reasons discussed above, we decided to use the in-memory technique for sound effects and the streaming technique for music files, the latter being

in general a lot larger then the former.

One other asset with XACT is that it creates an interface between the programmer and the sound designer, which basically implies that the programmer need not know about sound design, and the sound designer need not know about programming, which could be of great use in a project larger than this one. This interface enables the sound designer to create variables that the programmer can make use of and update, such as distances, directions and revolutions per minutes. These variables are then updated in real-time during the game, creating sound effects accordingly. They can be global variables or instance variables, the main difference being that an instance variable is associated to and can only affect a specific sound object (called *cue*), whereas global variables can affect all sound objects.

6.4 Discussion

The choice of using XACT as a sound system was well justified. Its functionality and user-friendly design were major assets to us since we were on a tight schedule. We did however not get to use what could be considered as XACT's major asset, i.e., the interface between the programmer and the sound designer.

One of the plausible post-thesis improvements involves memory optimisation. In the current implementation, we are working with WAV files for both sound and music, since it seemed a lot easier. However, should the game expand with more sound effects and different themes, a format such as MP3 would be preferable, since it requires only about a tenth of the memory of the corresponding WAV file.

Also, had time not been an issue, we would have liked to add more advanced effects, such as echo in tunnels. We also discussed the use of so-called 3D sounds, with which one can simulate whence the sound originates, the speed of the object engendering the sound and the location of it. This was, for the godzillionth time, left as a future endeavour.

7 Conclusion

7.1 Results

The final product is, unfortunately, not as elaborate as was intended. Some parts are very satisfactory, such as the modelling, while some are less, the list of which grows longer. The utilised rendering techniques are efficient, easy to implement and resulted in visually appealing effects, which was indeed the purpose. However, the renderer can still be improved further by implementing more techniques, and had it not been for the never-ending problems with program coordination, code merging and model importing, the results would most likely have been a lot more pleasing.

7.2 Discussion

The outcome of this project is definitely not the outcome we foresaw four months ago.

One of our first mistakes was the decision to abstain from using the XNA framework, and instead to build our own using lower-level code. We were never able to translate the potential performance gains of optimising our code into actual quality gains. The time span we were given was simply not enough. There is no doubt that XNA's set of game asset pipeline management tools [43] [47] would have been of great help, and would most likely have enabled us to produce a more advanced and interesting end product.

When we started working on this project we did not quite conceive the difficulties involved in such an ample and vast assignment. The sole fact that our group consisted of seven people from four different programs should have been enough of a hint that some sort of software development methodology would be essential. It did, however, not and the consequences of that mistake would end up causing coordination issues and, alas, group conflicts. With hindsight, a methodology such as Agile Development [7] would presumably have resulted in propitious effects.

Moreover, the lack of organisation and teamwork had repercussions on the program itself. Many hours were wasted on synchronising, coordinating and adapting pieces of individually written code, and many times it had to be re-edited almost entirely.

7.3 Future work

There are a lot of features we still need to work on. The game design implementation is barely started, and the intended AI feature was never even brought up during the course of the project. Further, the sound engine only reached the very first step of its development, and a lot of effects still need to be conceived.

A multiplayer option, through networking or splitscreen, is yet another feature that would enhance the gaming experience substantially, as well as more tracks, ships and power-ups.

References

[1] Akenine-Möller, T., and Haines, E. (2002) *Real-time Rendering.* Second Edition. Natick, MA: A K Peters.

[2] Angel, E. (2003) *Interactive Computer Graphics: a Top-Down Approach With OpenGL.* Third Edition. Reading, MA: Addison-Wesley.

[3] Appel, A. (1968) *Some techniques for shading machine renderings of solids.* Proceedings of the Spring Joint Computer Conference.

[4] Ashdown, I. (1994) *Radiosity: A Programmer's Perspective.* John Wiley & Sons, Inc.

[5] Assimp Development Team (2009) Open Asset Import Library Features. *Sourceforge.*
`http://assimp.sourceforge.net/main_features.html`
(5 May 2011).

[6] Autodesk (2011) 3ds Max - 3D Modeling, Animation and Rendering Software. *Autodesk Inc.*
`http://usa.autodesk.com/3ds-max/` (8 May 2011).

[7] Beck, K. et al. (2001) Manifesto for Agile Development. *Agile Alliance.*
`http://agilemanifesto.org/` (7 Apr. 2011).

[8] Bikker, J. (1999) The Coverage Buffer. *Flipcode.*
`http://www.flipcode.com/archives/The_Coverage_Buffer_C-Buffer.shtml`. (16 Apr. 2011)

[9] Blinn, J.F. (1977) *Models of light reflection for computer synthesised pictures.* Proc. 4th annual conference on computer graphics and interactive techniques.

[10] Blinn, J.F. (1978) *Simulation of wrinkled surfaces.* Proceedings of SIGGRAPH, vol. 12, no. 3.

[11] Blinn, J.F., Newell, M. (1976) *Texture and reflection in computer generated images.* Communications of the ACM, vol. 19 .

[12] Bouknight, W. J. (1970) *A procedure for generation of three-dimensional half-tone computer graphics presentations.* Communications of the ACM.

[13] Bullet Physics Wiki (2010) Broadphase. *Bullet Physics.*
`http://www.bulletphysics.org/mediawiki-1.5.8/index.php?title=Broadphase` (13 May 2011).

[14] Caron, F. (2008) Gaming expected to be a $68 billion business by 2012. *Ars Technica.* `http://arstechnica.com/gaming/news/2008/06/gaming-expected-to-be-a-68-billion-business-by-2012.ars` (7 Apr. 2011).

[15] Catmull, E. (1974) *A subdivision algorithm for computer display of curved surfaces.* PhD thesis.

[16] Catmull, E. (1975) Computer Display of Curves Surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures.* Los Angeles.

[17] Cignoni et al. (1998) *A general method for recovering attribute values on simplifed meshes.* IEEE Visualization.

[18] Cohen et al. (1998) *Appearance-Preserving Simplification.* Proceedings of SIGGRAPH.

[19] Cook, R., Torrence, K. (1982) *A reflectance model for computer graphics.* ACM Transactions on Graphics, vol. 1, no. 1, pp. 7-24.

[20] Crow, F.C. (1977) *Shadow algorithms for computer graphics.* Computer Graphics (Proceedings of SIGGRAPH), vol. 11, no. 2.

[21] Deering, M., Winner, S., Schediwy, B., Duffy, C., Hunt, N. (1988) *The triangle processor and normal vector shader: a VLSI system for high performance graphics.* Proceedings of SIGGRAPH, vol. 22, no. 4, pp. 21-30.

[22] Donnelly, W., Lauritzen, A. (2006) *Variance shadow maps.* Proceedings of the 2006 symposium on Interactive 3D graphics and games.

[23] Farin, G. (1996) *Curves and Surfaces for Computer Aided Geometric Design - a Practical Guide.* Fourth edition. Academic Press Inc.

[24] Goodrich, M.T., Tamassia, R. (2006) *Data Structures & Algorithms in Java.* Fourth edition. John Wiley & Sons, Inc.

[25] Goral, C., Torrance, K.E., Greenberg D.P., Battaile, B. (1984) *Modelling the interaction of light between diffuse surfaces.* Proceedings of SIGGRAPH, vol. 18, no. 3.

[26] Gouraud, H. (1971) *Computer display of curved surfaces.* IEEE Transactions on Computers, vol. 20, no. 6, pp. 623-629.

[27] Greene, N. (1986) *Environment mapping and other applications of world projections.* IEEE Computer Graphics and Applications Archive, vol. 6, no. 11.

[28] Hansson, H. (2007) *Craft Physics Interface.* [Electronic] Linköping: Institutionen för Systemteknik.

[29] Heidmann, T. (1991) *Real Shadows, Real Time* Iris Universe, no. 18, pp. 23-31. Silicon Graphics Inc.

[30] Hennessy, J., and Patterson, D. (1996) *Computer Architecture: A Quantitative Approach.* Second Edition. Burlington, MA: Morgan Kaufmann Publishers.

[31] Kajiya, J. (1986) *The rendering equation.* Proceedings of SIGGRAPH, vol. 20, no. 4.

[32] Kajiya, J., Kay, T. (1989) *Rendering Fur with Three Dimensional Textures.* Proceedings of SIGGRAPH.

[33] Krishnamurthy, K. and Levoy, M. (1996) *Fitting Smooth Surfaces to Dense Polygon Meshes.* Proceedings of SIGGRAPH.

[34] Lander, J. (1998) The Ocean Spray in Your Face. *Game Developer*, vol. 5, no. 7, pp. 13-19.

[35] Lax, P.D. (1997) *Linear Algebra.* John Wiley & Sons, Inc.

[36] Levy, S. (1984) *Hackers: Heroes of the Computer Revolution.* New York City: Anchor Press/Doubleday.

[37] Lin, M.C. (1993) *Efficient Collision Detection for Animation and Robotics.* Berkeley: University of California (Ph.D. thesis).

[38] Lindell, M. (2010) The Swedish game chart 2004-2009. *Swedish Games Industry.* `http://www.swedishgamesindustry.com/blog/2010/2/16/the-swedish-game-chart-2004-2009.aspx` (7 Apr. 2011).

[39] Luft, T., Colditz, C., and Deussen, O. (2006). *Image Enhancement by Unsharp Masking the Depth Buffer.* ACM Transactions on Graphics, vol. 25, no. 3, pp. 1206-1213.

[40] Markoff, J. (1994) Sony starts a division to sell game machines. *The New York Times.* `http://query.nytimes.com/gst/fullpage.html?res=9E0DE1DA1538F93AA25756C0A962958260&scp=27&sq=video+game+industry+1995&st=nyt` (7 Apr. 2011).

[41] Maughan, C., and Wloka, M. (2001) Vertex Shader Introduction. *NVIDIA White Paper* `http://developer.download.nvidia.com/assets/gamedev/docs/NVidiaVertexShadersIntro.pdf` (2 May 2011).

[42] McReynolds, T., Blythe, D., Grantham, B., and Nelson, S. (1999) *Advanced Graphics Programming Techniques Using OpenGL course notes.* Proceedings of SIGGRAPH.

[43] Microsoft (2004) Next Generation of Games Starts With XNA. *Microsoft News Center.* `https://www.microsoft.com/presspass/press/2004/mar04/03-24 xnalaunchpr.mspx` (7 Apr. 2011).

[44] Microsoft (2011) Stream-output Stage (Direct3D 10) *Msdn library* `http://msdn.microsoft.com/en-us/library/bb205121(v=vs.85).aspx` (14 May 2011).

[45] Microsoft (2011) X3DAudio overview *Msdn library.* `http://msdn.microsoft.com/en-us/library/ee415714(v=VS.85).aspx` (5 Apr. 2011).

[46] Microsoft (2011) XACT overview *Msdn library.* `http://msdn.microsoft.com/en-us/library/ee416126(v=VS.85).aspx` (10 Apr. 2011).

[47] Microsoft (2009) XNA Game Studio *Msdn library.* `http://msdn.microsoft.com/library/ee416788(VS.85).aspx` (7 Apr. 2011).

[48] Mittring, M. (2007) *Finding Next Gen-CryEngine2.* SIGGRAPH 2007 Advanced Real-Time Rendering in 3D Graphics and Games course notes.

[49] Owen, S. (1998) Painter's algorithm. *ACM Siggraph* `http://www.siggraph.org/education/materials/HyperGraph/scanline/visibility/painter.htm` (8 May 2011).

[50] Phong, B.T. (1975) *Illumination for computer generated pictures.* Communications of the ACM 18..

[51] Reeves, W., Salesin, D., and Cook, R. (1987) *Rendering antialiased shadows with depth maps.* Proceedings of SIGGRAPH, vol. 21.

[52] Samet, H. (1989) *The Design and Analysis of Spatial Data Structures: Computer Graphics, Image Processing and GIS.* Reading, MA: Addison-Wesley.

[53] Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H. (2001) *Texture Mapping Progressive Meshes.* Proceedings of SIGGRAPH.

[54] Scanlon, J. (2007) The video game industry outlook: $31.6 billion and growing. *Bloomberg Businessweek.*

http://www.businessweek.com/innovate/content/aug2007/id20070813_120384.htm?chan=search (7 Apr. 2011).

[55] Stamminger M. Drettakis G. (2002) *Perspective Shadow Maps.* Proceedings of SIGGRAPH, vol. 21, no. 3.

[56] Takafumi, S., Tokiichiro, T. (1990) *Comprehensible rendering of 3-D shapes.* Proceedings of SIGGRAPH, vol. 24, no. 4, pp. 197Ű206.

[57] Torrence, A. (2006) *Martin Newell's Original Teapot.* New York, NY.

[58] Ward, G. (1992) *Measuring and Modeling Anisotropic Reflection.* Proceedings of SIGGRAPH.

[59] Warnock, J. (1969) *A hidden surface algorithm for computer generated halftone pictures.* University of Utah.

[60] Watt, A., and Watt, M. (1992) *Advanced Animation and Rendering Techniques—Theory and Practice.* Addison-Wesley.

[61] Wikipedia (2011) List of Wipeout video games. *Wikipedia.* http://en.wikipedia.org/wiki/List_of_Wipeout_media (7 May 2011).

[62] Wikipedia (2011) Microsoft D3D. *Wikipedia.* http://en.wikipedia.org/wiki/Microsoft_Direct3D (7 Apr. 2011).

[63] Wikipedia (2011) Scanline Rendering. *Wikipedia.* http://en.wikipedia.org/wiki/Scanline_rendering (24 Apr. 2011).

[64] Williams, L. (1978) *Casting curved shadows on curved surfaces.* Proceedings of SIGGRAPH, vol. 12, no. 3.

8 Appendix - Game design

This appendix contains the original WipeIn - F-ϵ game design. As has been mentioned before, time beat us quite roughly and we were able to implement but a smidgen of the following ideas. In the unlikely event that this game should indeed have a future, this is what is to be expected.

8.1 Goal

The goal of the game is to circle the track as quickly as possible. The number of laps to complete is adjustable in the corresponding menu, as is explained in Section 8.6, and ranges from one to 20, with the default value being three.

8.2 Controls

The space ships are controlled with the keyboard and mouse. Available functions are accelerate, decelerate, turn left, turn right, merge left, merge right, use offensive power-up, use defensive power-up and use combo of power-ups. The default keys for these operations are w, s, d, a, e, q, left mouse button, right mouse button and middle mouse button respectively.

When the player holds down the accelerate button, the ship will accelerate until it reaches the maximum speed. The acceleration will trigger a jet-flame effect from the ship's exhaust pipes and a cool thrust sound will be heard. When the player holds down the decelerate button, the ship will do so until it reaches 10% of its maximum speed.

The turn left / turn right commands cause the ship to rotate accordingly along its yaw-axis as well as slightly along its roll-axis, and animated effects on the ship's wings help creating a realistic impression. The commands are similar to the accelerate/decelerate ones, i.e., the ship keeps turning until the player releases the key.

The merge left / merge right commands move the ship slightly to the corresponding direction without rotating, allowing the player to make small adjustments to the ship's position on the track.

The use offensive / defensive power-up commands trigger the power-up. Should the player be out of them, a clicking sound akin to that of an empty gun is heard, and a red empty circle is seen on the corresponding power-up's icon on the system bar (as described in Section 8.5).

8.3 Energy

A ship gathers energy that is necessary to use power-ups. When the energy level is below 100%, it constantly increases until it reaches the maximum level. This maximum level can be heightened by a certain power-up. The ship's energy capacity, as well as energy regenerating speed, depends on the ship type.

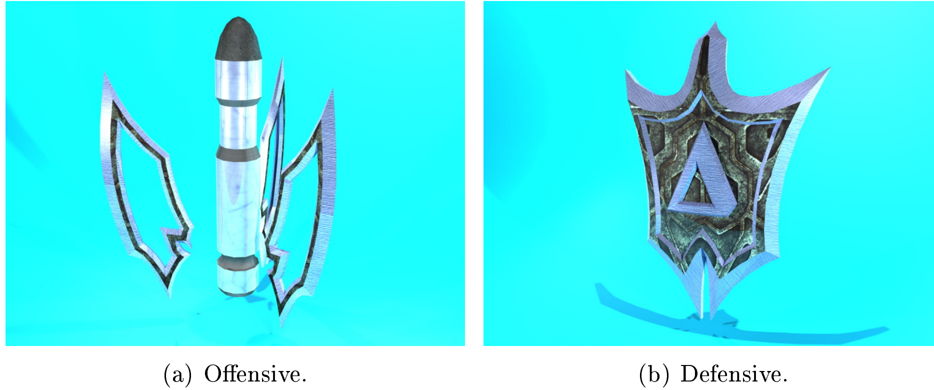
The energy is displayed in the middle of the status bar, illustrated with a circle with different colours; gold, representing energy, and white, representing the lack of it.

Each player's interface is equipped with a viewfinder, that the player controls with the mouse. The viewfinder is used to aim the fire at other players, in order to shoot them down.

8.4 Power-ups

8.4.1 General

A *power-up* is a miscellaneous component commonly used in computer games. Power-ups benefit or add extra abilities to the player's character, or in this case, ship.



(a) Offensive. (b) Defensive.

Figure 29: *Power-ups.*

The power-ups are scattered on predefined locations all over the map, floating slightly over the track. The nature of the power-up, i.e., offensive or defensive, is decided randomly. A player picks up a power-up by steering its ship on it, and once it is done the spot is deprived of power-up for 30 seconds, after which a new random one will emerge. Upon picking up a power-up, a sound will be heard and the player will be able to see a corresponding symbol on the status bar. However a player's ship can only store one power-up of each type at a time, and therefore a ship that drives over a power-up spot while already having a power-up of that sort will have no consequences whatsoever.

In order to use a power-up, a ship needs to have enough energy to do so. The energy level is, as previously mentioned, constantly increasing while playing, and decreases when a power-up is used.

8.4.2 Offensive power-ups

There are few different offensive power-ups available in the game, recognisable thanks to the missile-inspired design (see Figure 29(a)).

Missile

The use of a missile necessitates 100% of the energy level, and therefore a player needs to wait until the energy has reached its maximum before being able to shoot. Once triggered, the missile will blast off in a velocity higher than that off the ship, and will be accompanied by a jet flame and a specific sound. In the event that it hits another player, an explosion will follow and the hit ship will slow down abruptly.

The power-up corresponding to the missile contains three of them, which implies that the player will not be able to pick up any other offensive power-ups until all three missiles have been fired.

Seeking missiles

A seeking missile is similar to a missile, the difference being, as the name suggests, that it follows the enemy ship until reaching it or crashing into a wall. This implies that it is possible to shake off a seeking missile by appropriate manoeuvres. Also, the corresponding power-up only contains one item, i.e., once the seeking missile has been released, the player can pick up offensive power-ups anew.

A seeking missile follows an enemy ship for no longer than ten seconds, after which it continues in the last computed direction until colliding with an object, ship or wall.

Laser

A ship might also collect a laser power-up. A laser requires only 25% of the maximum energy level, and can be used eight times per power-up. When activated, the laser generates a straight red ray that reaches the destination instantly. An enemy ship getting hit by a laser loses 20% of its energy, as well as half of its speed. The laser is also accompanied by a characteristic sound.

Magnetic ray

When a player hits an enemy ship with a magnetic ray, both ships are connected by an invisible string that drags them closer to each other, in other words slows down the enemy ship and increases the speed of the attacker. In addition, whilst the ray is engaged, the rear ship drains the energy of the former. The ray is active until the ships are separated by a wall or until the emitter is, in turn, hit by a third ship.

8.4.3 Defensive power-ups

The defensive power-ups also have an adequate design, as can be seen in Figure 29(b).

Shield

When a ship acquires a shield power-up it is activated automatically. A shield then pops out at the ship's rear and a new energy status bar emerges on the player's interface, corresponding to the shield's state. A shield gives full protection against lasers and magnetic rays but only halves the effect of the missiles. When a shield has been hit six times it is no longer effective and it fades out.

Noos

The noos power-up provides the ship with a powerful turbo effect, increasing the speed to twice the original maximum value. When activated, the noos induces the jet flame generated from the ship to turn blue and produces a swooshing sound. The effect continues until all energy is utilised.

Gravitational field

A gravitational field is a static power-up, that once activated floats idly on the spot where it was released for 30 seconds, or until an enemy ship drives through it. The effects of doing so include decreasing speed and less responsive controls for ten seconds. Releasing a gravitational field requires 75% of the launcher's maximum energy, and it can only be done once per power-up.

Mines

Mines are also static power-ups that, once released, stay on the same spot. When a ship flies over a mine an explosion occurs, resulting in the ship getting twirled around. Both the releasing of and the flying over a mine are followed by characteristic sounds. The mine power-up holds three mines, and again, once these are utilised the player can pick up a new defensive power-up.

8.4.4 Combos

A player that possesses both offensive and defensive power-ups has the possibility to combine these into a *combo*. The nature of the combo, i.e., offensive or defensive, depends on the power-ups that constitute it. A combo is much more efficient than a normal power-up. The energy level required to activate a combo is the same as that of the most energy-craving power-up composing it. When a combo is triggered it utilises all power-ups included.

Missile - gravitational field

When firing a missile or seeking missile while possessing the gravitational field power-up, the missile hitting a target will have more effect on it, i.e., the deceleration will be more intense and will last five seconds longer.

Magnetic ray - shield

This combo has the same effect as the magnetic-ray combo, but in addition, the screen of the targeted player will turn black for eight seconds.

Laser - mine

This combo works just like the laser, with the additional feature that the targeted ship's shield is useless against it.

8.5 Grafical User Interface

When playing the game, the screen contains a status bar, located on the lower right-hand side. This bar provides the user with all necessary information relevant to the ongoing game, such as speed, energy level, available combos, laps done and laps remaining.

8.6 Menus

Upon starting the game, the player is greeted with a main menu, where several choices are available: start new game, options, credits and quit.

The user navigates through these options with the help of the arrow keys, and selects one by pressing *return*. An arrow icon next to the selected option provides information on the selected menu. This navigation is accompanied by sound effects.

Start new game

This option is followed by several other choices, such as track selection, ship type, number of laps, number of opponents and level of difficulty.

Options

This sub-menu contains additional system options, such as keybindings, resolution, sound settings and graphics settings.

Credits

Upon selecting this option, a scrolling list of the programmers and game designers is displayed, accompanied with flashy animations and pumping music that help create the illusion that the game makers did, indeed, surpass the very highest criterias of excellence.

Quit

As the name suggests, this option exists the program, with the mandatory prompt dialog box inquiring if the user is, in fact, thoroughly and unreservedly certain that he or she really wishes to close the application.