Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Design and implementation of application
independent easy-to-use game engine

by

Krzysztof Kondrak

LIU-IDA/LITH-EX-A--09/024--SE

2009-04-30
The objective of this Master's thesis is to study whether it is possible to create a versatile game engine that can be both application independent and easy to use. In order to study this issue, we have implemented a prototype game engine that supports 2D game development. The system includes predefined programming constructs in order to make game development faster and easier.

The conclusion of this thesis work is that the presented problem, while theoretically possible to solve, would introduce too many practical problems during the game development. Furthermore, we conclude that growth in ease of use may limit the functionality of the engine.

Keywords
game development, engine, 3D games, 2D games, easy, programming
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida [http://www.ep.liu.se/](http://www.ep.liu.se/)

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: [http://www.ep.liu.se/](http://www.ep.liu.se/)

© Krzysztof Kondrak
Design and implementation of application independent easy-to-use game engine

by

Krzysztof Kondrak

LIU-IDA/LITH-EX-A--09/024--SE

2009-04-30

Supervisor: Jalal Maleki
Examiner: Jalal Maleki
# Table of Contents

ABSTRACT .................................................................................................................................6

I. INTRODUCTION ..................................................................................................................7

1. Problem domain .................................................................................................................7
2. Research questions .............................................................................................................7
3. Thesis overview ..................................................................................................................8

II. PRINCIPLES OF GAME ENGINE DESIGN ............................................................................8

1. Overview of game engine technologies ..............................................................................8
   The BSP Algorithm ..............................................................................................................9
   BSP Engines and Portal Engines .......................................................................................11
   BSP Engines .......................................................................................................................11
   Portal engines ....................................................................................................................13

2. Requirements of an independent game engine .................................................................15

3. Basic requirements for programming a computer game engine .......................................17
   Firm understanding of programming ................................................................................17
   Know your Operating System .........................................................................................17
   Learn a graphics API .........................................................................................................17
   Learn about optimization ....................................................................................................17
   Mathematics .......................................................................................................................18

4. Technologies used in engine development ......................................................................18
   OpenGL ...............................................................................................................................18
   DirectX .................................................................................................................................19
   SDL and Allegro ..................................................................................................................19
   Mathematical requirements ...............................................................................................19
   Ease of use issues in game engine design and development .............................................20
   Selection of the programming language .............................................................................21
   Assembler .............................................................................................................................22
   C++ .....................................................................................................................................22
   Java .....................................................................................................................................22
   Flash and Macromedia Director .........................................................................................23
   Python .................................................................................................................................23

5. Overview of existing engines. Why try making another one? ........................................25
   Torque Engine .....................................................................................................................25
   Source Engine .....................................................................................................................25
   FIFE engine ........................................................................................................................25
   AGS Studio .........................................................................................................................26
   Spring .................................................................................................................................26
   Quake Engine ....................................................................................................................26
Abstract

The objective of this Master's thesis is to study whether it is possible to create a versatile game engine that can be both application independent and easy to use. In order to study this issue, we have implemented a prototype game engine that supports 2D game development. The system includes predefined programming constructs in order to make game development faster and easier.

The conclusion of this thesis work is that the presented problem, while theoretically possible to solve, would introduce too many practical problems during the game development. Furthermore, we conclude that growth in ease of use may limit the functionality of the engine.
I. Introduction

1. Problem domain

Even a brief involvement in computer game industry is probably one of the most thrilling experiences for a software developer. It's not only part of computer science that provide entertainment to the masses – it is one of the few opportunities, when you can become a true creator of worlds. It is the programmer who encodes the rules governing his own universe. It is the coder who enforces his own laws of physics by employing complex mathematical algorithms. It is also he, who can give actual life to the world using artificial intelligence and realistic models of trees, animals and beasts. Being a game developer is more than just being a programmer – it's being a god of your own small computer-generated world.

Sad as it may seem, it's not easy being a god. In order to create and give life as well as maintain a computer game world it is essential to know the basic rules that govern it. What this means is that before we can actually start coding, we have to dive deeply into basic mathematics, learn the ins and outs of graphics processing, as well as be very proficient in the API of our operating system. What adds up to the problem is that we would want to reuse the engine code with more than one project, so it should be written in a way that is easy and straightforward to use. Finally, we would want the engine to be as flexible as possible, letting us develop a myriad of completely different types of games in a short amount of time. The task at first seems daunting and impossible to achieve. But is that truly so? Hopefully this thesis will answer this and many other questions!

2. Research questions

In order to determine how difficult it might be to actually develop a fully capable computer game engine we will try to solve the following problems:

- What kind of knowledge do we need to start development of a computer game world/universe?

To answer this question we'll try to provide a set of guidelines, making it easier to create an efficient, application independent game development framework. “Application independence” in this context means, that the engine will not contain any game specific code, which would force us to modify it for every new project [2]. Here we will also address the requirements of technical knowledge: should we dive deep into concepts of 3D math and complexity of quaternion calculations or is it just enough to use pre-compiled set of libraries? Is it necessary for us to have in-depth knowledge of GPU optimization or is it just unnecessary burden for our minds? With each programming language there comes a different amount of knowledge required for game development. Depending on the language, there might be a various number of ready-to-use libraries that could facilitate the entire design process tremendously. This is a critical issue when timing the project is of importance!

- Given a set of different programming languages, which one would provide us with the best means of writing a fast, efficient game engine?
We will take a look at performance of several most popular programming languages: Python, ActionScript and Shockwave, C, C++ and Java which are currently core languages used in development of computer games. We will also try to estimate which programming language would be best, depending on the complexity and performance demands of a game project.

- **How much independence from the application itself can a game engine have?**

This is a critical question in computer game industry. In order to successfully reuse an engine with (sometimes completely) different product than originally intended, it is imperative that it's independent from the actual game project as much as possible. This is not only elegant but also solves a lot of trouble during game design, because there is no need for modifications of game specific code in the engine itself.

- **Is it possible to make our game engine completely universal and make it very easy to use at the same time?**

The key issue when we release the engine to the public: what we want to achieve is a programming framework which would allow us to create any type of game we want with as little effort as possible. On the other hand, the ease of use of engine code must be straightforward itself, so that even the programming novice can enjoy it.

### 3. Thesis overview

The first part of this work will concentrate on basics of computer game engine design and overview of already existing technologies used in this area. We will try to establish a methodology that would support its independence from game application to maximum extent. We will also address all problems listed above in order to create a list of goals that we want to achieve in our implementation. Part two of this work will demonstrate a simple example of a game engine designed in C++ using the established methodology. We will also analyze the level of independence of our designed engine and foresee how it could be further improved (providing a sample game as an example). In part three we’ll see, whether the given problem has been successfully solved.

### II. Principles of game engine design

#### 1. Overview of game engine technologies

To give a better overview of what a game engine designer is trying to overcome, let’s take a look at some of the technologies used in this profession. While it may seem surprising, the early concepts of game engines started to appear already in the 1980s, when game design business was still in its infancy. Computer games were not a software branch that people in the IT industry treated seriously, mainly due to the fact that personal computers were very slow and did not allow the programmers to introduce more complex effects and graphics without a severe loss of performance. This suddenly changed, when John Carmack, a programmer at Softdisk created a demo run of a *Super Mario Bros 3* clone titled *Dangerous Dave* in copyright infringement. This was the beginning of a new era in computer game industry [3].

The reason behind the great success of this simple game was that it used smooth screen
scrolling on a PC for the first time in history. Carmack’s solution proved to be so successful, that *Id Software* (a company which Carmack later co-founded with John Romero, Tom Hall and Adrian Carmack) later improved this engine with more features (collision detection, sprite interaction) and based their trilogy of *Commander Keen* games on it. The games were an instant hit and soon gave inspiration to game programmers all over the world on how to approach the problem of coding games.

![Dangerous Dave](image1.png) ![Commander Keen](image2.png)

*Figure 1: The game engine used in “Dangerous Dave” (left) was a base for the “Commander Keen” game series (right), which was a smash hit in arcade-game sector for the home computers in late 1980s and early 1990s (ingame screenshots, copyright Id Software).*

*Commander Keen* is a classic example of a reusable game engine. While the simplicity of its use might be questioned by contemporary programmers, it was still the same base code that evolved slightly with every subsequent game. However, 2D games is just the top of the mountain – the most popular association with the phrase *game engine* nowadays is obviously that related to 3D games.

**The BSP Algorithm**

At the dawn of computer game development there were many problems to be solved. Possibly the greatest mischief was rendering on home computers, which at the time did not possess too much computation power. This was the reason why different techniques of *scene management* had to be developed: one of them was a method that involved the Binary Space Partitioning algorithm (BSP) invented in early 1970s at the University of Texas at Dallas. At first its purpose was to create representations of 3D objects at various research facilities, but it was soon discovered that it could be used to render complex 3D environments in real-time even on computers that did not have any additional hardware support for graphics processing, such as the home computers. What developers didn't know at that time, however, was that BSP would revolutionize the computer entertainment [2].

Before further explanation of the BSP algorithm some basic terminology must be explained. In geometry there are 2 types of polygons that are especially important in 3D graphics: the *convex* polygons, and *conclave* polygons. Convex polygons words are a type of primitives that do not have any dents. This means that the inner angles of convex polygon are never greater than 180 degrees. A polygon that doesn't meet these demands is classified as a conclave polygon.
These terms can be easily translated into 3D space. If we were to be locked in a convex room, we would be able to see its every corner, no matter where we would stand (we disregard the fact that we would have to move our heads in order to see what's right behind us). Analogically, in a conclave room there would be certain areas, which could be only seen when standing in certain positions (it would be possible to hide from the viewer, hence some areas would be occluded). The same rules also apply to groups of convex or conclave polygons. This means that a group of convex polygons never occlude each other – a key property, that BSP algorithm employs.

The idea behind the BSP algorithm is very straightforward. Its objective is to split the game geometry into convex partitions using arbitrary partitioning planes. Each split results in two distinct groups: Geometry behind the splitting plane (called the backlist) and geometry in front of the splitting plane (called the frontlist). Each resulting set is then again partitioned using new partitioning plane. We perform these steps as long as the result of a split can produce a backlist and a frontlist – should the result contain only one of them, there is no further way of splitting the geometry. Figure 3 shows the creation of a BSP tree for a simple game map.

The idea of “front” and “back” for a given plane is usually solved by calculating a normal vector for the plane. By definition, these vectors always point outwards the surface and are aligned at the angle of 90 degrees to the surface, therefore they are very often used in graphics rendering for determination whether we are facing the front or the back of the surface. This is especially important during the rendering process when we want to skip the surfaces that the viewer cannot see.

While the BSP algorithm is easy to perform, there are more problems than meets the eye.
at first. In real life application it is important to choose the best splitting plane in order to get the BSP tree in the shortest time possible. Also, it is always good to get a balanced BSP tree, which would keep the search time at pretty much the same level for every rendered part of geometry. These however are the problems beyond the scope of this work and will not be discussed further.

**BSP Engines and Portal Engines**

Even though the sample implementation will not involve them, it is necessary to mention the two most popular scene management solutions which had a great impact on the world of computer game development: BSP based engines and portal engines.

**BSP Engines**

The history of the Binary Space Partitioning algorithm goes all the way back to 1970s, but it was not until 1993 when it was incorporated in a computer game for the first time. *Wolfenstein 3D*, a hit title of the early 90s released by *Id Software* is officially known to be the first FPP game that was able to draw the game world with a level of realism never before seen in computer entertainment. The “realism” of the game is a bit overrated if we look at the game from the perspective of modern computer games, due to a lot of restrictions that had to be introduced in order to get the game playable on home computers of that time. For example, every room in the game needed to have exactly the same height and walls had to be positioned to each other at an exact angle of 90 degrees. *Wolfenstein 3D* used a technique called *raycasting* to draw the environment. While the concept of semi-3D world was present in the games even in the early 1980s, *Wolfenstein 3D* was the first game to ever display textured walls and enemies [2][3].

![Image 1](image1.png)

**Figure 4:** “*Wolfenstein 3D*” (left) and its level editor: “*WolfEdit*” (right). Notice that the level editor was based on a 2D top-down view of the game world, due to the technical limitations at that time (ingame screenshots, copyright *Id Software*).

In 1993 the BSP algorithm was discovered as an excellent means of rendering 3D objects in real-time. *Doom* was the first game ever to use BSP for that purpose, even though the implementation of the algorithm had to be simplified: once again due to hardware limitations. For the first time it was possible to align the walls at angles different than 90 degrees and the rooms could have varying heights, although the floors and the ceilings still had to be horizontal and the walls vertical. The reason behind this last constraint is because *Doom* (and its successor, *Doom 2*) used the so called ‘2D variant’ of BSP algorithm. This means that the algorithm was used to split *lines* instead of actual 3D *polygons*. This was possible only with an assumption that there is no additional geometry between floors and ceilings, which would have to be included in the
calculations. With these assumptions it was possible to perform splitting of 2D lines instead of 3D polygons in real-time even on a simple home computer in 1993 (note that a map in Doom looks like an interconnection of lines when looked from above, which back then was an additional advantage that enabled game developers to introduce area maps during gameplay) [2].

![Figure 5: “Doom” in-game screenshot (left) and its level editor (right). Notice that even though the graphics quality changed tremendously, the level editor is still top-down view based (ingame screenshots, copyright Id Software).](image)

It was not until 1996 when Quake (another masterpiece by id Software) literally shook the grounds of computer game world by fully implementing the BSP algorithm. This game was neglecting all previously introduced constraints: it was finally possible to align surfaces and 3D polygons in the game world at any angle, any height and at any position in space. Quake also introduced real 3D models for the game-controlled characters, which added another degree of realism. The ability to move freely in 3D space in the game made its engine a base for most of the future 3D games – a process which we see even today. The initial problem was the fact that Quake engine was not portable to other platforms. However, due to low popularity of non-Microsoft operating systems it was not considered a major flaw at the time [3].

![Figure 6: “Quake” in-game screenshots. This game introduced a completely new dimension of realism, which back in 1995 was quite astonishing. Notice that the game characters (left) are now fully three-dimensional unlike standard 2D bitmap sprites used until then (ingame screenshots, copyright Id Software).](image)
Portal engines

At approximately the same time when *Quake* was being developed, another group of programmers from *3D Realms* started developing their own game engine, hoping to be at least as successful as their competitor’s product. In 1995 they released *Duke Nukem 3D* which displayed a completely new type of quality since the release of *Doom*. The game included multi-level buildings, vast cities bursting with life, mirrors (which up until then were thought to be impossible to achieve in a FPP game) and possibilities to interact with the environment in a lot of different ways (using the toilet, breaking furniture, smashing windows etc.). Even though the enemies were still represented by 2D sprites, the game was more realistic than anything seen before. *Duke Nukem 3D* was the first example of using what is now known a portal engine [2][4].

![Duke Nukem 3D](image)

*Figure 7:* “Duke Nukem 3D” was one of the few games to display a realistic sky and outer space (using a parallax effect), even though a similar effect was already used in Id Software’s “Doom” (ingame screenshots, copyright 3D Realms).

The key difference between a BSP based engine and a portal engine is that the former uses a tree structure to render game environment, while the latter does not. Figure 8 explains in better detail the ideology of portal engines.

![Portal Engines Diagram](image)

*Figure 8:* The concept of portal engines.

The concept behind portal engine is pretty straightforward. First, an internal processor splits the entire game map into separate sectors, each of which composes a convex part. The sectors are interconnected with each other by the means of portals, which are in principle a form of gateways from one sector to another. The task of a portal game engine is to render only the sector the player currently resides in, and the sector that resides within the view frustrum. What is important to note here is that neighboring sectors don’t have to reside next to each other in the map data file. This means that the structure on figure 8 could be very well stored as a set of seemingly unrelated shapes, scattered in an unordered manner. This was not the case when using BSP, which requires that all adjacent areas in the map are “physically” connected [2].
In other words, moving in a world created by a portal engine is somewhat similar to teleportation between different spots on the map. This has been used as an advantage in the first portal engine map editor *Build*, supplied with *Duke Nukem 3D*. The designer was able to interconnect adjacent sectors using *sector effectors*, which literally teleported the player between them. During level design it was very important to make passing between sectors as realistic as possible – in many cases during the gameplay the player was transported when riding an elevator or walking into a dark corridor, which to his perception looked very natural. Also, the idea of portals was used in the first FPP implementation of underwater environment – diving (or emerging from underwater) was done by triggering the portal and moving the player to a sector with “underwater” or “surface” flag set. At that time the portal engine was at an early stage of development, so introducing transparent water surface was for obvious reason impossible. This however was soon changed, when 3D Realms released *Shadow Warrior*, a game using an improved version of the *Duke Nukem 3D* engine (called “*Build*”), allowing not only to produce transparent water, but also 3 dimensional object sprites, such as weapons and keys.

The feature that gave portal engines the biggest applaud from masses of gamers worldwide was introduction of mirrors in the game. The design behind mirror was to simply create two adjacent sectors separated by a wall with a “mirror” attribute set. Depending on the player's location, the engine would render the sector he currently resides in on the other side of the mirror, only in reversed order, including the sprite representation of the player. In this case *Alice in Wonderland* surely comes into mind, as the idea is quite similar, the difference being in that the engine prevents the player from passing through the mirror. The idea is depicted on the following figures.
The fact that it's not necessary to physically connect adjacent sectors and the ability to create mirrors is not the only great advantage of this technology. With portal engines it is possible to create games which can behave in ways that would be impossible to implement using BSP. One example is creating a labyrinth of rooms interconnected by doors which are in fact portals to different sectors of the map. Using randomization it would be possible to make the door lead to different parts of the labyrinth with every different instance of the game. This is but one of the many ideas that could contribute to more interesting (and not necessarily realistic but quite entertaining) gameplay.

2. Requirements of an independent game engine

What makes a game engine independent? The most obvious answer to this question is: no relation between its code and the specific project. What this means is that the engine must be capable of working with other projects without a need of any drastic code modification. The bottom line: engine code must be autonomous [2].

How can we achieve code independence? The first thing that comes to mind is modularization of the design. Obviously, an object oriented programming language sounds like a perfect tool for this task. One must however remember, that object orientation does not fully solve the problem – some specific design measures have to be undertaken before the actual coding can start.

This means that it’s necessary to come up with means of separating game/OS specific parts from the actual engine code. One way of achieving this is to use the abstract interfaces. These are basically code skeletons without actual implementation – the user has the access only to class declarations and its methods. The engine developer has to take care of the implementation of an interface class, which might vary on different operating systems. The interface itself, however, is at all times unchanged and independent of the actual implementation. There is a variety of ways this effect can be achieved in most contemporary programming languages: for example, in case of C++ we can use virtual classes and Java provides developers with an interface class.
Figure 12: Interaction of the application and the engine using an interface [2].

There is one more advantage of using interfaces in a game engine code – it makes it possible to maintain independence from the graphical API. This may not sound so unusual from what has already been said, but there is a lot more to this statement than meets the eye at first. The main problem is that there is a great variety of different graphical APIs available to game developers, DirectX and OpenGL being the two most popular fractions in this area. For a better visualization of the problem let’s assume that we implemented our engine with support for OpenGL. This sounds like a quite reasonable solution, because we can target both Windows and Linux audiences, so our products can become quite versatile. However, let’s imagine now that we found a company that was interested in using our engine with games dedicated to the Xbox console. This poses a certain problem, because Xbox supports only DirectX, so the natural approach would be to release our engine with support for that particular API – and this is where the magic of interfaces come into play. Having an abstract interface it would be necessary to only add a new implementation dedicated to DirectX, while the other parts of the engine could be left untouched. Without interfaces this would be not as easy, as we would have to search through the entire code for any graphics related function calls.

With this brief introduction it is time to think about what our engine should contain. Unfortunately for game developers of the 21st century, the gamers themselves have become very demanding – they require high quality graphics, sound, support for all types of different interfaces (joysticks, mice, keyboards etc.) and for most of them releasing a game without at least basic network support is unacceptable. To additionally extend this list, we have to bear in mind that there is a lot of work “behind the scenes” - things that make the entire engine “tick” are by far the most challenging to implement. We’re talking here about the game physics, handling of different types of lightning, shadows and of course scene management. Figure 13 shows an example of a game engine with accompanying modules and how they interact with each other.

Figure 13: Example of a modular engine structure [2].
Notice that we will be using interfaces as a means of communication between the engine and the game application in all cases. This is a necessity in context of what we've already said about engine independence – not only the code should be autonomous in all cases, but also it should be easy to replace each and every module at need without excessive engine code modifications.

3. Basic requirements for programming a computer game engine

Having reviewed the basic techniques used in a computer game engine we can now ask the question: what exactly do we need in order to create one? Following is a short list of the most important skills that we have to acquire [1][2][4][5].

Firm understanding of programming

This is the single and foremost skill, without which progress is impossible. Currently the advancement of programming development is so high, that it’s possible to write a game engine from scratch using majority of the most popular languages: C++, Java and Python being just the top of the mountain. In theory it is possible to create a playable entertaining product using a language that has at least a basic support for advanced graphical processing. In practice, we want to find solutions that will provide us not only with nice visual and playable effects, but also high speed and as small encumbrance of the processor as possible.

Know your Operating System

It's not a secret that in order to maintain high level of performance the designer should know the operating system inside-out. Without this knowledge most of the optimization routines cannot be fully implemented, because of which the entire playability of the engine might become quite jittery and unsatisfactory.

Learn a graphics API

Unfortunately for today's game engine programmers, a swift comprehension of AT LEAST one graphical API has become a necessity: OpenGL and DirectX being just two most popular examples. The advancement of our technology is currently at the level of pushing the graphical adapters to their limits – this is being achieved by developing even faster GPUs and rewriting the basic assembly code to provide as quick access to rendered data as possible. Because of this and a high demand on visual quality, creating a game engine that has no support for advanced hardware rendering support is doomed to fail in the long run unless we want to target just a specific type of games (for example simple 2D puzzle games).

Learn about optimization

An invaluable asset and an addition to the skills of programming – perfect maintenance of a complex computer world with thousands of objects require a high degree of knowledge about making the computation processes as fast and efficient as possible. In dire cases it might come down to the level of Assembly programming, where the implementation of basic mathematical operations is needed. This however is not the only case: on the higher levels of the engine's core there is still a lot of work to be done, concerning rendering of the basic primitives, proper scene
management and handling of objects hidden from the player's view, so that they are not unnecessarily rendered to the screen (sometimes drastically increasing the requirements for memory in the process). In other words – we need to take care of pretty much everything so that our engine’s performance does not frustrate the users.

**Mathematics**

At first it may seem that the only thing math is responsible for is implementation of physics and handling different types of lightning, depending on the position of the player/object against the light source. There is, however, a lot more to this than meets the eye at first. In a game engine mathematics is virtually everywhere – starting from the basic movement of the models, finishing on the movement of the camera itself. All those processes involve deep understanding of operations on matrices, vectors, rays and surfaces and in some situations even complex numbers. This thesis does not concern mathematical introduction to the problem of game engine programming – the topic is wide enough to write at least several books about it. The bottom line is that mathematics is the language of digital nature and without it it's impossible to get our game world into motion.

As we can see this short list of requirements is enough work for an average developer for at least a couple of months, before he/she can actually get a grip on how to handle such a complex programming construct. However, if we don't have that much time on our hands, high level programming languages and ready-to-use libraries come to our aid. In the following chapters of this work we will try to dig into this problem and see, whether it is possible to create a very easy to use programming framework, that would allow us to create a fully legitimate fairly-advanced computer game with a limited amount of technical knowledge. The basic assumption however is that the developer has a good grip on the basics of his/hers programming language of choice, which is in every case mandatory.

**4. Technologies used in engine development**

Lets now take a brief look at the most popular technologies used in game development. These vary from low-level graphics and input/output routines to more simplified libraries which handle most of the complicated work for the programmer.

**OpenGL**

One of the most popular graphics libraries, OpenGL was developed as an open standard, meaning that any external contributor could take part in the development process. OpenGL provides the user with a large set of functions that allow the programmer to render triangles and more complex polygon structures in a fairly straightforward way. It also supplies the user with some basic mathematical transformation functions for vectors and matrices which is essential when performing movement or rotation on the screen. OpenGL is known to be a highly portable graphics rendering platform and is available for nearly all system architectures, varying from Windows and Linux to MacOS. This graphics library is said to be one of the best documented, mainly due to abundance of contributors throughout the world. On the other hand it also makes some parts of documentation inconsistent for the same reason. Nevertheless a computer graphics adept will feel quite comfortable while learning and applying OpenGL in his/her software.
**DirectX**

With the growing popularity of OpenGL, Microsoft decided to make a move of their own. In 1995 the company presented their break-through operating system called Windows 95 and along with it the first edition of their own graphics/audio/input/output programming library called DirectX. The audience was very excited, not only because of the fact that Microsoft is releasing an operating system which would support game development. The key reason behind it was the first game ported to DirectX: *Doom* by Id Software - the same game that in 1993 caused mass hysteria among the computer gamers all over the world. Even though the first releases of DirectX were far from successful (due to high amount of programming bugs and arguably low functionality in comparison to OpenGL), the library itself evolved through years, becoming a very tough opponent for OpenGL. DirectX not only was written natively for the Windows platform, it also had its own extensions for handling mouse and keyboard events (DirectInput) and sound (DirectAudio). Even though DirectX is designed only for Windows (there are no known ports of DirectX for other systems due to legal issues) it is widely used today both for PC development and console games (for example XBox 360).

**SDL and Allegro**

Unlike OpenGL and DirectX, Allegro and SDL (Simple DirectMedia Layer) are libraries designed specifically for creating multimedia applications. Both of them come with a set of functions that automatically handle input, audio and graphics, although SDL is said to be a more advanced variant of Allegro and gives the possibility to directly access screen and audio buffers contents. When making games using SDL and Allegro, the programmer doesn’t have to think about handling low-level routines such as drawing primitives: all is taken care of by the built in routines working behind the scenes. Both SDL and Allegro are portable libraries and have been known to have their own variants designed for different programming languages (such as Python or Java).

**Mathematical requirements**

Apart from technical knowledge, there is one more ingredient that is tightly related to game end engine development: math. Most contemporary computer games and development libraries rely on mathematical transformations in order to perform basic actions, such as movement or rotation of objects on the screen. Things start getting worse when we try to implement realistic physics or other special effects such as fire (which involves random particle movement) or liquids. In most cases, the programmer is required to have a sound knowledge of matrix and vector operations and in some cases even complex numbers. The good news is that most libraries have these operations built in, so things get a little less complicated. The bad news: in many cases the speed of operations performed by these libraries is not optimal enough to provide satisfactory performance on modern computers, given that we’re planning to use a lot of complicated effects. Many contemporary game engines take a step forward and implement complex transformations themselves, in most cases using the assembly language which generates the fastest code possible. What makes these problems a bit less daunting is the fact that they are usually encountered only in games which use high-end graphics and highly rely on optimal usage of available hardware resource. There are however several things that both advanced and simple 2D games share, such as performing collision detection between objects on the screen. Advanced math can be used in such simple games as well, for example to generate certain objects with random properties (for example patterns, textures or even sprites).
Ease of use issues in game engine design and development

The first problem to face when developing a game is the complexity of the libraries and other tools we are using. This is especially difficult for beginner and intermediate programmers who really want to make games but don’t have enough technical background to do it efficiently. Let’s analyze the situation from the point of view of such a programmer.

Assuming that we would like to use OpenGL or DirectX to develop a game, we would first have to deal with the problem of drawing contents of the screen. Both of these libraries are fairly-low level and the concept of a shape is based on drawing groups of triangles which later form more complicated constructs. As simple as it might seem there is a slight problem here, because the programmer has to think not only about the imaginary coordinate system on the screen (so that he can place the triangle at the exact spot he wants) but also about optimal usage of graphics card memory which is the bottleneck when it comes to rendering high-end graphics. As an example, here is a short code snippet to draw a triangle in OpenGL and DirectX (notice how complicated it looks in the latter case!):

```c
glBegin(GL_TRIANGLES);
glVertex(0,0,0);
glVertex(1,1,0);
glVertex(2,0,0);
glEnd();
```

(Pseudo code, and incomplete)

v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0; v++;
v->x = 1; v->y = 1; v->z = 0; v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);

**Code 1:** Example of a source code in OpenGL (left) and DirectX (right) to draw a triangle (source: John Carmack’s .plan file).

While the OpenGL way of programming seems to be a bit more friendly at the first glance, we must remember that the presented code handles only drawing of content on the screen. In practical game programming there is still a lot of other things to handle: audio, input, memory management being just the top of the list. While we don’t disregard OpenGL and DirectX as excellent libraries for software development, they are most certainly not the best choice when starting off with the adventure of game development and may prove cumbersome when trying to implement other features of the application than those related to rendering graphics.

Now consider using a more high level library, such as SDL [12]. Here things start getting a bit more simple, because we don’t have to worry ourselves about drawing routines and complicated handling of audio and input. SDL comes with a nice set of functions, that do it for us automatically. Consider the following examples of loading and playing a sound or drawing a rectangle on the screen:

```c
/* Draw a rectangle */
void draw_rectangle(SDLSurface *surface, Uint32 color) {
    SDL_FillRect(surface, NULL, color);
    SDL_BlitSurface(surface, NULL, _screen, NULL)
}
```
void play_music(const char *filename) {
    Mix_Chunk *sample = Mix_LoadWAV(filename);
    Mix_PlayChannel(-1, sample, 0);
}

Code 2: Example code of two functions in SDL used to draw a rectangle and load and play a sound.

It does look much simpler. However there are some issues that we will still have to cover due to lack of functionalities in pre-made libraries. These things concern mostly efficiency: optimal usage of CPU and GPU as well as memory handling. Every decent game should manage its resources in such a way that it does not load copies of certain objects (for example textures) more than once. The same problem concerns sounds and other types of reusable data. In many cases there is a certain set of operations that we will have to perform in a game (such as collision detection) and this is still something that needs to be implemented (although one can argue whether a true game engine should actually support this feature, since it might be treated as a game specific feature). Nevertheless, SDL is a better solution for beginner game programmers though it lacks those simple elements that make the game fast, non-resource consuming and, most importantly, scalable.

This is the problem that we will try to solve. Providing the users with a simple to use game engine we will also supply them with necessary optimization features that will be working behind the scenes without bothering the programmer. We will also try to go even further and offer the user some basic structures (such as ready to use implementation of obstacles, moving objects) which will make the game development faster than ever. Before we dive into coding though, we should first choose the best programming language for the job and also familiarize ourselves with design principles of a game engine.

Selection of the programming language

Let’s now take a deeper look at the technology we will use to develop the game engine: the programming language. This is a very important step in the design process which determines the future code structure as well as the flexibility of the framework. Another key issue is the desired performance of software using the engine which in this particular situation is crucial, such as handling of the drawing routines, most data processing and complex calculations. In order for our application to be efficient, we have to ensure that all tasks will be performed in the fastest way possible which (apart from code optimizations) is highly dependent on the structure of generated binary code. So how do we choose the best programming language for the job?

Let’s consider the “top players” in game development world. Currently the most often used programming languages in this area are: Java, C, C++, Python and Flash/ActionScript along with Macromedia Director. We will focus on development for the PC platform only, disregarding the console market. High popularity of these languages is a result of on different historical and marketing events: C++ has been a language used for most top-performance application development since 1980s. Java is known for its simple syntax and extensive amount of additional libraries, making it a fairly powerful language. Python gained popularity through almost the same reasons, as well as the built in force-mechanism to write clear code using indentations. Finally, Flash and Director have been long known in development of rich online content. Let’s now learn something more about them and try to decide which one will best suit our needs.
Assembler

Even though the Assembler is considered a relic now, it has to be taken into consideration due to its part in the software industry. Assembler is almost as ancient as computers themselves: only one step behind pure machine code it is considered the most powerful and the fastest programming language known to man. The downside of this is very high complexity of syntax and a requirement of excellent computer hardware knowledge. Writing code in assembly is basically putting instructions directly into the CPU. For this reason it is possible to write really fast applications. The downside however is length of the code (usually several thousand lines in case of more complex programs) and very low portability between different CPU types. That’s why Assembler is no longer used as a primary language in game development. It is however used for writing inline routines which are prone to be executed in as least time as possible (for example inline assembly code in C++ programs). Thus far, the most popular game written in Assembly was Mortal Kombat – a fighting-style game by Acclaim which in 1992 was as big a hit as Doom in 1993.

C++

C++ is known to be the de-facto standard in the world of advanced game development. We used the word “advanced” on purpose here for several reasons. One of them is the amount of control over the system that the programmer has when using C++. It is not only an object-oriented programming language (which is nowadays very common) but it is also one of the few that gives its user total control over entire memory (both available and already used by other programs). This however makes it one of the most difficult languages to master, which is why C++ is used primarily by expert programmers and in products that require maximum performance. Less experienced developers prefer languages with automated memory control, which in most cases is sufficient to develop a fairly fast and optimal application (especially on current home computers which come with a lot of computation power). C, the older brother of C++ is in general opinion considered to be a language used for developing faster and less resource consuming application. However the fact that it does not come with a built-in object oriented programming model seldom makes it a choice in game development, with small exceptions in the area of programming additional utilities and/or testing applications.

Java

Java was a secret project at first, the goal of which was to develop a high-performance language for internal use at Sun Microsystems. After a course of time the project evolved and turned into Java, which became a general-purpose programming language available for everyone to be used for free. Java very soon found recognition among developers, especially because of its syntax being close to the one used in C++. This made the learning curve somewhat gentler for programmers. Java featured automated memory management and a built in garbage collection (automatically taking care of unused data in memory), something that made the language popular among beginner coders or those who found the concept of memory management too difficult in C++. The major plus side of Java is the amount of additional packages and libraries that come with the standard edition of the language. This means that Java developers can start taking advantage of advanced language features right away, unlike the C++ programmers which have to get hold of any additional libraries themselves. Java also uses a concept of virtual machine, which makes it possible to run the exact same code on any operating system that has a Java VM installed on it. This means no need to make code modifications in order to make it portable: something that was not always possible in C++. On the other hand, Java is considered a slow language: mainly because
the programmer lacks the possibility to arrange data in memory by hand and slight delays introduced by using the virtual machine instead of direct code execution. Nevertheless, there are advanced graphical libraries available for Java, making it a potent platform for game developers.

**Flash and Macromedia Director**

Unlike Java and C++, Flash and Director are frameworks which allow the coder to create very rich graphical content in a short amount of time. Primarily designed for creating online presentations and multimedia, Flash soon caught the interest of game programmers due to ActionScript: an internal scripting language that Flash supported. While not being able to create advanced standalone applications, Flash turned out to be an excellent tool for making simple and rich games which afterwards could be put on the Internet. Director is considered an advanced edition of Flash. It supports graphics acceleration and is virtually capable of creating complex 3D content on the screen: a fact that has also been utilized by game developers in making really advanced and playable games. Unfortunately the only platform that supports Director is Microsoft Windows, which dramatically limits the audience of such applications.

**Python**

Python is a substantially new player on the gaming market. Designed primarily for scripting simple applications using a very clean syntax, it soon grew beyond its creator’s imagination. Apart from advanced string processing libraries, a series of extensive graphical APIs have been ported to Python. A great advantage of this language is its possibility to use C++ libraries as well, provided that they have been previously compiled into external libraries. This makes Python a very powerful tool, although the fact that it requires an interpreter to execute programs is why many programmers disregard it as a serious development application. Extensive amount of different types of libraries makes Python a great tool for programming fully eligible applications. Python is in continuous development, currently there are several projects underway (such as the SDL port for Python: PyGame [13]) which aim at making Python an even better platform for game programmers. The author of this work is currently involved in a Python game development project called *SnowballZ*, which utilizes OpenGL and PyGame in a real time strategy game aimed at younger audiences.

![Figure 14: Examples of games written in Assembler (“Mortal Kombat”, left) and Python (“SnowballZ”, right) (ingame screenshots, copyright Midway Entertainment and Zeolite Studios).](image)

Given the above choices, is it possible to determine which programming language is the best tool for the job? There is no simple answer and everything is highly dependent on what we
want to achieve. The following table confronts our set goals and how easy they are accomplished in each corresponding language:

<table>
<thead>
<tr>
<th></th>
<th>Assembler</th>
<th>C++</th>
<th>Java</th>
<th>Python</th>
<th>Flash/Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of use</td>
<td>Low</td>
<td>Average</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Development speed</td>
<td>Low</td>
<td>Average</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Game development friendliness</td>
<td>Low</td>
<td>Average</td>
<td>Average</td>
<td>Average</td>
<td>High</td>
</tr>
<tr>
<td>Data management friendliness</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Average</td>
</tr>
<tr>
<td>Memory management support</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Execution speed*</td>
<td>High</td>
<td>High</td>
<td>Average</td>
<td>Average</td>
<td>Low</td>
</tr>
<tr>
<td>Code portability</td>
<td>Average</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Multi platform support</td>
<td>Average</td>
<td>High</td>
<td>High</td>
<td>Average</td>
<td>Low</td>
</tr>
<tr>
<td>Low level routine support**</td>
<td>High</td>
<td>High</td>
<td>Average</td>
<td>Average</td>
<td>Low</td>
</tr>
<tr>
<td>Library structure support***</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Overall score:</td>
<td>20</td>
<td>27</td>
<td>23</td>
<td>24</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 1: Comparison of different programming languages' performance (based on author’s empirical experiences).

*In some cases execution speed may differ due to different language implementation schemes. Here however we assume that it was done in the most optimal way possible.

**This property determines how easy it is to insert/create low-level code in the program for additional performance improvements.

***Determines how easy it is to turn code into an external library which can then be used by other languages.

The grading was done according to established requirements in the preface of this thesis. Scores have been calculated according to the following scheme: Low – 1pt, Average – 2pts, High – 3pts. We can see that C++ stands out from anyone else in the comparison, making it the best language of choice for us. Even though Ease of Use is considered slightly lower than in cases of other languages, the built in properties of the language make it possible to create a friendly programming API for external users (mainly due to its object oriented nature which makes it possible to create separate pieces of code with only a single interface visible to the user). What this means is that even though the process of engine development might be a bit slower than in Java or Python we will still have the advantage of higher performance (given that the code is written correctly).

Provided that we selected C++, does this disqualify other programming languages in the field of game development? The short answer is: no. Depending on what we are planning to make, it is sometimes best to choose different programming tool which will help us achieve the goal in shorter time and with little or no difference in overall performance whatsoever. Surely designing a game with 2000 AI units and rich graphical content might perform better when written in C++ than in Java. On the other hand, in case of a simple game with only 10 AI units and low or average graphics we might not even feel the difference, therefore making Java the programming language of choice might lead to faster development with less frustrating issues (such as memory leaks which is quite common in C++).
5. Overview of existing engines. Why try making another one?

When we take a closer look at the existing game engine “market” we can actually see a good variety of available software, some of which can be used and downloaded free of charge while others might require buying out a license from the creators. Let’s take a brief look at the most popular ones and try answer the question: why would we want to make just another game engine?

**Torque Engine**

Developed by GarageGames, Torque is mainly aimed at development of FPP games. The most known disadvantage of this engine is its robustness and difficult usage, which causes a lot of problems for novice game developers. What adds up to this is the fact that it requires an expense of $250 for licensing fees. Nevertheless, Torque is also known for having an extensive set of physics modules which in some cases makes it worth the money spent, despite an average learning curve when using it. Torque is available for Windows, Linux and MacOS systems and has been recently ported to the iPhone platform [16].

**Source Engine**

This is by far one of the most popular game engines known today. Being in constant development by Valve Software it is said to be one of the best 3D engines available for the PC. Source was popularizes by such hit titles as the Half-Life 2 series, Portal or Team Fortress 2. While the performance of the engine is commendable, it is optimized mainly for high-end FPP games. Currently the Source Engine supports the Windows, Playstation 3 and Xbox platforms only and requires a high licensing fee, making it available to only the big gaming companies on the market [11].

**FIFE engine**

Unlike Source and Torque, the goal of FIFE project is to develop an optimal game engine for 2D games with an isometric perspective, like the one used in strategy games such as Warcraft and the Settlers series. The engine code is currently in early beta stage and is under the GPL license, meaning that everyone can download the code for free as well as contribute. Since the API is being constantly modified it is difficult to determine how easy the end product will be to use by game developers. FIFE is being developed in mind for the Windows and Linux platforms [15].

![Figure 15: Examples of games created with the Source Engine (“Half-Life 2”, left) and FIFE engine (right) (ingame screenshots, copyright Valve Software and the FIFE Project Group).](image-url)
**AGS Studio**

This is an engine that comes up with its own development environment enveloped in a form of a GUI interface. AGS is a game engine aimed at creating old-style adventure games (i.e. based on simple 2D graphics). With AGS it’s possible to create a game that has the same look and feel as the adventure games made by Sierra-On-Line and LucasArts in 1980s and early 1990s (both with a mouse point-n-click or just a command line interface). While not purely code-based, it has an extensive number of scripting features which makes it really easy to create additional effects or special sequences in a game. Though only 2D adventure-game oriented, AGS is highly regarded as an excellent tool for game development and is widely used by several independent gaming studios (the most popular being AGD Interactive who is known for releasing high quality remakes of old Sierra adventure game) [14].

**Spring**

*Spring* is an engine unlike all others. This piece of software basically doesn’t require the programmer to have any knowledge of programming languages typically used for game development. In case of *Spring* only knowledge of a scripting language called LUA is required. It is used to code the entire gameplay of the designed game with a possibility to import your own textures and character models. While fairly easy to use, the major concern with this engine is the fact that it’s made specifically for Real Time Strategy games, greatly limiting its application to only one type of gameplay [9].

**Figure 16:** Examples of games made in AGS Studio (“Quest for Glory 2”, left) and Spring (right) (ingame screenshots, copyright AGD Interactive and The Spring Project Group).

**Quake Engine**

The prodigy of Id Software, *Quake Engine* has been release under the GPL license. Like most publicly available game engines it is designed for development of 3D FPP games. The difference though is the fact of it being publicly available, which caused it to fork into many different branches maintained by Id Software fans. This resulted in an offshoot of different variations of *Quake Engine*, some of which have been completely modified and adjusted to different type of gameplay, such as third person perspective games and even racing games. The *Quake Engine* project is officially closed and is currently kept somewhat alive by hobbyist programmers.
**Voxlap Engine**

While Voxlap is just another 3D FPP game engine it is nevertheless worth mentioning in this work. Designed by Ken Silverman, it is available under a GPL license, though it didn’t seem to have caught too much attention by the programming public. What makes this engine unique is the fact that its graphics renderer uses voxels (three dimensional pixels) as the basic building block of the game world instead of classic polygons. What this means is that the Voxlap Engine gives the developer theoretically unlimited possibilities of re-shaping the environment (from digging holes in the ground to taking down the whole buildings in completely random ways). One widely known game using the voxel technique is Voxelstein 3D, which is a remake of the old hit Wolfenstein 3D. The major disadvantage of this engine is the fact that high performance can be achieved only at fairly low screen resolutions, 320x200 being the most optimal one. This is probably the main reason why Voxlap is considered a rather inferior technology. What is interesting is the fact that a voxel-based game engine doesn’t require a 3D accelerator or a fast graphics card: all calculations are being performed by the CPU [10].

![Figure 17: “Voxelstein 3D”, a game made using the Voxlap Engine. The screen on the right was taken a few seconds after causing random damage to the nearby environment. Notice that the voxel technology allows nearly unlimited manipulation of the game world, making it a lot more flexible than the classic polygon-based engines (ingame screenshot, copyright The Voxelstein Team).](image)

By now it is has become evident that most of the available engines are aimed at development of 3D games. A deep-through research shows that there is no widely known game engine that would support both 3D and 2D game development, which in most cases can be explained by infeasibility of such projects. This is understandable, considering that a large part of game engine software is being developed by big game companies which are always constrained by deadlines. The Open Source projects, such as the aforementioned FIFE, also concentrate on one type of gameplay. This can be easily explained if we consider that focusing on one particular feature of the product is the best way to make it perfect. Still, the idea of choosing either 3D or 2D features usually makes it difficult to develop games in the second format. This might lead to a situation where creating a 3D game might be quite easy (using the given API), but when trying to make a 2D game we are running into numerous problems or are forced to use obscure syntax. Given these conclusions, we will make an attempt to design an engine that fills this void and makes it possible to implement both 3D and 2D games in as simple way as possible.

6. **Sample structure of a game engine**

To assure that the engine is completely independent of the application, we have to make sure that its modules are completely separated from it. In order to achieve this we will create an
interface between both parts that will play the part of a liaison between the application and engine’s modules. Such a structure is depicted on figure 18.

![Figure 18: Structure of an example application independent engine](image)

Note that the interface is in a bilateral relation with all its modules. This means that the interface is not just a static caller of module methods – the modules themselves can communicate between each other. The reason why we’re using the interface for this is because we want to keep the modules as much independent of each other as possible. This is not only a good programming practice, but also makes it very easy to further extend the functionalities of the engine.

While the structure is fairly general, the selection of modules is dependent on the amount of functionalities we would like to support the programmer with. We may divide these functionalities into two separate groups:

**Basic functionalities:**
- **Renderer module:** set of methods which will supply the application with drawing routines, texture handling and framerate optimization.
- **Audio module:** set of methods which will supply the application with sound support and different file format handling.
- **Core module:** set of methods providing other types of functionalities, such as special hardware handling, some input handling etc.
- **Utility module:** This module contains any additional helper functions that do not fit in any other category, as well as necessary tools that maintain high performance of the engine. This would include scene managers (optimizing handling of different scenes in the game), resource managers (preventing duplication of resources in memory) and any external libraries that the engine might be using (package management, different file format management, additional helper libraries etc.).

**Advanced functionalities:**
- **Network module:** this could be arguably classified as a basic functionality due to the fact that most contemporary computer games come with at least basic support of network gameplay.
The network module makes it possible for the game to communicate with others computer by the means of a local network or the Internet, allowing multiple players to share a single gaming experience.

- **Logic module**: The core of handling all game related routines, such as collision detection, path finding algorithms etc. This modules is subject to most changes during development lifetime of the engine due to its extensive amount of possible functionalities.

- **Font module**: A decent game engine should have a possibility to display not only high-end graphics but also simple plaintext on the screen. In many cases this is not only limited to True-Type Fonts but also texture-based bitmap fonts (i.e. fonts built from a set of bitmap images).

- **Math module**: If we want the engine to perform mathematical transformations in as shortest time as possible we are most likely inclined to implement our own mathematical library. This module usually contains definitions of basic mathematical constructs (such as a vector or a matrix) and implementations of mathematical operations, most often written in a fast low-level language (such as Assembler).

As we can see the list of modules can be very extensive and it’s completely up to the designer which ones he would like to support in his engine. In the next part of this work we will try to implement the presented structure.

### III. Example implementation in C++: The PongEngine

#### 1. Description, contents and capabilities of the engine

In this chapter we will analyze a sample game engine aimed at trying to fulfilling the goal of this work, meaning that it will have the following properties:

- Easy to use API
- Code portability and platform independence
- High performance
- Built in mechanisms to manage resources and memory
- Backend allowing to create any type of game

The engine has a modular structure, which means that it’s very easy to extend its capabilities with additional functionalities and any adjustments or bug fixes can be done directly in a module without having to modify other parts of the engine. To make the game development even easier, the engine is supplied with several additional external modules which can be used for different purposes, such as creating a moving/active object on a screen (referred to as actor), creating time dependent events (using specially crafted timers) and several more. The available API is constructed in a way that gives the programmer a free choice as to which module he wants to use. The entire engine structure is depicted on the following figure.
The modules have been divided into **internal** and **external**, the difference being in that external modules can be used by game programmers directly in their applications (i.e. the programmer can create standalone instances of these objects), while internal modules are meant only to be used via the engine interface (user cannot create a standalone instance of an internal module). For the purpose of this project several new data types have been defined in order to facilitate usage of different resources (images, sounds) which the programmer can use in his own programs as well.

### 2. Description of the engine modules

To get a better overview of what the capabilities of PongEngine are, let’s take a closer look at the implemented modules:

**Internal modules**

- **Renderer module**: This part of the engine is responsible for visualizing content on the screen. It also implements basic drawing functions (basic shapes). The renderer has a built in frame-rate regulation, meaning that it reduces CPU usage by the engine significantly.
- **Audio module**: Contains routines related to playing and loading sound and music files. PongEngine currently supports most known audio formats: WAV, MID, MP3, OGG, XM, S3M, MOD and AU.
- **Logger module**: This internal utility helps keep track of error messages during program execution. Available to the developer are 3 levels of code debugging: info (non-critical messages), warning (non-critical messages which might have some impact on application’s performance) and error (critical messages which might be a result of a fatal error).
- **Core module**: Contains all non-grouped functionalities as well as input handling (mouse and keyboard).
- **Logic module**: contains all game related useful functionalities, such as collision detection and performing simple path-finding algorithms.
- **Font module**: handles true-type fonts and their display on the screen.
- **Animation module**: this could be considered as a part of the renderer but due to the significance of animation in games it was given a separate branch in the code. Animation module is responsible for all special effects that occur on the screen including basic frame-by-frame animations of the sprites. Both 3D and 2D effects should be put in this module.
- **Package module**: this engine component is responsible for handling resource data that are stored within a compressed archive. This is a key feature, allowing the programmers to deploy their games with only one additional file containing all textures and sounds, making it more convenient. PongEngine currently uses zip files as a resource storage, though it is also possible to read data directly from an uncompressed file on the disk.
- **Resource Manager module**: This is the core memory optimization module of PongEngine. Resource Manager keeps track of all data loaded into the program and makes sure that there is only one copy of each resource in the memory at all time. In other words its purpose is to save the memory and avoid duplication. This module uses the C++ concept of templates, which makes it possible to create a separate manager for every type of resource (graphics, audio and potentially more as the functionalities of the engine expand).

**External modules**
- **Object module**: this is the base of any PongEngine game object. It’s available for use by the programmer so that he has the flexibility of creating new game data types that would suit the needs of the application.
- **Actor module**: inheriting directly from Object, the actor is a representation of all active objects in the game. This includes monsters, the player and other animated creatures, though the usage of actors is not limited to just “living” characters.
- **Obstacle module**: same as in case of an actor, the obstacle inherits directly from the Object. Its purpose is to represent any static structures in the game (mainly intended for obstacles, walls, floors etc.).
- **Particle module**: particles have a very profound meaning in game design, for they are used to make many special effects, such as fire and smoke. Using the particle module the programmer can create any type of particle effects he wants in several simple steps.
- **Timer module**: timing is essential in any game and it is often use to pre-determine the course of events during gameplay. This module can be used for this purpose and it can be also tied to actors, obstacles as well as used independently.
- **Bitmap font module**: this module handles fonts built from bitmap data. With its help, the programmer can easily create different sets of bitmap fonts and connect them to his application without having to call internal module methods of PongEngine.
- **Game Screen module**: This module is intended to facilitate scene management in the application. When using game screen it is possible to create new scenes in a very easy and straightforward way without the need of writing any additional code. All the programmer has to do is to inherit from this module and fill the contents of the screen with whatever he wishes.

As we can see the diagram of PongEngine fairly well corresponds to the general engine structure presented in Part I. What we are neglecting, however, is the math module. For the purpose of this thesis we will come up only with the 2D solution of the PongEngine, limiting its capabilities in relation to the thesis goals. We will however produce such a code structure to make it possible to easily extend the implementation with 3D features.
Technologies used

PongEngine uses SDL as the core functionality drive. This library was chosen as a good base, with mind of future replacement by a bit faster option: DirectX or OpenGL. Nevertheless, SDL is proven to be fast enough which makes it a good tool to base the engine on. Other external dependencies include a zip handler library by Mark Adler, the purpose of which is to let PongEngine read external resources from a compressed archive.

The reason why there are so few dependencies is due to our desire to keep the engine as independent from other software as possible. Future release of PongEngine will include removing the zip library and replace it with its own package format. Also, using OpenGL and DirectX (as an option for the Windows platform) as the base of audio, video and input handler will successfully remove the need of having any additional libraries included with the package.

3. Development progress

The development of PongEngine turned out to be a more educative process than initially suspected. The code started off as a simple batch of code and several unrelated functions, which could do as much as only draw several shapes on the screen in a very scattered manner. At certain point of development the amount of methods grew so rapidly that in order to keep it easily maintained (and make it more obvious to use for the programmer) the concept of separate modules was invented. At first the entire set of methods was supposed to be grouped only into the graphics, audio and input part, however as coding and development of several sample programs progressed it became clear, that more advanced games might require additional effects and special handling. Therefore the animation module was introduced, taking over several drawing methods from the renderer. At this point it became clear that the number of modules might still increase during development, therefore to ease the use of the entire API the modules were all grouped in one single interface object, called simply ‘PongEngine’. Through this interface the programmer was able to communicate with all existing modules, while the functionality extension of the engine was still easily maintained.

When the project was in its early beta stage the idea of built in game types (objects, actors) was proposed as no other existing engine seemed to offer this capability. For this reason the code was remade so that it could support internal modules (the already existing ones) as well as external classes that the game developer could use directly without having to refer to the main engine object. External modules have another excellent advantage: they can directly call necessary internal modules without user’s interference. Also, the development code allows the programmer to add his own modules in a very easy and straightforward way. The goal of PongEngine is to be an Open-Source project to which everyone can easily contribute, so ease of extension is especially important. This implementation was an official version 0.2 of the project.

Next code iteration included adding resource managers, the logging module and package handling. At this point the structure of PongEngine was so concise that implementing these features took very little time and integration into existing code didn’t prove problematic. Version 0.3 was very close to being the final one, though it still suffered from several performance issues. It turned out that in many function calls memory content was constantly copied from one place to another, which in a long run could cause severe slowdown of application execution. This was easily resolved using the C++ language construct called references. Basically, when a function call uses a reference it tells the program to use the argument of the function directly, instead of making a
local copy of it (which took place before the modification). At this point several memory handling functions were introduced in order to prevent unwanted “memory leaks” during application runtime (i.e. all unused memory should be released to the operating system so that it could be reused again). Also several minor changes were updated in the code, such as keeping variable naming consistent and fixing the code layout to be more readable.

The final stage of development was documentation. For this purpose DoxyGen system was used which produces a clearly laid out documentation both in pdf and in html format. This documentation will be further expanded when PongEngine will officially be released online. Also, the development of sample games was started at this point during which several minor bugs were found and fixed.

Game samples

For the purpose of demonstration of PongEngine’s current capabilities two games have been created: Space Pong and Fire Madness!. Both of these games have a different type of gameplay, though they are 2D based which is a result of the current limitations of the engine. Nevertheless, they demonstrate both a simple game (in case of Space Pong) and a more complex production, both of them coded in under 5 days total (excluding the actual work on assets, such as sounds and textures).

“Space Pong”

Space Pong is a clone of a very old game, called Pong. The goal of the game is simple: the player controls a rectangular paddle and his objective is to win a ping-pong styled game against the computer (the game supports 2 player game as well as a battle against a moderate computer AI). The game is using only built in data types without any external textures, making it a very basic and simple production. It also features a space-star effect in the background which was made using the built in particle module. Considering the development time (1 afternoon!) in relation to rather low gameplay complexity it is noted that PongEngine meets the demands of the desired engine performance.

“Fire Madness!”

The aim of this game was to create something unique and a bit more complicated than Space Pong, which would resemble a typical gaming production. Fire Madness! is an arcade game, where we play the role of Blake Frost whose task is to defend a mushroom village from the raging flaming heads. The game features several types of enemies exposing different types of behavior, various powerups that enhance player’s skills and weaken the flaming heads, as well as hand drawn pixel graphics. The game also features a contribution from a musician, Mikael Johansson, which makes the game feel like one of the old productions from 1980s. Fire Madness utilizes the animation module of PongEngine and the built in physics with collision detection. Thanks to resource management the game only takes up 15MB of memory at most (when checked the results without resource management a peak of 26MB memory usage was noted: nearly twice as much!). In principle, the only things that had to be implemented during development was the gameplay scheme, enemy behavior, player behavior and scoring system. The game uses a finite state machine to switch between different game stages (title screen, game screen, help screen), each one of them implemented using the Game Screen module. Considering rather moderate game complexity, it is noted that PongEngine meets the demands of our desired framework.
Code example

Let’s now take a quick look at how to write code when using PongEngine. Bear in mind that the example is meant to demonstrate the basics of PongEngine programming and must be extended in length in order to produce more usable features. Consider the following code snippet:

```cpp
class Monster : public Actor {
public:
    Monster(PongEngine *engine_ptr, int wdth, int hght, const char *image) :
        Actor(engine_ptr, wdth, hght);
    ~Monster() {};
    void move(); // custom functions defining movement
};

Monster::Monster(PongEngine *engine_ptr, int wdth, int hght, const char *image) { 
    _engine = engine_ptr;
    _actor = _engine->load_image(image);
}

void Monster::move() {
    if(get_direction_x() == DIR_LEFT)
        _offset.x--;
    else
        _offset.x++;
}
```

**Code 4**: Example class implementing a simple monster creature

The `Monster` class is a representation of a moving game character. In this case we just implement the `move()` method which is responsible for its movement. Should the monster face left we decrease the `x` coordinate by one. Otherwise, we increase it. The constructor initializes internal engine pointer and the bitmap image of the monster which is required by the `Actor` class. In order to initialize the object in the game and draw it on the screen, the following code can be used:

```cpp
PongEngine* engine = PongEngine::create(); // fetch PongEngine instance
Actor* monster = new Monster(engine, 100, 100, "monster.bmp");
monster->draw_at(10, 10); // draw monster at (x,y) screen location
monster->set_direction_x(DIR_LEFT);
monster->move();
```

**Code 5**: Initialization, drawing and movement of the monster
This is enough to set up an empty screen with a monster, set its bitmap size to 100x100 pixels and draw it. The last function call initializes monster movement. In this very simple case we’re not implementing any controls, so the monster will keep moving constantly to the left until it vanishes at the side of the screen. Nevertheless we can see that it is possible to create basic game units in a very few lines of code when using PongEngine.

4. Similar work

NeoEngine

NeoEngine is a product of Reality Rift Studios and Power Challenge AB in Sweden [6], an online computer games developer. Version 1 is available for free use according to the GPL license, while version 2 is confident and kept for internal use by Power Challenge. The reason why NeoEngine is being mentioned is due to its extensive 3D capabilities, which allow the programmer to create games with completely different types of gameplay. From a technical perspective, the engine supports both DirectX and OpenGL and has built in scene and resource management, thanks to which the memory state is maintained to a fairly good extent. Also, it implements its own mathematical libraries along with basic data types (such as a vector or a multidimensional matrix) making the overall performance quite satisfying. As of today, there is one fully developed production using NeoEngine: Power Soccer, an online football game resembling a hit product by Electronic Arts: FIFA 2009. While the quality of graphics is not as high, the general feel of the game is almost identical, making it a highly playable product. Power Challenge is currently developing Power Racing and Power Football, which are in principle a racing and an American football styled games. Both Power Racing and Power Football represent games with completely different properties and gameplay, leading to a conclusion that NeoEngine is a product that might meet the requirements of this work. The main flaw of NeoEngine, however, is its complex usage and a rather sharp learning curve. The engine code has a multi level modular structure, making it quite difficult to learn by novice and even intermediate programmers. On the other hand, application development goes quite smoothly and fast after mastering the complex function calls.

```
neo::core::Array<std::pair<neo::core::HString,
 neo::core::Array<neo::animation::Op*> > >::iterator x;
```

Code 6: An example of declaration of the iterator variable in NeoEngine. While the syntax might prove extremely complex and cumbersome, the engine itself does provide a lot of flexibility and extended features.

NeoEngine is targeted at 3D games only and has no easy to use interface for 2D game development, making its application a bit limited. Being under constant development, there are occasional performance issues surfacing up, though they are resolved in a fairly short amount of time by the development team. NeoEngine is designed for the Windows operating system and has a limited support for Linux and MacOS platforms [8].
Figure 21: “Power Soccer” (left) and “Power Racing” (right) are examples of two different types of games developed in NeoEngine (ingame screenshots, copyright Power Challenge AB).

**SDL Engine**

This is a small one person project which, just like PongEngine, uses SDL for basic graphics, audio and input handling. The difference is in that it’s written in the C programming language, meaning that it has a non-object oriented structure. Also, SDL Engine is designed to support only 2D games, therefore its development process will not include more complex applications. Nevertheless, the fact that it’s oriented at C programmers means that it can be used to write very small and fast games, though not necessarily in a short development time due to lack of the object oriented model. The basic functionality of SDL Engine is to provide a simple wrapper for SDL functions and some additional effects (such as particle emitters, parallax scrolling and bitmap fonts). Depending on a point of view, the main disadvantage (or advantage) of SDL Engine is the fact that it’s non-object oriented nature forces the programmer to write his/her code in a procedural manner. Of course there is a possibility to write a wrapper in C++ turning the engine into an object oriented construct, however due to a high level of indirection in function calls the application performance might drop. On the other hand, SDL Engine is designed for 2D games only, so this slight slowdown might not make that much of a difference. The project is in early beta stage and there are no publicly known games made using it. It is currently released under the MIT license and the source code is available to download for free [7].

**5. Future work**

Even though this thesis project came to an end, PongEngine is already in further development to support creation of 3D environments. A very early alpha version of the code allows drawing simple shapes (such as cubes and spheres) and constructs that are based on them (i.e. simple characters and other shapes). A total conversion to OpenGL as the graphical device is underway in order to drop external SDL dependency and make the entire source code easier to read. Also, this will solve the problem of having to attach additional dynamic library files with developed application which is now the case due to usage of SDL. PongEngine 2.0 will currently use only OpenGL due to its open format and portability, as well as slightly easier syntax when coding procedures. Apart from supporting Linux the project is aiming at being able to develop applications even on mobile devices, such as Symbian OS based mobile phones and iPhones. The current documentation for PongEngine will be available on the Internet at [http://www-und.ida.liu.se/~krzko208/pongengine](http://www-und.ida.liu.se/~krzko208/pongengine) for 1 year after this report had been published. For a more recent code and updates contact the author at krzysztof.kondrak@gmail.com.
IV. Summary and conclusion

The goal of this work was to come up with a game engine design that would support any type of game (both 2D and 3D based) as well as answer the question whether it’s possible to create an API that would fulfill this goal and yet be very easy to use even for novice programmers. Even though the current implementation supports 2D games it is nevertheless possible to extend this functionality even to 3D games. We can achieve this thanks to the simple and modular structure which allows adding new effects and functionalities in a very easy and straightforward way. However the question whether engine usage will be still easy at that point is a matter of uncertainty.

This is because software usability is constantly tied to its complexity. In case of game engines this is especially important. A potential problem that might arise is the fact that adding new effects and features might increase the number of required functions which forces the programmer to relearn the API extended with new methods. At this point PongEngine easily supports 2D game development and most complex drawing routines are closed within single function calls which do all the backend work for the developer. In case of 3D development this might be not enough, because lack of more detailed function calls might limit the potential functionalities of the engine. This is the case with the aforementioned NeoEngine, which while being quite complex and difficult to use provides the application developers with highly flexible interface and allows creation of various special effects. Therefore this might be a problem that PongEngine 2.0 could face as soon as it reaches its beta stage. A general relation between software complexity and ease-of-use is shown on figure 22. As we can see, one solution to maintain this thesis’ goals would be to use simplification, such as enclosing predefined complex procedures within one simple function call. This however could greatly influence development time and could prove to be less efficient coding and performance wise in relation to the initial design.

For this reason it’s difficult to give a straight answer whether PongEngine fully meets the demands of an easy to use versatile and universal game engine. We can tell that it’s on a good path, however whether it will stay on it in the course of development cannot be answered at this time. Early demo versions of the initial 3D capabilities look promising, as we are able to create complex shapes and perform simple interaction with the environment, however it’s difficult to predict how additional special effects will influence the entire structure. The current design intends to extend the animation module as well as provide the user with a very easy switch that will determine beforehand whether the game will be 2D or 3D based. This will cause several
functions to have multiple applications though they will be called in the same way in both cases.

Apart from potential increase in complexity there is another, bigger problem: game designers. The gaming industry is incredibly extensive, the ideas for new types of games are being created every day and there is always a chance that some innovative concept might base on a completely unpredictable type of gameplay. In other words, human creativity might be a natural PongEngine limitation that is impossible to overcome. It might turn out that a certain gameplay will require a completely new engine design and new functionalities, making PongEngine useless and thus failing in fulfilling the goals of this work. However due to the fact that this is a problem that cannot possibly be solved (at least not with current technology) it is safe not to count it in as a design flaw.

Another question we might ask is whether the current project stage is truly universal (i.e. even if we consider only 2D games at this point). Considering the limited capabilities of 2D space and its properties we might assume that PongEngine is capable of supporting any kind of game of this type. The problem arises when we try to quantify the number of effects it can produce automatically as well as other things, such as unconventional behavior of physics. In this area the developer still MIGHT have to rely on his own programming skills. However, since PongEngine is in constant development that problem might soon dissipate or be reduced to a minimum. Therefore we can assume that at the current stage of development PongEngine is indeed a universal easy to use 2D game engine. Whether we will be able to extend this term also to 3D games is to be answered by its successor, PongEngine 2.0.
References


Appendix: terminology

FPP – First Person Perspective – type of a game, where the player sees the game world through the eyes of the game character.

TPP (3PP) – Third Person Perspective – type of a game, where the player sees the game world from a certain perspective behind the game character.

Sprite – in early days of computer games sprites were used to project characters and objects in the game world. They are 2D images that can be easily scaled and shifted, which gives the player a fake perception of 3D. In modern computer games real 3D models are used instead, however sprites are still often used as means of displaying special effects (smoke, explosions, fire etc.).

Rendering – generation of an image on the screen.

Scene management – a set of algorithms and methods used to select polygons needed for rendering, depending on the current location and orientation of the player in the game world.

Resource management – ability of the game engine to handle duplicate copies of objects in memory (ie. saving memory by keeping just one copy of each object).

Depth buffer – a special buffer (implemented in hardware or in software) that stores information about the distance of a given polygon from the viewer. Depth buffers are very important when solving the problems of visibility – thanks to them it is very easy to determine which objects are visible on the screen depending on the viewer's position in space. Depth buffers also help establishing which objects occlude each other etc.

HSR – hidden surface removal – removing of polygons which cannot be seen from a given position and orientation of the viewer.

Collision detection – a set of algorithms and methods that can determine whether two or more distinct objects in the game world share common space at a given instance of time. This is most often achieved by creating a bounding box around active models and static polygons. This reduces the problem of differently shaped object to checking whether distinct cubes intersect each other.

View frustrum – the area visible to the game camera. Anything beyond the view frustrum is clipped away from view and is not rendered on the screen.

Parallax effect – A sky parallax is a technique of rendering a certain texture in such a way that it appears to be fixed in one place no matter of the current camera position, hence creating an illusion of a large bounding space most often used as the sky. A parallax scrolling effect is used mostly in 2D games where we want to keep the perspective of distant object, i.e. while the character is moving on the screen we can see more distant objects scrolling at lower speed, therefore having an illusion of screen depth.