Otto Hietala

Developing a Game Engine With SDL

Thesis
Kajaani University of Applied Sciences
Business
Business Information Technology
26.04.2011

THESIS
ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Otto Hietala	
Title Developing a Game Engine with SDL	
Optional Professional Studies	Supervisor(s) Janne Koponen
	Commissioned by
Date 26.04.2011	Total Number of Pages and Appendices 36

Simple DirectMedia Layer is not a game engine, but a multimedia programming library. It might not be the best one available and it certainly has less functionality than most similar libraries. However, it is one of the most popular libraries available. Why? It is open-source, cross-platform, easy to use, simple and expandable. SDL has been around for some time, so it has gathered quite many followers. As its functionality is rather limited, most people use SDL only for window and input management and leave the rest of the necessary functionality to other libraries.

During the last few decades, video game engines have become an important aspect in the industry and have even become a business for some companies. As today's games require a lot of functionality in almost every aspect of gaming, a ready-made game engine can help a gaming company (whether it is a new company or an industry veteran) save time, money and effort. The Internet is full of open-source game engines, rendering engines and such like, but the industry has been trusting an old, trustworthy proprietary engines.

The purpose of this thesis was to study both game engines and SDL and to develop a simple and small game engine using SDL, OpenGL and some other libraries. The engine would be used in a game project. The project consisted of four developers and the deadline was 3-6 months. The game engine should support all the functionality the game needs, no more and no less. SDL was chosen as a base library because of its cross-platform and expandable functionality.

Language of Thesis	English
Keywords	programming, game, game engine, SDL, C++, Project
Deposited at	☒ Electronic library Theseus ☒ Kajaani University of Applied Sciences Library

CONTENTS

Legend

.NET	A software framework. Programs written for .NET run in an application virtual machine.
Alpha	A phase in the release cycle of a software, where developers usually test the software themselves.
API	Application Programming Interface. A set of rules, standards and/or specifications that the programmer follows when developing an application.
Batch rendering	A method of rendering, where multiple sprites are rendered at the same time with a single rendering call.
Beta	A phase in the release cycle of a software. This phase follows alpha and usually the software in this phase might be tested by a third-party.
C	One of the most popular programming languages of the world. Originally created in 1972 by Dennis Ritchie, C is procedural and structured language.
C#	Pronounced C Sharp. Microsoft's programming language that first appeared in 2001. C# is structured, object-oriented and functional language.
C++	A programming language designed to be an enhancement of the original C language. Created in 1979 by Bjarne Stroustrup. C++ is object-oriented, procedural and generic language.
Collision Detection	Methods of checking whether two objects in a game world collide or not.

Collision Detection, Broad Phase

A phase in the collision detection, where colliding objects are checked from a larger area. Usually the collisions themselves are not calculated in detail.

Collision Detection, Narrow Phase

Follows broad phase, in this phase the collisions and resolutions are calculated in more detail.

Compiling

The process of creating an executable software from source code.

Direct3D

Microsoft's API for 2D and 3D computer graphics, designed to work only on Microsoft's platforms. Part of DirectX.

Doxygen

A software that creates documentation from a program's source code.

First Person Shooter

A game where the game world is viewed from the player characters eyes and contains more or less shooting of some sort.

FPS

Frames Per Second or First-Person Shooter

Frames Per Second

Refers to how many frames are rendered per second. The bigger the value, the better the hardware and/or software.

Framework

Provides specifig functionality to developer by providing the developer with source code that provides specific functionality that can be easily expanded or used by itself.

Game engine

A programming library or a piece of software, which provides developer(s) certain functionality related to developing games.

GCC

GNU Compiler Collection. A popular compiler for Unix-like operating systems and part of the GNU toolchain. GCC is the standard compiler for Linux, BSD and Mac OS X. GCC can compile C, C++ and a number of other programming languages.

GLSL

Short for OpenGL Shading Language, a high level shading language created so that developers have more direct control of the graphics pipeline.

GPL

Abbreviation of General Public License. A free software license written by Richard Stallman. A software licensed under GPL gives anyone the right to use, copy, modify and share the software and it's source code under some limitations.

GPU

Graphics Processing Unit, the graphics card, the piece of hardware that does all the drawing/rendering on the screen.

HLSL

High Level Shading Language, developed by Microsoft and is analogous to GLSL.

LGPL

Abbreviation of Lesser General Public License. A modification of GPL which allows linking between GPL and non-GPL licensed software.

Linking

The process of combining all of the object-files created by the compiler and creating the executable. Object-files contain machine code.

Low-level

In computing, usually refers to how close to the actual hardware a user/program is operating. IE. Assembly, a programming language, is low-level language, because Assembly-code is assembled directly to machine code. C++ is a high-level programming language, because the source code is compiled to executable.

MinGW

A port of the GCC compiler to the Microsoft Windows operating system. Can also be used to cross compile source code from Windows systems to Linux systems.

MMO

Massive Multiplayer Online, refers to a game that usually has hundreds or even thousands of players in the same game-world at the same time. Development and maintaining requires lots of resources.

Mods

Short for modifications. Simply a modification made to a game, like new characters or levels. These days many games come with modding tools.

NPC

Non-Player Character. Refers to a character in a game that is not controlled by any player.

OpenGL

A cross-platform API for 2D and 3D computer graphics.

OpenGL ES

A version of OpenGL modified and optimized for Embedded Systems.

Programming Library

Contains source code providing certain functionality/functionalities that helps the programmer. IE. a math library provides functionality related to math, so the programmer does not have to create that functionality himself/herself.

Release Candidate

A version implying that the software is almost ready to be released. Usually all the major bugs in this version has been fixed and only minor ones exists. Follows beta.

Rendering

Drawing an image from resources, for example, drawing an 3D model or sprite image on the screen.

Repository	Storage that holds all the files and other data from Version Control.
Ruby	A object-oriented programming language designed by Yukihiro Matsumoto, first appeared in 1995.
SDK	Abbreviation of Software Development Kit. Program(s) that allow the developer(s) to create software, usually with a certain library or editor (or both).
Separating Axis Theorem	A theorem that can be used for collision detection. This theorem works only with convex bodies.
Version Control	A software that is used for managing changes to program source code, documentation, etc. Also known as revision control.
WebGL	An 'expansion' to JavaScript-language that allows the developer to generate 3D graphics in a web browser.

1 INTRODUCTION

Game engines have provided developers with basic game building blocks for decades. Developers use a lot of time, money and other resources on creating a game engine, while other developers save a lot of time, money and other resources by using engines created by other developers. Choosing to create an engine or using a ready-made engine might save the whole company, because both choices have their flaws and merits. Using a ready-made engine saves time and money on engine development, but if the developers have no previous knowledge of the engine, the learning process might take some time. Creating an engine requires lots of resources, but the developers can implement the exact features they want to the engine and if the engine is top notch, the company can sell the engine itself. Also, developers must choose libraries which the engine uses and platforms the engine supports. There are two major rendering APIs, OpenGL and DirectX. Choosing to use both for supporting most platforms and getting most out of rendering on Windows is time consuming, because the two APIs are not similar to each other. Choosing one is simpler and faster, but restricts developers to certain platforms, especially with DirectX. OpenGL supports more platforms, but DirectX is more powerful on Windows.

OpenGL

A cross-platform API for generating 2D/3D graphics. First released in 1992 by Silicon Graphics, it is now one of the most used APIs for graphics generation. It competes with Microsoft's DirectX on Microsoft's Windows platforms. OpenGL has been ported to many many programming languages and works on many different platforms. OpenGL is developed and maintained by a consortium called Khronos Group.

OpenGL is considered to be a stable, trustable and dynamic API. OpenGL is designed to be simple and easy to use, thus it is used only for graphics generation. All other functionality, including window creation/handling, has to be managed by third-party libraries. OpenGL's usage is not limited for PCs only as OpenGL ES is an OpenGL version for embedded systems such as mobile phones, PDAs and handheld gaming consoles. OpenGL has also jumped to the browser world, as WebGL is becoming more and more popular alongside HTML5. OpenGL SC is a stripped-down version of OpenGL, where game functionality (and the like) is removed for minimal code size and simplicity. OpenGL SC also has some previously removed functionality, like display list, for legacy code and auto-generated code.

OpenGL SC is used in areas where safety and stability are critical, like in avionics, industrial, military, medical and automotive markets.

OpenGL is not without criticism and scepticism though, as OpenGL version 3.0 caused stir among developers as the release was not what the initial proposal had promised, but was a disappointment and a let-down and many developers said that they would switch to DirectX. Although OpenGL and DirectX both render 3D graphics, their functionality differs which can be seen in their graphics pipelines. (Image 1.) (Image 2.) (Wikipedia. OpenGL.)

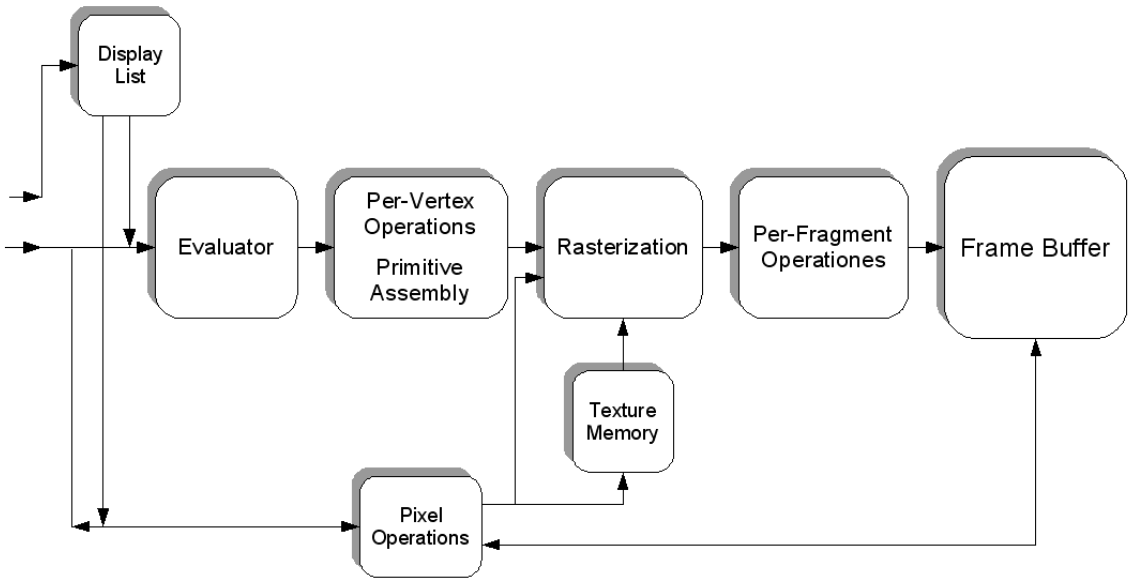

Image 1: OpenGL graphics pipeline. (Wikipedia. OpenGL.)

DirectX

DirectX is not an actual API itself, but a group of APIs with each one having functionality over certain area, for example, Direct3D renders 3D graphics, Direct2D renders 2D graphics and DirectSound plays and records sounds. Many people used the term DirectX when they talk about Direct3D. DirectX is generally favored for it's features, Windows and XBox support and general easiness of use. Many favor DirectX over OpenGL for support of functionality other than 3D rendering and larger support of functionality of 3D rendering than OpenGL. DirectX is mostly used for game development, as DirectX is supported in Xbox and Xbox 360 platforms. DirectX 11 is the latest version at the writing of this document. Direct3D was backward compatible all the way to version 10, but version 10 and later ones have only minimal backward compatibility, up to version 9. Direct3D has it's own shading language called HLSL which can perform vertex shaders, geometry shaders and pixel shaders.

DirectX's original logo was a modified radiation warning symbol and DirectX's original project name was 'Manhattan Project'. The creator of Microsoft's original DirectX specification, Alex St. John, said that the name 'Manhattan Project' was a reference to the original 'Manhattan Project', which was the original nuclear bombing of Japan. John said that it was a symbol for supersede Japanese video game makers, as they were dominant in the industry. This created some controversy, but it was publicly denied by Microsoft, which stated that the logo's design was just artistic. (Image 3.) (Wikipedia. Dir-ectX.)

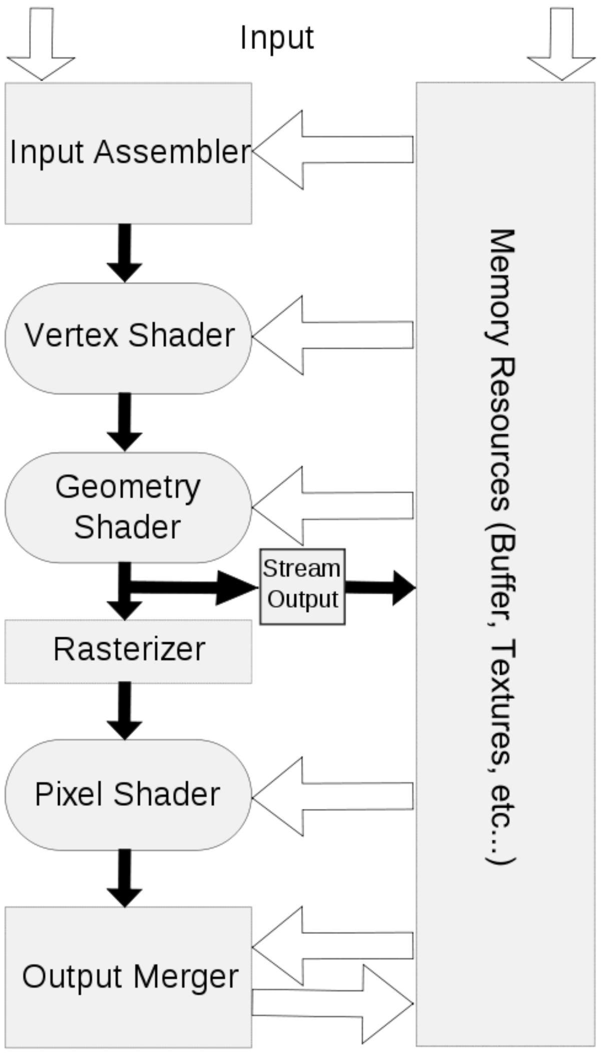

Image 2: DirectX pipeline. (Wikipedia. DirectX.)

Image 3: Old DirectX logo. (Wikipedia. DirectX.)

2 SDL

What is SDL?

SDL is a multimedia library. It is cross-platform and is written in C. It provides low-level functionality to handling keyboard, mouse, joystick, audio and 2D and 3D graphics. SDL works on over 13 platforms and SDL has been ported to over 23 programming languages. SDL has been used to port games to Linux, such as Doom 3, FreeSpace 2, Quake 4 and Unreal Tournament 2004. Like it's name suggests, it's a simple library and a great tool to start game programming with. Since SDL is licensed under GNU LGPL (Lesser General Public License), it's free to use and under certain terms, you can share or sell your program. (Simple DirectMedia Layer.) (Wikipedia. List of games using SDL.)

SDL's history

SDL was written by Sam Lantinga and SDL's first release was in the early 1998. While working for Loki Software, Sam got the idea for SDL while he was porting a Windows application to Macintosh. Sam used SDL to port the classic FPS (First Person Shooter) to BeOS (Be Operating System). The libraries SMPEG and OpenAL was developed to work with SDL. (Wikipedia. Simple DirectMedia Layer.)

2.1 Functionality

SDL's functionality is quite simple and is quite easily explained. As SDL's name suggests, SDL has been designed with layering in mind. (Image 4.)

Windows Management

SDL supports minimal window management, like changing the icon of the window, creating the window without frames and toggling fullscreen mode. (Simple DirectMedia Layer. SDL Library Documentation.)

Video

SDL's support for video functionality is greater than any other SDL-subsystem. SDL has data structures known as surface which represent images in SDL. Surfaces store pixel data, width, height and other data necessary for drawing the surface. SDL also supports rendering with OpenGL, which enables programmers to render 3D and access hardware acceleration. (Simple DirectMedia Layer. SDL Library Documentation.)

Audio

SDL's support for audio is minimal and simple. SDL can load and play only audio files which are WAV-format. (Simple DirectMedia Layer. SDL Library Documentation.)

Input

SDL can read input data from a mouse, a keyboard and gaming peripherals like joysticks. Joysticks have to be opened before any data can be read from them. SDL can read data from joysticks' axis', hats, balls and buttons. (Simple DirectMedia Layer. SDL Library Documentation.)

Events

SDL adds transpired events to the SDL's event queue, where the programmer can see what events have transpired and make the program act accordingly. Programmers can define events of their own, called user events, which the programmer adds to the event queue. (Simple DirectMedia Layer. SDL Library Documentation.)

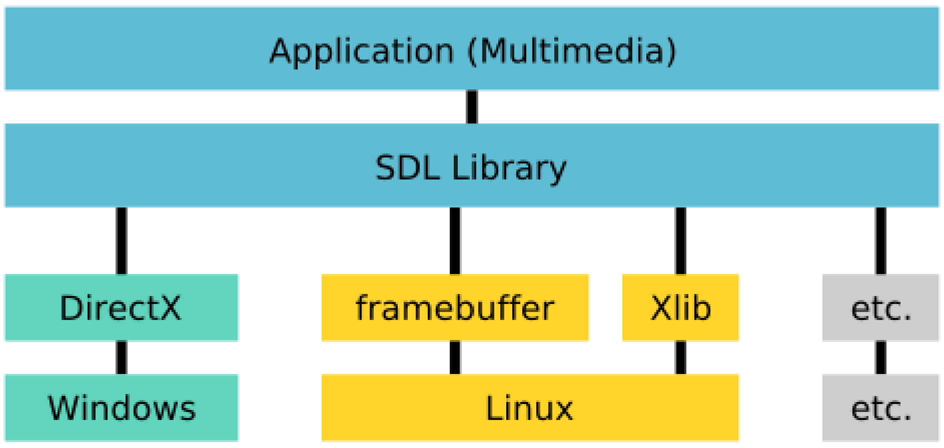

Image 4: SDL's abstraction layers.
(http://en.wikipedia.org/wiki/File:SDL_Layers.svg)

2.2 Similar Libraries

When comparing SDL with similar libraries, SDL is usually the underdog because of it's simplicity and lack of functionality. One could think "How SDL is so popular if it lacks almost all of the important functionality that many other libraries provide?". Because SDL is small and simple, it has a short and comprehensive documentation, SDL has many users around the internet, so one can get support very easily. SDL's simplicity also enables developers to build they're own systems around SDL.

Allegro

Probably the most similiar library to SDL in functionality, Allegro (Atari Low-LEvel Game ROutines) was originally programmed for the Atari ST in early 1990's by Shawn Hargreaves. Shawn turned away from Atari when the platform was taking it's final breaths, and Allegro was implemented for Borland C++ and DJGPP compilers in 1995. After combining ports and dropping support for some compilers, Allegro 4.0 was the first stable version to work on multiple systems. Although Shawn Hargreaves is no longer developing Allegro, an iPhone port is in development.

During the writing of this document, Allegro is available in two major versions: Allegro 4.4 and Allegro 5.0, the latter being a Release Candidate. Allegro 4 is licensed giftware and Allegro 5 is licensed with zlib/libpng license. Allegro version 5.0 is the main focus in this document. Allegro 5.0 is not backwards compatible with previous versions. Allegro 5 supports PC and Mac platforms and iPhone. Allegro 4.4 supports same platforms as 5, but also PSP. (Image 5.)

Allegro 5 supports configuration files, 2D graphics, OpenGL and Direct3D integration, input with keyboard, mouse and joystick, threading, timers, memory management and UTF-8 strings. (Wikipedia. Allegro.) (Allegro. Introduction.)

SFML

SFML (Simple and Fast Multimedia Library) is a multimedia library supporting many programming languages and a great range of functionality. The SFML website says "You can see it as a modern, object-oriented alternative to SDL.". SFML first appeared somewhere around 2008. SFML supports Microsoft Windows, Linux and Mac OS support is in development. SFML has native features that can only be used in SDL through addon libraries. SFML is available in C, C++ .NET (C# and others), Python, D and Ruby. During the writing of this document, SFML 1.6 is the latest version of SFML. (Image 5.)

SFML supports Unicode, OpenGL interfacing (SFML uses OpenGL for 2D drawing), input with mouse, keyboard and joystick, shaders, multiple image formats, multiple font file formats, multiple audio formats, 3D sound, audio capturing, audio streaming and networking. SFML has been separated into following packages: Audio, Graphics, Network, System, Window. These packages can work independently.

Audio includes functionality for loading audio from file, audio playback, 3D audio playback, audio capturing, audio streaming.

Graphics includes functionality for loading and saving image files, loading font files, rendering images and rendering font. Network includes functionality for TCP and UDP sockets and handling HTTP and FTP traffic. System includes functionality for time measurement, threading and unicode conversions. Window includes functionality for creating and handling windows and handling input data. SFML has been built upon other libraries like GLEW, libpng and freetype. SFML is licensed under zlib/png license. (Wikipedia. SFML.) (SFML. Features.)

ClanLib

A SDK dedicated to game development, ClanLib was first released to public in 1999. ClanLib supports many features, even going as far to have functionality for databases via SQLite. During the writing of this document, ClanLib 2.2.5 is the latest version. Out of the four libraries in the comparison, ClanLib is the most game-oriented library, offering functionality such as collision detection and a networking engine called NetGame. ClanLib also offers built-in GUI complete with components. There are even some exotif features like XML and CSS, even DNS and regular expressions are supported. During the writing of this document, ClanLib's documentation is quite poor. ClanLib's online documentation is incomplete and only samples and examples are included with the SDK.

ClanLib supports memory management, threading, shaders, OpenGL integration, collision detection, networking, sounds and some exotic features like XML, CSS, DNS and regular expressions. ClanLib supports OpenGL 3 and Direct3D 10. (Image 5.) ClanLib is licensed under ClanLib license. (Wikipedia. ClanLib.) (ClanLib. Main Page.)

	SDL	Allegro	ClanLib	SFML
Languages				
C / C++	✓	✓	✓	✓
D	✓	✓	✗	✓
.NET (C#)	✓	✗	✗	✓
Python	✓	✗	✗	✓
Ruby	✓	✗	✗	✓
Platforms				
Windows	✓	✓	✓	✓
Unix/Linux	✓	✓	✓	✓
Mac OS X	✓	✓	✓	✓
iPhone	✗	✓	✗	✗
Rendering				
OpenGL integration	✓	✓	✓	✓
Direct3D integration	✗	✓	✓	✗
Font rendering	╲	╲	✓	✓
File formats				
Multiple image formats	╲	✓	✓	✓
Multiple audio formats	╲	╲	✓	✓
Functionality				
Threading	✓	✓	✓	✓
Networking	╲	✗	✓	✓
Scripting	✗	✗	✗	✗
Math	✗	✓	✓	✓
Audio	╲	╲	✓	✓
Licensing				
Commercial use	✓	✓	✓	✓
Open source	✓	✓	✓	✓

✓	Feature is included
✗	Feature is not included
╲	Feature is included with addon

Image 5. Comparison chart.

2.3 SDL Applications

All of these applications are free and open source.

DOSBox

DOSBox is a software that emulates Intel x86 PC. DOSBox supports emulating sounds, graphics, mouse, joystick and modem. DOSBox is designed to run old MS-DOS games that current systems do not support any longer. In theory, DOSBox can run any MS-DOS applications and it has been optimized for emulating games, but DOSBox is able to run, for example, Microsoft Windows 3.1. DOSBox has been used commercially by Good old Games, id Software, Lucasarts and Activision Blizzard, many of these companies use DOSBox to rerelease their old gaming classics. DOSBox has been licensed under the GPL and is being actively developed by four developers.

The DOSBox version analyzed in this document is 0.74. When building DOSBox from source, the only external library needed is SDL, every other library is optional. Other libraries used are: Curses, for debugging. Libpng, for screenshots. Zlib, needed by libpng. SDL_Net, needed for modem/ipx networking. SDL_Sound, needed for compressed for audio on diskimages. ALSA_Headers, needed for Alsa support on Linux. DOSBox uses a slightly modified DLL of SDL, the DLL was modified to use DirectX over Windib and to check if the GPU can store surfaces on GPU's memory. Users of DOSBox are able to record movies of their games, as DOSBox uses zMotion to encode the video. If someone wants to play the recorded video, it is necessary to install a codec, which is included in (at least) the Windows version of DOSBox. (Wikipedia. DOSBox.)

FreeCiv

FreeCiv is a game, in which the player(s) are the leaders of civilizations. The object is to defeat the others and become the greatest civilization, one way or the other. The games is heavily influenced by Sid Meier's Civilizations game series, from which the name is derived from. FreeCiv is a big project and a few hundred people has been working on it through it's lifespan.

The FreeCiv version analyzed in this document is 2.2.4. FreeCiv supports multiple window managing/graphics rendering libraries like SDL, GTK+ 2.0, Xaw and Window's natice windowing library Win32. When building the client with SDL, besides the SDL library you also need to compile SDL_image for loading PNG files and freetype for rendering text from TrueType fonts. FreeCiv does not rely on external software like Windows' registry, so to play the game, only the executable and the data files are necessary. This means that FreeCiv can be played from removable medias, like USB sticks, and users can have the executables for all platforms with them in the same directory, as these executables handle the save games and data files all the same. (FreeCiv. Main page.) (Wikipedia. FreeCiv.)

StepMania

StepMania is a rhythm game, where the player(s) dance to the game's songs with dance mats. The songs are accompanied with rising arrows and the players has to touch their dance mats' according arrows at a specific time. The idea is to simulate dancing with touching the dance mats' arrows with your feet. StepMania is the most popular dancing game on the PC and the commercial game In The Groove uses StepMania's source code. StepMania's source code was re-licensed from GPL to MIT to allow In The Groove to remain proprietary. In The Groove was developed by the core StepMania developers.

The StepMania version analyzed in this document is 3.9. Besides SDL, StepMania uses zlib, ffmped, libjpeg, libpng, libmad and a few others. These libraries are included in the SVN repository, so when a copy of the repository is downloaded, the user can build the game without downloading and configuring any additional libraries. StepMania uses code from Xpaint, an open source image editor for Unix-like operating systems and as absurd it may seem, PuTTy, an SSH/Telnet client.

The code from Xpaint is used to load GIFs and the code from PuTTy is used for crypting. StepMania is licensed under the MIT license. (Wikipedia. StepMania.)

2.4 Future

During the writing of this document, the latest version of SDL is 1.2.14 and 1.3 is under development but available in SDL's version control repository. SDL 1.3 will bring new features to SDL, like support for OpenGL 3.0 and higher, multiple windows, multiple displays, multiple mices and tablets and force feedback. SDL 1.3 will also support iPhone, iPod Touch and Nintendo DS platforms. There will also be two licenses available for SDL, LGPL and Commercial. With LGPL license you get the same features as with SDL 1.2 and with SDL 1.3 you will get the possibility to statically link SDL, iPhone/iPod development, Embedded Systems development and personal e-mail support. There is no release date for SDL 1.3.

SDL 1.3 might also have support for android also, because during the writing of this document, there is documentation for building an android application in SDL 1.3 repository. It is unclear wheter you need a commercial license to build android apps or is the LGPL license enough. (Galaxy Gameworks. Home.)

3 GAME ENGINE

Defining a game engine is not easy, because many platforms and tools (mainly programming libraries) that have functionality related to video games may be called a game engine. Engines might have functionality spanning over multiple areas (Unreal Engine) or focus only on one specific area (OGRE).

Using third-party game engines has gained more popularity due time and some companies focus only on game engine development and not on actual games, although most popular commercial engines have started being an engine for a certain game or a series of games. As game engines need to support larger range of functionality, many companies use third-party engines as it saves time and money from the actual engine development. Using third-party engines has it downsides too, as developers need to learn how to use the engine and experiment what it can and can't do. In the book "Postmortems From Game Developers", Warren Spector tells about the development process of Deus Ex and says: "However, to my surprise, licensing technology didn’t save us all the time I’d hoped it would. You’d think cutting a year or more of engine-creation off a schedule would result in an earlier release date. On DEUS EX, that didn’t prove to be the case. Time that would have been lost creating tools was lost instead to learning the limitations and capabilities of “foreign” technology." (Postmortems from Game Developers, page 202.)

For a student or a beginner programmer/developer, it might or might not be a good thing to create a completely new engine from scratch. Naturally these new engines might never reach the quality and range of functionality as those engines that have been around for decades, but the engine creation process might provide valuable learning and experience on a 'lower-level' than actual game development. Most engines support rendering and animating on both 2D and 3D, networking and physics among others, so the developer(s) have to study at least some of these areas. More and more people use ready-made engines and more and more schools teach using ready-made engines, but there must always be people to create and maintain these engines. (Wikipedia. Game engine.)

3.1 Functionality

A game engine can be compared to a car engine, it takes the driver from point A to point B and the driver does not have to know how the engine works as long as it does what the driver wants. A game engine gives generic gaming functionality to a programmer, thus making game development process a little easier for the programmer.

Stefan Zerbst has said in the book 3D Game Engine Programming "One of my favorite topics you can start lots of flame wars about is whether to use already existing helper libraries or to reinvent lots of wheels with your own implementations. I will walk the hardest way I can see in this book, following my motto: Only those who have been in Hell can see Heaven." (3D Game Engine Programming, page 28)

Game engines usually provide functionality independent of the actual game and commercial ones often come with external or integrated tools.

"An engine should be independent of the project and capable of working with other video game projects and non-game multimedia software projects without the need to modify the engine's code. In other words: The engine's code must not include any game-related code and it has to be reusable." (3D Game Engine Programming, page 6)

A.I.

A.I's role has become more and more prominent, as most shooter games rely quite heavily on it. Most game engines support A.I. functionality for shooters, such as pathfinding and line of sight and/or field of vision. A.I. is a huge and vast area of theory in gaming and otherwise, thus it is hard and maybe even unnecessary to implement generic A.I. that could be used in every possible game. Usually is is a better choice to create an A.I. from scratch for most games. Almost every gaming genre uses different kind of A.I.

Graphics

Probably the most important aspect of all games is the ability to render graphics. Whether the game's graphics consist of only text or latest state of the art 3D worlds, all games render some kind of graphics. Because most of todays biggest and popular games' biggest selling points is cutting-edge graphics, the better graphics a game engine can render, the more popular it will become. Although most game engines also have other functionality, some engines focus only on rendering graphics and are usually called graphics engines, rendering engines or 3D engines.

Localization

Although english is one of the most used languages in the world, games gain a wider audience if they are localized to the areas' primary language(s). It is much easier to localize only the text portions of the game, but some games localized audio for major languages.

Networking

As multiplayer is one of the most prominent features in gaming today, this functionality is expected from any professional engine, propietary or not. Networking is an area where a lot of attention should be payed, as networking usually occurs among multiple machines and the possibility for errors is much larger compared to graphics rendering, where (usually) all the calculations are done on one machine. A working networking functionality is especially important in MMO games.

Physics

Ragdoll physics was introduced to gaming in 1998 in Jurassic Park: Trespasser. There are four main approaches to ragdoll physics: verlet integration, inverse kinematics, blended ragdoll and procedural animation. Ragdoll physics have come to be a prominent part of many games, especially FPS'. There are many physics engines (or simulators) for both 2D and 3D worlds, probably the most popular being Havok. There are many open-source options too, like Bullet.

Scripting

Scripting is used to affect a games functionality without actually needing to compile the game every time a new script is applied. This allows the developers to change gaming functionality every time the game is restarted and sometimes on the fly, depending on the script. Scripting is mostly used for NPC A.I. and affecting the game world.

Sound

Sound effects have been with games for a long time, from the earliest beeps to the MIDI songs and all the way to the latest orcestral masterpieces and realistic gun sounds. Most popular game engines support audio playback and features like 3D sound positioning and audio effects have become more popular. (Wikipedia. Game engine.)

3.2 Existing Engines

Propietary

The biggest, most powerful and most popular engines around are propietary. Having the expertise of an experienced team of game industry veterans, propietary engines give users functionality that open-source engines must work hard to come around.

Source

Developed by Valve, Source gained popularity in 2004 powering the massive hit known as Half-Life 2, Source is among the biggest commercial game engines out there. Although it has not been used in as many commercial games as Unreal Engine or even id Tech engines, Source engine has been heavily used in creating mods. Half-Life 2 was not the first game to use Source though, as Troika Games' Vampire: The Masquerade - Bloodlines, was finished before Half-Life 2, but Valve did not want another game to be released before Half-Life 2, Vampire's release was delayed until Half-Life 2 was released. Vampire was not graphically as beautiful and cutting edge than Half-Life 2, as can be seen. (Image 6.) Then again, the two games have completely different game worlds and development teams behind them.

Source might be considered a distant descendant of id Tech engines, as Source is based on GoldSrc (the engine powering Half-Life 1) which is based on id Tech 2. John Carmack, creator of id Tech engines, said in his blog that "There are still bits of early Quake code in Half Life 2." (http://www.armadilloaerospace.com/n.x/johnc/recent%20updates/archive?news_id=290) (Source. Wikipedia.)

Image 6: Vampire: the Masquerade – Bloodlines (Gamesradar)

Unity

Unity is a fairly new but really popular in the area of game engines. Although, Unity is not just an engine, it is also development tools for the engine. Most of the development for Unity happens with the scene editor (Image 7.), which has a multitude of tools for handling scenes, models, assets and more. Unity's development began in 2001 and the first version, Unity 1, was released af Apple's Worldwide Developer Conference in 2005. Unity can be divided to two parts: player and editor.

The editor is where the actual development happens and the actual application will be deployed from. The player runs the finished application. The editor runs only on Windows and Mac OS X, but the editor runs also on consoles, mobile platforms and browsers. Unity is quite similar to Blender game engine, Torque Game Builder and Gamestudio as the graphical environment is the main development application. (Unity. UnityScript Introduction.)

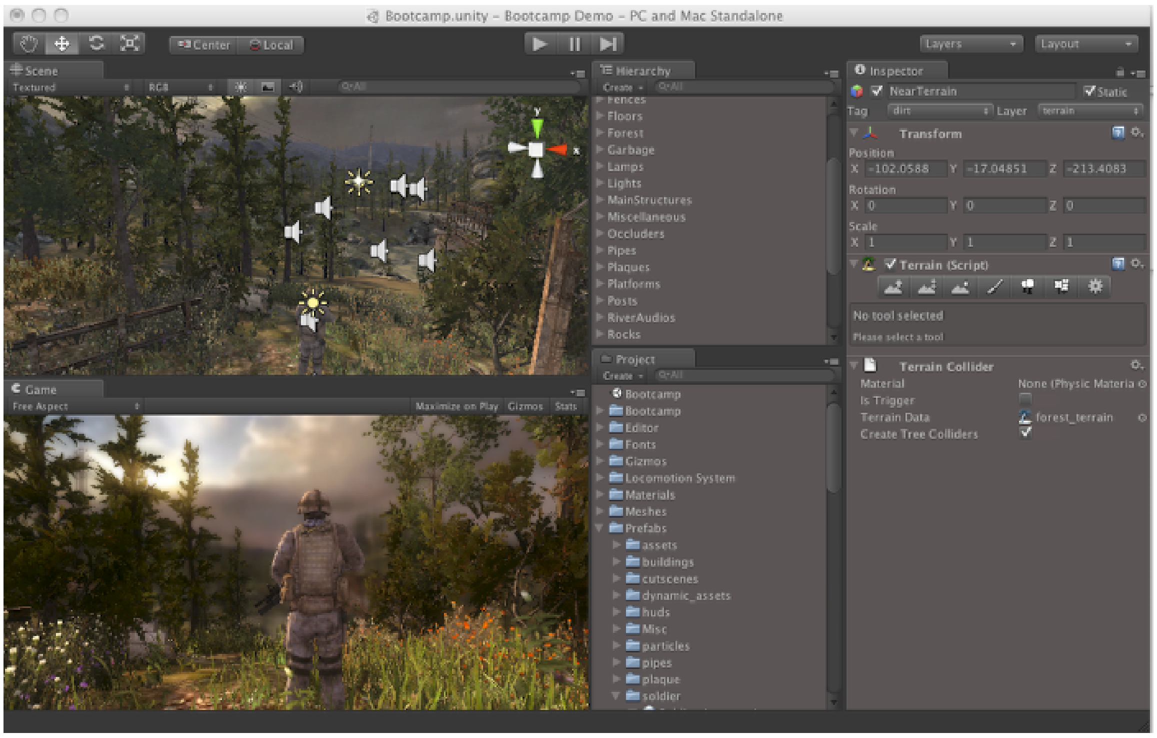

Image 7: Unity's editor. (UnityScript)

Unreal Engine

Unreal Engine is one of the biggest and most popular game engine in the world. Originally created for the first Unreal game by Epic Games, the engine has been around the industry since 1998, the release of Unreal. The engine has spanned for three major versions: Unreal Engine 1, 2 and 3. Epic Games used the first engine for Unreal and Unreal Tournament, but the engine was licensed for many other games, probably most popular being Deus Ex. Unreal Engine 2 was introduced with a game called America's Army, which was not developed by Epic Games. This next version was used in Epic Games' developerd Unreal Championship 1 and 2, Unreal Tournament 2003 and 2004 and many other games by other developers. Unreal Engine 2 has been used in over 70 games.

The latest version, Unreal Engine 3, adding support for consoles, made it's debut in 2006 with Epic Games' Gears of War. Unreal Engine 3 has been used for over five years, in almost 100 games, including Gears of War 2 and Unreal Tournament 3. Unreal Engine has come a long way in 13 years. (Image 8.)

Unreal Engine
1
2
3
Image 8: Unreal engine model comparison. (Wikipedia)

The Unreal Development Kit is free for free games and commercial games that do not sell more than a pre-set amount of money. Anyone can use UDK and create games with it, share them for free or if feeling confident, license the engine and sell the game. This allows for starting, or old, companies to create a whole game with UDK before actually licensing the engine. This is also a good way for Epic Games to get more people to use UDK and create games with it, as schools start to use it more since it is a free and renowned game development tool. (Wikipedia. Unreal Engine.)

Open-source

Some of these engines are not game engines per se, as they don't have any game functionality like sounds or networking, but they are on this list as they are used in developing games and make a great base for a game engine. They do provide one of the most important part for most games: rendering graphics.

id Tech

When id Software released Doom in 1993, little did they know that it would become one of the best known game series in the world and little did they know that the engine used in Doom would become one of the oldest, powerful and popular game engine families in the world. id Tech engines have been around for almost two decades and has been used in over 40 games. The id Tech engines have not been as popular in licensing to third-parties as, for example, the Unreal Engine, but id Tech engines have been known for being waymakers, especially in 3D rendering. id Tech 1 paved the way for first-person-shooters, and 3D games altogether, as it was one of the first engines to support 3D-like rendering. Hexen was one of the most popular games made with id Tech 1. (Image 9.)

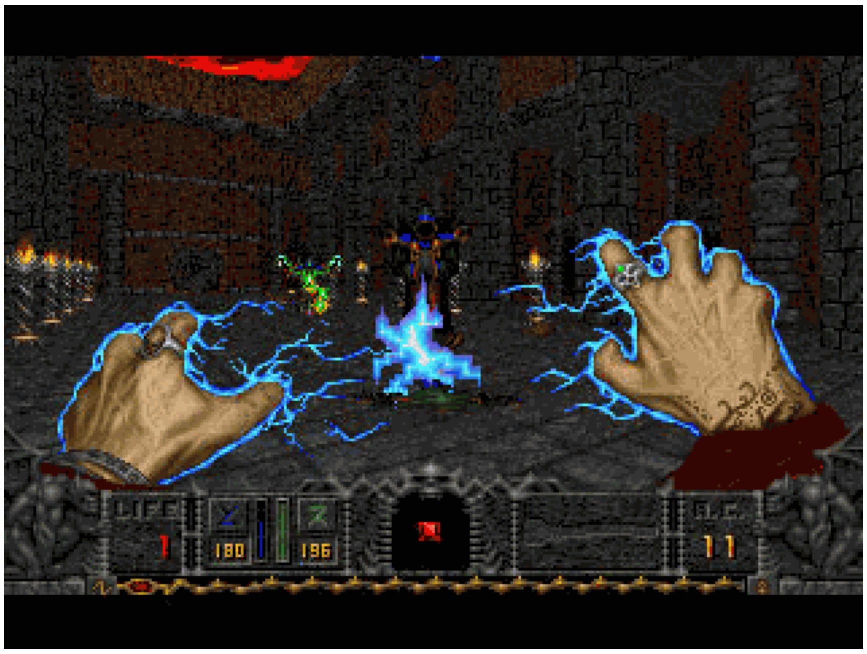

Image 9: Hexen, 1995. (Svatopluk.com)

It was not true 3D though, as it was not possible to look up and down properly and two sectors can't be placed above and under each other, allowing only a kind of pseudo-3D rendering. It was the next 'evolution' of the engine, id Tech 2, that allowed true 3D rendering. Again, pushing the shooter-genre and the world of 3D forward, id Tech 2 (originally known as Quake Engine) came along with Quake and dethroned other FPS titles and revolutionized the way multiplayer games were developed. From there on, id Tech engines have pushed the technology onward and have come a long way since 1993, as can be seen from preview material of Rage. (Image 10.) (Wikipedia. id Tech.)

Image 10: Rage, 2011. (Grasz.org)

OGRE

OGRE is a 3D rendering engine. OGRE's development began in 2000 by Steve 'sinbad' Streeting and OGRE was first released in 2005 under the LGPL license, but was later changed to MIT license. It has no built-in game functionality, but it can be integrated in to a custom game engine with ease. OGRE is licensed under MIT license and can be used for free and commercial applications, although the OGRE license has to be included in the distribution of the application. There is no need to display OGRE's logo in the application or tell the user that the application has been made with OGRE, besides the license text. OGRE differs from most open-source engines in the way that it has been used in commercial games like Ankh, Jack Keane, Torchlight and the upcoming Steam Slugs. Torchlight and Steam Slugs are excellent examples of OGRE's rendering capabilities and based on Steam Slugs' preview material, the game seems to be quite beautiful. (Image 11.) (OGRE. Features.) (Ogre3D. Wikipedia.)

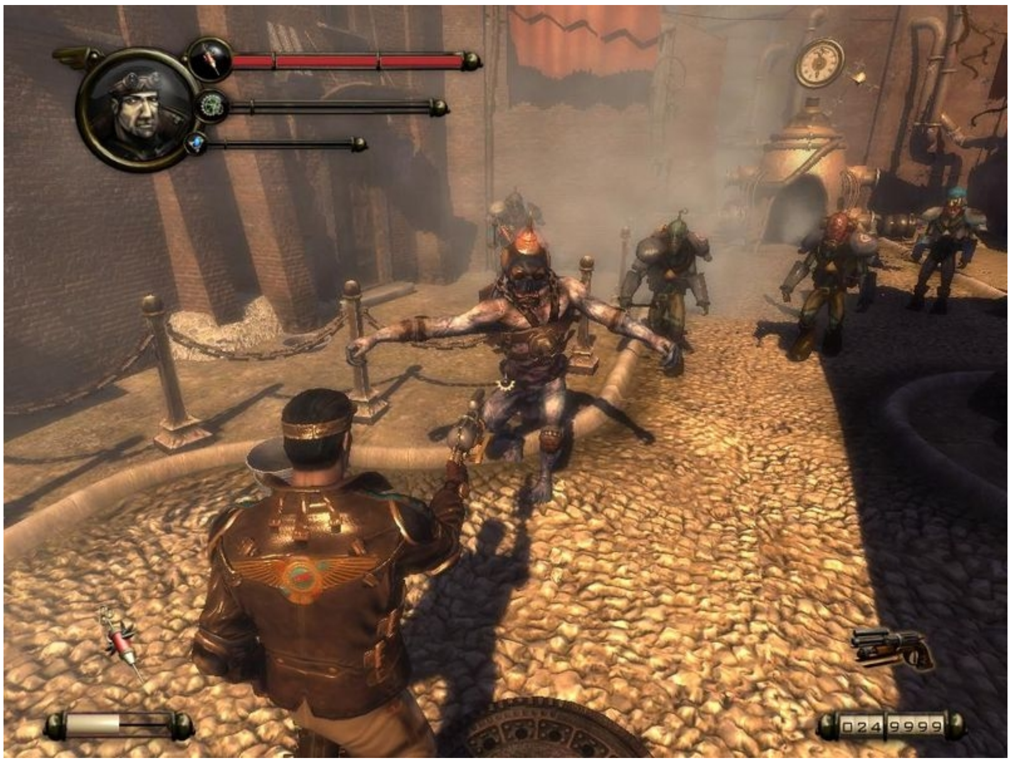
Image 11: Steam Slugs. (Ogre3D.org)

4 ROBODASH PROJECT

The Project

The project consisted of four members:

- Otto Hietala, programming, game engine and gameplay
- Toni Räsänen, programming, level generation and gameplay
- Sasu Kemppainen, Design and project management, sound effects
- Markku Wiik, Graphics

The purpose of the project was to create a video game within three months. The game and every team member's participation in it was designed so that everyone would work in area that would apply to their theses. The game's initial design was created during the beginning of the project, mostly the first month, but the design lived through changes throughout the project's duration.

Several software tools were used during the project. Subversion was used for revision control and as a central database for necessary files. MediaWiki was used for documentation, calendar management, bug tracking and project management. Wordpress was used for blogging and it was mainly maintained and updated by the project manager. The Subversion repository and Wordpress was located in SourceForge and MediaWiki was in one of the project member's server for easier configuration and addon functionality.

4.1 Design

RoboDash is an infinite platformer game, which means that the game world is created (and destroyed) with player's progress through the game. The only way for the game to end is with player character's death. The game levels and enemies would be generated with Procedural Content Generation.

The player character can be modified during gameplay. If the player finds new parts for the character, those parts can be changed with the existing ones to make the character perform different. The player can change character's legs for making jumping and running different, hands for weaponry and a so called "body battery", which is a special battery found at the torso of the player character. The body batteries have varying effects, from health icrease to weight increase/decrease. The character's hands are controlled independently, so that the player can have different combinations of weapons for different situations and different kind of gameplay.

During the design, one of the technical problems noted would be memory management. When would game objects become inadequate and should be purge from the memory. It was decided, that the game would generate levels of certain size and in between these levels would be a transition when the new level would be generated and the old one would be purged. When this transition would have been passed, access to the old level would be impossible. It would also be impossible to stop this transition when it had been begun. The project never reached the point where memory management would have become an issue and memory management was not designed in the initial designing process. Regrettably, this memory management was never designed or implemented in to the game.

4.2 Game story

The game story is from the project's wiki, which is not available to public.

In the year 2249 in an orbiting space station, a 120 years old scientist named Sarah Cristini had become dangerously obsessed with the concept of immortality. The technology of the day had allowed humans to expand their life but Cristini has just recently reached her generation's life expectancy. The mainstream science had decreed the human brain incapable to handle extremely long periods of time, purging the brain regularly may allow one to live forever but it would be more like being born again every now and then as a completely different individual.

Against her better judgment, Cristini begun building an android body for herself, modeled after her perfect vision of herself. She planned to use old theories of mind planting to continue living in the new perfect body. She tested the body first with the help of an A.I. and named the new experimental entity Sarai. The mind planting experiment didn't go as planned. Instead of moving her mind to the new body, Clarke managed to only copy her brain in digital form and survive the process. She effectively copied herself and was left inside her old dying body. Disappointed and convinced that whatever she did, the new entity would just be a mere copy, Clarke wanted to get rid of the new being. She shot Sarai in the head and dumped her in space.

Years later, Sarai's mind miraculously woke up. She had somehow managed to end up on an unknown planet. The A.I. still inside her took over for the features lost when being shot and landing on the planet. Her mind a strange meld of her original creator and an A.I. she began her life's work: trying to emulate her creator. Hundreds of years pass. Sarai has managed to build a mobile lab from wreckage found on the planet. Her mind continues to deteriorate year after year. She has reached the point that every single day she puts together robots, aiming for that one perfect bot that can continue her own existence. Her supplies and brainpower are limited and this twisted copy of her creator's obsessions keeps churning out stranger and stranger little robots, every single one of which she throws away disappointed.

The planet used to be inhabited by deeply religious humanoid people. As with many religions, theirs had a lot to do with immortality, which parallels Cristini's scientific pursuits for immortal life. Many shrines devoted to worshiping immortality and infiniteness resemble different symbols that are meant to evoke the feeling of endlessness or immortality (Ankh, Ouroboros...)

Scientific pursuits had mostly taken a backseat to the religion. Virtually everyone on the planet considered the universe as an infinite collection of flat surfaces, all stacked on each other. When a creature dies, their spirit ascends to the upper level and becomes a new life there, forgetting some of its earlier lives but retaining its characteristics. The clouds above the planet were seen as the next level and the surface was seen as the current level of existence. Visiting other levels while occupying the body meant for another level was considered highly improper, albeit interesting notion for many. Building very high towers or burrowing deep into the ground was forbidden and frowned upon.

When Sarai landed on the planet, the civilization was on par with earth in the 19th century. People saw her as someone who had fallen from the upper level of existence, owing to the fact that she really did come from the sky and looked very different than the inhabitants of the planet. She was seen as a celestial being and respected, even if most people frowned upon her for exploring the levels she should not explore. A small group of free thinkers with more or less scientific minds became her disciples, obeying her orders while hoping she would answer their questions about life beyond death. They never really did get their answers but managed to provide Sarai with enough raw materials to put together her mobile lab and start her robot production. Soon enough the new metallic beings started forming from the junk left behind by Sarai's lab experiments.

The original inhabitants tried to coexist with the new metallic creatures at first, but eventually the robots made living harder and harder. They had the same effect than bringing some European animals to Australia: they overthrew the original inhabitants, even if they didn't mean to do it. No one really dared to go against the robots, knowing they were the children of a celestial being. The vast amounts of robots took away the living space and trampled the crops. After a few short centuries, the organic beings we're no more and the planet was completely inhabited by metallic creatures trying to mimic human behavior and forming their own weird societies.

Content generation

The game world, player, enemies, weapons and more was designed to be procedurally generated, but in the current incarnation, only game world is generated procedurally. The game world is generated as the player goes forward in the game world, which presented a problem: memory management. Because the player's computer would eventually run out of memory if the world would only be created but never destroyed, it was decided that the game world would have junction points. When the player passes these junction points, the world behind would be destroyed and the world after would be created. These junction points would be generated so that the player could not go back once it has been passed.

4.3 Engine

Engine requirements

At the beginning of the project, RoboDash engine was designed to have the following requirements:

- Render 2D images with scaling and rotation support, since the game is 2D platformer, basic 2D rendering was necessary, scaling and rotation could be used for animating specifig sequences for the enemies and the player.

- Audio playback with support for at least mp3 and wav files, the game was designed to have sound effects and background music. Mp3 and wav are popular and widespread formats, so the game should support at least them. Since SDL_mixer could only playback sound effects as wav-format, it was necessary to have that support. Luckily, SDL_mixer supports loading all formats it can playback, so no extra work was done in loading the audio files. Ogg was a secondary choice for music format.

- Physics simulation (includes collision detection and such). This is necessary for most games. Using a third-party physics simulator would save the team time and effort and would allow the team to create cool physics-enabled objects and effects in the game world, while having all the physics-functionality for a basic platformer. It did not go as planned.

- Input reading and handling from keyboard. Every game needs to read some sort of input from the player and keyboard was the main input-device in this game. Mouse would be used for menus and possibly in the future the game would support joypads.

- Visual effects like gradient shading and depth blur. When the player would shoot an especially powerful weapon or something in the game would explode, cool effects would make a big difference. Although the game relies more on gameplay than graphics, the game should still be visually appealing.

Most of these requirements were met, only the visual effects were not implemented, because time and effort should be directed on other parts of the engine and the game. Effects could be created with basic sprite rendering and animation, but there is no support for effects in the engine. The physics module was an issue, although it was implemented in the engine, it was not implemented in the later versions of the game for it's complexity. During development, the engine did not get many new requirements, only the custom physics module for collision detection. It was implemented as a separate module and is fairly simple to use, as it was to develop.

SDL and it's addon libraries were chosen to provide functionality, for window management, input, audio, etc. OpenGL was used for the actual rendering, as it is faster and more powerful than SDL's default rendering methods. The game needed 2D physics simulation and the project's both programmers suggested Box2D. Box2D was chosen, because it is probably the most well-known 2D physics library and both programmers did not come up with a better solution for 2D physics, at the time. Box2D's reputation for being stable and working library was a decisive factor, plus it has been used in games like Crayon Physics Deluxe and Angry Birds.

Engine design

The engine was designed to give basic functionality for a platformer game. The focus on this engine was in 2D graphics rendering, text rendering, audio playback and physics. It can be used in other games that need this kind of functionality, platformers or not.

Modules

Every module has a main source code file that uses the actual module. Most modules are designed to work independently, but Text-module depends on Graphics-module. Some functionality is outside of the modules and in the actual Engine-class, like FPS limiting. (Image 12.)

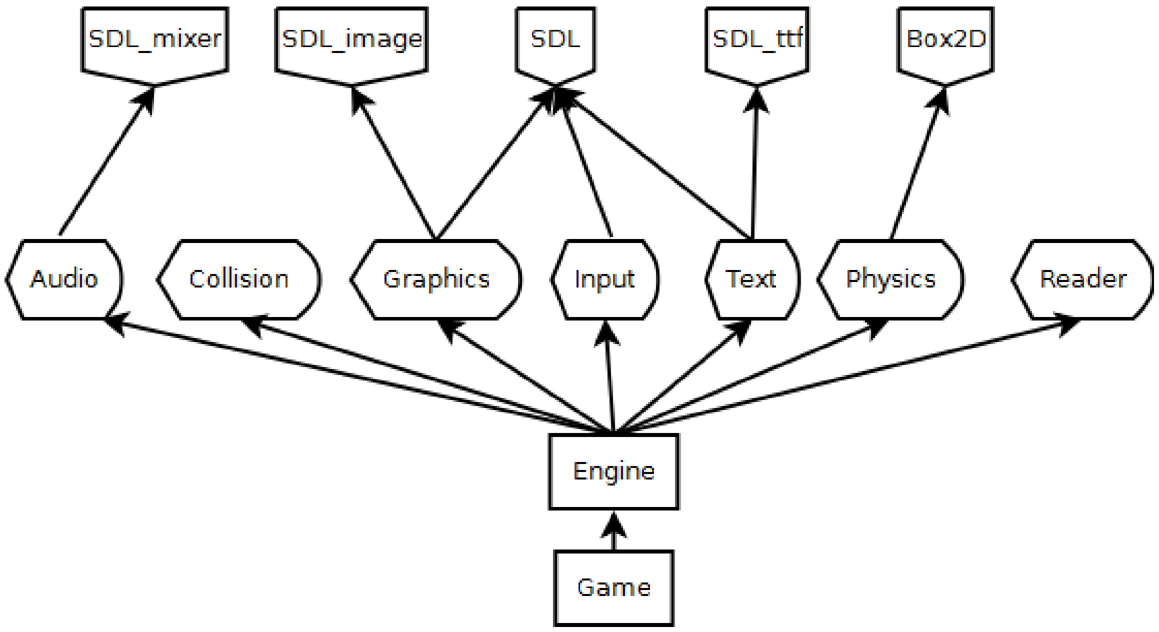

Image 12: Engine dependencies.

Audio

Audio-module handles loading music and sound effects and their playback. This module uses SDL_mixer library and thus supports the same file formats. The module has two classes: Sound and Music. Music is an audio clip that plays in the background, usually constantly, and any supported file can be loaded as music. Sounds are clips that are short and can be played over and over again, only wav-files can be loaded as sound-clips. These file restrictions are defined in SDL_mixer. (Image 13.)

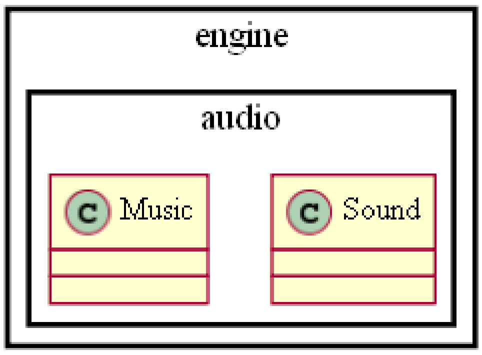

Image 13: Audio module.

Collision

Collision-module handles basic collision detection between two objects. This module was created when it was realised that Physics-module and Box2D would be too complex to use and adapt in to the game. This module uses separating axis theorem and it can only check collision for two objects at once, also it doesn't support broad phase testing. It checks for fast moving objects automatically though.

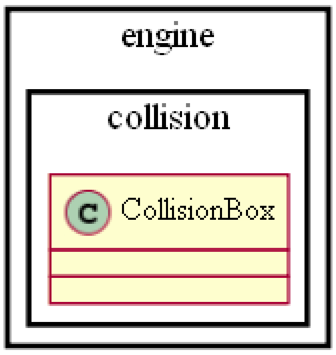

Image 14: Collision module.

Graphics

Graphics-module takes care of image loading, image rendering, animation handling and spritebatch rendering. This is probably the biggest module in the engine. This module creates a window using SDL and SDL procudes an OpenGL rendering context on it. After that, the user can load images and animations and render them on the window's context. This module uses SDL and SDL_image libraries and thus supports the same file formats as those libraries. (Image 15.)

Animation-class

Holds multiple Frame-classes which are rendered on the specified position on the screen, each after another. Frames are used instead of Images so that parts of the same image can be rendered without having the image in memory multiple times.

Color-class

Holds data for a RGB color value. Is used for screen background color. The engine never got to a point where this class could be used properly.

Frame-class

Frame is a simple class that holds a reference to an Image-class and some additional data like position, render border (which part of the image is rendered) and rotation. Multiple Frames can reference to the same Image without actually having the same image in memory multiple times.

Image-class

Holds actual image data. Can be rendered independently or through Frame-class which is more dynamic and recommended for most cases.

SpriteBatch-class

Is used for rendering parts of an image to multiple positions on the screen as a batch. Can make rendering faster, especially if the same part of an image, or multiple parts of the same image, is rendered on the screen.

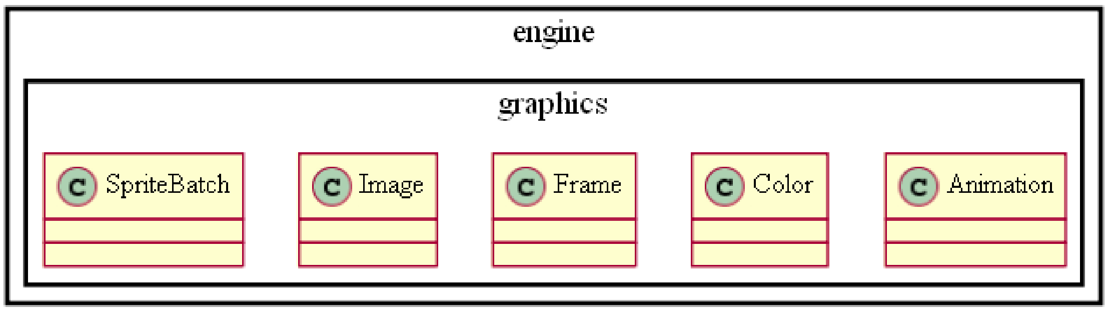

Image 15: Graphics module.

Input

This module reads input from mouse- and keyboard-devices. It uses SDL and is probably the most simple module in the engine. (Image 16.)

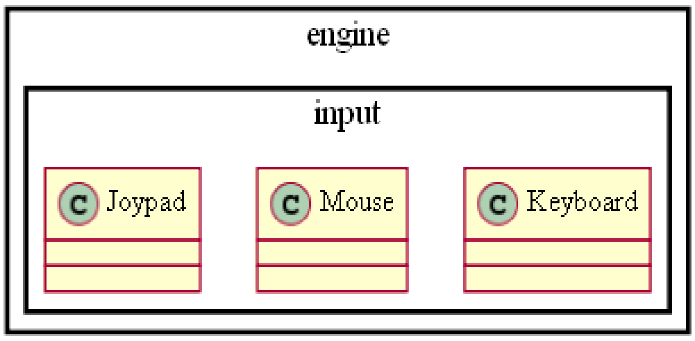

Image 16: Input module.

Physics

This module uses Box2D to simulate 2D physics, RoboDash was originally supposed to use physics to simulate movement, gravitation and collision between objects, such as player, enemies and random game world objects such as rocks and barrels. This module was scrapped and it's development seized after it was decided that it was too much of a hassle for the project to start configuring Box2D and it would be easier to make a simpler physics from scratch. Though the module still exists and works in the engine. (Image 17.)

The module simulates a world and three different kind of physic boxes in it. StaticBoxes are not affected by forces, can only be moved manually, have zero mass and they collide only with dynamicboxes. KinematicBoxes do not repond to forces, can be moved with velocity and collides only with dynamicboxes. DynamicBoxes are fully simulated, can be moved by forces and velocity, has finite mass and collides with all types of boxes.

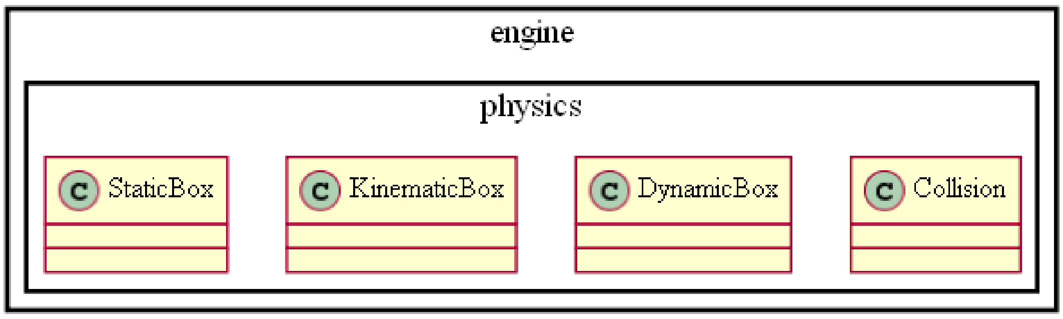

Image 17: Physics module.

Reader

Reader-module handles reading data from configuration with a certain format. The format includes to data items: key and value. Both of these items must be embedded in double quotes and key/value pair must be separated by an equal mark. Examples:

```
"Key" = "Value"      <- Correct

Key" = "Value        <- Incorrect

Key = Value          <- Incorrect

"""" = """"          <- Correct

"" = ""              <- Correct
```

If a line in the settings-file doesn't start with a double quote, the line is processed as a comment and is not handled in any way. This allows developers to create notes inside the settings-file. (Image 18.)

Text

Text-module loads and renders TrueType-fonts. This module depends on Graphics-module, as it uses it's low-level SDL and rendering functionality to render text on the screen. This module uses SDL and SDL_ttf libraries. (Image 19.)

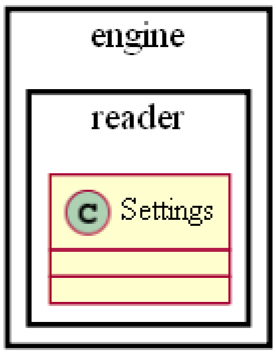

Image 18: Reader module.

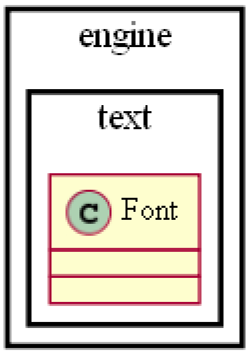

Image 19: Text module.

Problems and solutions

The biggest problem in the development of the engine and the game was the physics module. At first it was a good idea to implement a realistic physics simulations into the game, because it would handle player and enemy movement, falling objects, explosion blasts, etc. When the module was implemented to the engine, the team noticed that setting correct physic properties for the player and enemies was a lot harder than making a module for basic physic simulation from scratch. The team could've used the physics-module, but player and enemy movement would've been hard to get to work and would have not followed the design. The physics-module was forgotten and a custom module was used instead.

What went wrong

RoboDash was not finished in the six months that was the original deadline. Mainly because the project was too big for too few people with not so much skills in game development. The project's original deadline of three months would have reached only if the game was small enough, so either the game's design bloated in the beginning of the development, or the deadline was too short to begin with. In the area of programming, the workload was not divided properly, as there should've been one responsibility for each programmer.

If there would've been one more programmer who's main responsibility would've been gameplay, the two other programmers could have concentrated on the engine and content generation. Also, no one in the team had any actual experience in working on an actual game project. The team had been in classes about project management, but no one had any concrete experience from working on a project, especially through the whole lifecycle of a project.

What went right

The team was motivated, especially in the beginning. Everyone was involved in the design process from the start, which allowed everyone to give their input to the games development process on all stages of development. In the middle of the project's development cycle, when Box2D's complexity was noted, some were reluctant to make a custom physics simulator from scratch (even though it would be really simple), but later on everyone agreed that a change was needed. Everyone was motivated for most of the project, although when nearing the end and realising that the game would not be finished in time, people started to lose interest and will to continue the project.

How it could be improved

The team should have more members, at least one programmer. No area in the project should be considered as 'someone will work on it when they have nothing else to do', but at least one person should be working on that area as their main job. This project had 'engine', 'procedural generation' and 'gameplay' and the 'gameplay' was a side-job. It did not work out.Also, the team had an array of different tools which were never or rarely used, like a bug tracker. The team should have used the tools that would have made the project easier to develop. The team's version control, wiki and blog was used and the rest was quietly ignored.

If other development teams would start a similar project, they should remember these few keypoints: ave enough people working on the project, before you begin, have the necessary tools up and running and make sure everyone knows how to use them before you begin, don't be afraid to modify the design, prefer simple solutions to complex ones, because simple solutions most often bring only simple problems with them and prefer modifying to deleting code.

5 REVIEW

This thesis has come a long way. The idea first originated at the beginning of the summer of 2010, when four students at Kajaani's University of Applied Sciences started to think what they should do for their theses and had the idea of making a game. Everyone's thesis would cover a certain area of the game. One would make the engine using SDL, one would create a procedural level generator, one would be the designer and the fourth would create the graphics for the game. Some of those theses would rely heavily on the actual game, while others would rely more on theory and actual creation process. Everyone would still be working on the game. In the beginning, lots of different ideas for the game started flying out to the open. There was only a few requirements/restrictions for the final idea. The game has to use 2D graphics, because it would be a lot simpler to make the engine render only 2D. The game should use content generated procedurally, preferably levels. The game should be small enough to be finished in 3-6 months. Two out of three requirements were met.

The game was not finished in time and reasons vary, but a few main reasons can be pointed out. First, either the deadline was too soon OR the game was too big for the amount of time planned. Second, there should've been one more programmer in the project. Both programmers worked on their areas, but also had to work on gameplay, as there was nobody assigned to that area. Neither of the programmers could completely focus on their area, as they had to jump to coding gameplay every so often. Third, lack of experience. All four of the developers were students, so none of them had professional experience about developing a game. Some of the necessary skills were acquired through some of the classes taught in school, but mostly from teaching themselves at home.

The project began at the hot summer days of 2010 and everyone was excited. First few weeks were used for designing, planning and prototyping. There was schedules and a vague game idea in the beginning, but there was no actual game design. Lot of brainstorming occured and fun and creative ideas were implemented in the design. The prototypes were dumbed-down and had little to do with the game designed, but they were done and the schedule said we had to move on, so the actual development began. Everyone started working and slowly but surely, the basic building blocks got done. The engine started to hum and was running, the level generator did what all level generators do, the world got shape and color as graphics were made and the design started to fill with ideas and content.

SOURCES

Books

Grossman, A. 2003. Postmortems from Game Developers. Berkeley, CA: CMPBooks.

Zerbst, S. Düvel, O. 2004. 3D Game Engine Programming. Boston, MA: Thomson Course Technology.

Internet

Allegro. Introduction. Available: http://alleg.sourceforge.net (Read 14.04.2011)

BitGamer. Carmack: Direct3D is now better than OpenGL. Available: http://www.bit-tech.net/ (Read 14.04.2011)

Box2D. Available: http://www.box2d.org/ (Read: 13.04.2011)

ClanLib. About. Available: http://www.clanlib.org (Read 14.04.2011)

DOSBox. Main Page. Available: http://www.dosbox.com/ (Read: 13.04.2011)

FreeCiv. Main Page. Available: http://freeciv.wikia.com (Read 14.04.2011)

Galaxy Gameworks. Home. Available: http://www.galaxygameworks.com (Read 14.04.2011)

Gamesradar. The sneaky history of stealth games. Available: http://www.gamesradar.com/ (Read: 14.04.2011)

Grasz. Aktualizacje strony domowej RAGE. Available: http://www.grasz.org/ (Read: 14.04.2011)

John Carmack. OpenGl vs Direct-3D. Available: http://www.team5150.com/~andrew/carmack/(Read 14.04.2011)

John Carmack. Welcome, Q3 source, Graphics. Available: http://www.armadilloaerospace.com/ (Read: 14.04.2011)

King Svatopluk's Court. Hexen. Available: http://www.svatopluk.com/hexenscn/ (Read: 14.04.2011)

Lantinga, S. 2011. SDL Library Documentation. Available: http://www.libsdl.org/ (Read: 13.11.2011)

OGRE. Features. Available: http://www.ogre3d.org (Read 14.04.2011)

Ogre. Gallery. Available: http://www.ogre3d.org (Read 14.04.2011)

SFML. Features. Available: http://www.sfml-dev.org (Read 14.04.2011)

Simple and Fast Multimedia Library Main Page.. Available: http://www.sfml-dev.org/ (Read: 13.04.2011)

Simple DirectMedia Layer. Available: http://www.libsdl.org/ (Read: 13.11.2011)

Slashdot. Are Video Game Patents Next? Available: http://slashdot.org/(Read 14.04.2011)

SourceForge. Allegro. Available: http://alleg.sourceforge.net/ (Read: 13.04.2011)

StepMania. Main Page. Available: http://www.stepmania.com/ (Read 13.04.2011)

Unity. UnityScript Introduction. Available: http://www.unityscript.com/lessons1/basics.php (Read 14.04.2011)

Unityscript. Introduction. Available: http://www.unityscript.com/(Read: 14.04.2011)

Wikipedia. DirectX, DirectX pipeline. Available: http://en.wikipedia.org/wiki/ (Read 14.04.2011)

Wikipedia. DirectX. Available: http://en.wikipedia.org/wiki/ (Read 14.04.2011)

Wikipedia. Game engine. Available: http://en.wikipedia.org/wiki/ (Read: 14.04.2011)

Wikipedia. List of games using SDL. Available: http://en.wikipedia.org/wiki/ (Read: 13.11.2011)

Wikipedia. OpenGL. Available: http://en.wikipedia.org/wiki/ (Read 14.04.2011)

Wikipedia. SDL abstractions layers on different platforms. Available: http://en.wikipedia.org/wiki/ (Read 14.04.2011)

Wikipedia. Simple DirectMedia Layer. Available: http://en.wikipedia.org/wiki/ (Read: 13.04.2011)

Wikipedia. Unreal Engine. http://en.wikipedia.org/wiki/ (Read: 14.04.2011)