Politechnika Wrocławska

Faculty of Computer Science and Management
Field of study: **COMPUTER SCIENCE**
Specialty: Information Systems Design

Master Thesis

Multithreaded game engine architecture

Adrian Szczerbiński

keywords:
game engine
multithreading
DirectX 12

short summary:
Project, implementation and research of a multithreaded 3D game engine architecture using DirectX 12. The goal is to create a layered architecture, parallelize it and compare the results in order to state the usefulness of multithreading in game engines.

Supervisor	....	....	…………………….
	Title/ degree/ name and surname	*grade*	*signature*
The final evaluation of the thesis			
Przewodniczący Komisji egzaminu dyplomowego	....	....	…………………….
	Title/ degree/ name and surname	*grade*	*signature*

For the purposes of archival thesis qualified to: *
a) Category A (perpetual files)
b) Category BE 50 (subject to expertise after 50 years)
* Delete as appropriate

stamp of the faculty

Wrocław 2019

Streszczenie

W dzisiejszych czasach, gdy społeczność graczy staje się coraz większa i stawia coraz większe wymagania, jak lepsza grafika, czy ogólnie wydajność gry, pojawia się potrzeba szybszych i lepszych silników gier, ponieważ większość z obecnych jest albo stara, albo korzysta ze starych rozwiązań.

Wielowątkowość jest postrzegana jako trudne zadanie do wdrożenia i nie jest w pełni rozwinięta. Programiści często unikają jej, ponieważ do prawidłowego wdrożenia wymaga wiele pracy. Według mnie wynikający z tego wzrost wydajności jest warty tych kosztów.

Ponieważ nie ma wielu silników gier, które w pełni wykorzystują wielowątkowość, celem tej pracy jest zaprojektowanie i zaproponowanie wielowątkowej architektury silnika gry 3D, a także przedstawienie głównych systemów używanych do stworzenia takiego silnika gry 3D. Praca skupia się na technologii i architekturze silnika gry i jego podsystemach wraz ze strukturami danych i algorytmami wykorzystywanymi do ich stworzenia.

Celem naukowym pracy jest zaprojektowanie architektury silnika gry, który wykorzystuje wiele rdzeni i wątków procesora oraz algorytmu, który wybierze, jak przydzielić zadania do wątków, aby osiągnąć najlepsze wykorzystanie mocy obliczeniowej, którą daje nam procesor. Celem poznawczym pracy jest zbadanie wydajności zaproponowanej architektury i algorytmu. Praktycznym celem pracy jest porównanie różnych wersji silnika gry, aby określić przydatność nowej, zaprojektowanej architektury i algorytmu.

Abstract

Nowadays, as the game community gets bigger and it desires better graphics and higher framerate, there comes a need of faster and better game engines, as most of the current ones are either old, or use old approaches.

A multithreading is perceived as a task difficult to implement and not fully developed yet. It is often avoided by developers, as it requires a lot of work to correctly implement, but the resulting increase of performance is worth the costs.

As there are not many game engines that use multithreading in its full potential, the aim of the thesis is to project and propose a multithreaded 3D game engine architecture as well as to present the main systems that are used to create such a 3D game engine. The thesis will focus on the technology and architecture of the game engine and its subsystems together with data structures and algorithms used to create them.

The scientific goal of the thesis is to design an architecture of a game engine that uses multiple processor cores and threads and an algorithm that will select which component requires the most processing power. The cognitive goal of the work is to examine the performance of the proposed architecture and the algorithm. The practical goal of the work is to compare different versions of the game engine to determine the suitability of the new, designed architecture and the algorithm.

Contents

Streszczenie 2
Abstract 2
1. Introduction 7
 1.1. Purpose and scope of the thesis 7
 1.2. Literature review 8
 1.3. Game 9
 1.4. Game engine 9
 1.5. Problems in game engines 10
 1.6. Similar solutions 11
  1.6.1. Quake engine family (id Tech engines) 11
  1.6.2. Unreal Engine 12
  1.6.3. Unity 13
 1.7. Game engine architecture[9] 14
  1.7.1. Target hardware 14
  1.7.2. Device drivers 14
  1.7.3. Operating system 14
  1.7.4. Third-party SDKs and middleware 15
  1.7.5. Platform independence layer 17
  1.7.6. Core systems 17
  1.7.7. Resource manager 18
  1.7.8. Rendering engine 18
  1.7.9. Profiling and debugging tools 22
  1.7.10. Collision and physics 22
  1.7.11. Animation[10] 23
  1.7.12. Human Interface Devices (HID) 23
  1.7.13. Audio[11] 24
  1.7.14. Online Multiplayer/Networking 24
  1.7.15. Gameplay foundation systems[12] 25
  1.7.16. Game-specific subsystems 27
2. Project 27
 2.1. Engine support systems 27
  2.1.1. Dynamic memory allocation 27
  2.1.2. Memory access patterns 27
  2.1.3. Memory fragmentation 27
  2.1.4. Engine configuration 28
 2.2. Resources and file system 29

2.2.1. File system 29
2.2.2. Resource manager 30
2.2.3. Runtime resource management 31
2.3. Game loop 32
2.3.1. Rendering loop[13] 32
2.3.2. Singlethreaded game loop[14] 32
2.3.3. Multithreaded game loops[15] 33
2.3.4. Time 35
2.4. Human interface devices 35
2.4.1. HID requirements 36
2.5. Debugging 36
2.5.1. Logging and tracing 36
2.5.2. Debug drawing 37
2.5.3. Debug menus 38
2.5.4. Debug console 38
2.5.5. Debug camera 38
2.5.6. Time manipulation 39
2.5.7. Cheats 39
2.5.8. Screenshots 39
2.5.9. Profiling 39
2.5.10. Memory statistics 40
2.6. Rendering engine 40
2.6.1. Scene description 40
2.6.2. Rendering pipeline 42
2.7. Animation 42
2.7.1. Skeleton 43
2.7.2. Pose 43
2.7.3. Skinning 43
2.7.4. Animation blending 44
2.7.5. Animation pipeline 44
2.8. Physics 44
2.8.1. Collision detection[19] 44
2.8.2. Rigid body dynamics 45
3. Implementation 45
3.1. Tools 45
3.1.1. Programming language – C++ 45
3.1.2. Development Environment – Visual Studio 2017[20] 46

3.1.3. Version Control – GIT[21] 47
3.2. Engine support systems 47
3.2.1. Stack allocator[figure 3.1.] 47
3.2.2. Pool allocator 48
3.2.3. Memory defragmentation 48
3.2.4. Engine configuration 48
3.3. Resources and file system 49
3.3.1. File system 49
3.3.2. Resource manager 49
3.3.3. Runtime resource management 50
3.4. Game loop 51
3.4.1. Rendering loop 51
3.4.2. Single threaded game loop[figure 3.2.] 51
3.4.3. Multithreaded game loop 53
3.4.4. Time 54
3.5. Human interface devices 56
3.6. Debugging 57
3.6.1. Logging and tracing 57
3.6.2. Debug drawing 58
3.7. Rendering engine 59
3.7.1. DirectX initialization[24] 59
3.7.2. Scene description 61
3.8. Animation 63
3.8.1. Skeleton 63
3.8.2. Skinning 63
3.8.3. Animation blending 63
3.9. Physics 63
3.9.1. PhysX initialization 63
3.9.2. PhysX Visual Debugger (PVD) 63
3.9.3. Collision detection[25] 63
3.9.4. Rigid body dynamics 64
4. Experiments[26] 64
4.1. Load testing (performance)[27] 64
4.1.1. Singlethreaded architecture 64
4.1.2. Job system 67
4.1.3. Job system with dynamic thread allocation 69
4.1.4. Summary of load testing 72

4.2. Functionality testing 74
4.3. Compatibility testing 74
4.4. Soak testing 74
4.5. Regression testing 75
5. Conclusions and future work 75
5.1. Conclusions 75
5.2. Future work 76
6. Bibliography 77

1. Introduction

1.1. Purpose and scope of the thesis

The aim of the thesis is to project and propose a multithreaded 3D game engine architecture that will achieve a better performance than a singlethreaded counterpart, as well as to present the main systems that are used to create such a 3D game engine. The thesis will focus on the technology and architecture of the game engine and its subsystems together with data structures and algorithms used to create them.

The goal of the thesis is to create a multithreaded 3D game engine using a job system with the use of C++ programming language and DirectX graphics library.

The scope of the thesis contains:

- literature review in the field of C++ programming, multithreading as well as rendering of 3D graphics, physics, audio and other subsystems that make up the engine
- examination of algorithms that can be used to implement multithreading in a game engine
- examination of architecture of 3D graphical engines and consideration of problems that arise when implementing multithreading in such engines
- presentation of modules and classes that make up the multithreaded game engine and the accompanying code
- development of experiments that can be used to test the game engine's performance
- testing of the implemented system
- analysis of system's performance on different, modern hardware

The game engine with multithreaded architecture as well as all needed to compare other versions of the game engine (singlethreaded architecture, different versions of multithreaded architecture) should be implemented. The engine's implementation shouldn't focus only on multithreading, but rather on the game engine as a whole to replicate what can be seen in practice to simulate a properly working, ready to use game engine.

The thesis should include all comparisons, calculations and experiments performed to decide the performance of previously implemented versions of the game engine to choose the best architecture model that can be used as a base of further development of commercial products. It should also confirm the reliability of performed experiments and gathered results using statistics tests.

The scientific goal of the thesis is to design an architecture of a game engine that uses multiple processor cores and threads and an algorithm that will select which component requires the most processing power. The cognitive goal of the work is to examine the performance of the proposed architecture and the algorithm. The practical goal of the work is to compare the game engine with other game engines to determine the suitability of the new, designed architecture and the algorithm.

This work should be treated as an overview of few of many different techniques, algorithms and other solutions that can be used in order to create a game engine as it's impossible to describe all of them accurately. Each of them could be the subject of a separate topic. Furthermore, the purpose of this work is not to introduce any completely new solutions, but rather showing how to combine selected existing solutions in order to achieve the set goal. However, with such an extensive project and implementation, a lot of small things had to be innovative (ex. executor distributing thread workers, most of the custom data structures).

The first chapter describes the field which the work is based on and introduces many terms related to a 3D game engine as well as similar solutions and problems related to many of today's game engines. The second chapter shows the division of the game engine into different layers and multithreaded approaches that can be used to, in theory, increase the performance of the engine. The third chapter presents the tools used and describes the implementation of most of the elements of the engine, focusing mainly on the introduced multithreading. The fourth chapter describes carried out experiments and achieved results of different versions of the game engine. In the fifth chapter, there is a summary and final results, as well as information on the possibilities of expanding the presented engine in the future.

1.2. Literature review

The process of a game engine development requires a lot of programming and mathematical knowledge. In the thesis, the used programming language is C++. "*The C++ Programming Language"*[1] book served as a reference when writing C++ code. It describes the functions of the C++ language at the basic level as well as the advanced level so that every programmer can find something useful for themselves. The book also teaches object-oriented programming techniques and provides examples of proper code writing. Other books related to C++ language worth mentioning are books from the "*Effective C++*" series by Scott Meyers[2-4], as well as John Lakos' "*Large-Scale C++ Software Design"*[5], which provide excellent guidelines for creating a C++ code.

Game engines, and especially the rendering and physics processes, are firmly based on vector mathematics, matrix mathematics, and physics. Danny Kodicek's „*Mathematics and Physics for Game Programmers"*[6] very well teaches novice game programmers the skills needed to create computer games, starting with the basic aspects of mathematics and physics that are relevant to games, to more complex topics. The book combines theory and practice and contains many examples of games that the reader can implement. The author of the book also presents the basics of understanding 2D and 3D geometry. The second book on a similar subject is the book *„Mathematics for 3D Game Programming and Computer Graphics"*[7] by Eric Lengyel. It presents the necessary level of such areas of mathematics as vector geometry and linear algebra and provides a lot of useful information on more advanced 3D programming topics such as lighting and determination of objects' visibility.

Computer hardware, which is used in today's devices like personal computers, game consoles, or mobile devices, uses extensively parallel programming. The processors (CPUs) and graphics processors (GPUs) in these devices operate simultaneously, using the "divide and conquer" approach to achieve high-speed processing. Unfortunately, many of the game engines use single-core solutions, which in the case of large games, limit programmers from using high-resolution graphics, due to the need of intensive calculations needed to achieve the desired result. Using multiple processor cores and parallelizing work can allow the programmer to take full advantage of the available hardware resources, and as a result, increase the game's performance. The book "*Introduction to Parallel Computing"*[8] proved to be an excellent source of information when it comes to the theory of parallel computing. This book contains the basics of designing and programming parallel software, as well as detailed descriptions of design and analysis of many parallel algorithms.

1.3. Game

The general definition of a "game" is a play between players or teams according to rules set defined by regulations of a given discipline includes different types of games, like checkers, various card games such as poker and blackjack, many kinds of children's plays, video games, etc. Sometimes, especially in game theory, people talk about a game as any conflict situation, where the player is any of the game's participants (human, animal, or even enterprise). The game theory examines which strategies players should choose to achieve the best results within a previously defined set of game rules.

The simplest definition when it comes to the context of a computer, a game (in this case usually known as a *video game*) is a type of computer software that is intended for entertainment or educational purposes (called interactive entertainment) and requires the user (the player) to solve different kinds of tasks to achieve previously defined success. Many people associate video games with a three-dimensional, virtual world that contains a human, animal, or a vehicle as the main character that is under the player's control.

1.4. Game engine

The definition of a game engine first appeared in the mid-90s together with first first-person shooters, like for example *Doom* by id Software. The first engines were designed with a division of its components (rendering system, physics system, collision system, or audio system), resources, game world, and game rules, which created the ultimate experience for the player. The value of this division was seen when the developers first started to license the games and change them into new products through the creation of new graphics, word layouts, weapons, characters, vehicles and game rules with a minimal amount of changes in the software of the game engine. This division initiated the creation of a new community – a community of "mods" – group of gamers and small game studios that built new games through modification of already existing games by using tools provided by the original developers of the game.

At the end of the 90s, game engines were created with mutability in mind, so the game could be easily customized by the use of different scripting languages in order to minimize future costs of modifications of the game, or of the creation of other games based on the same components used to create the game engine. This practice is still used nowadays, as most current companies create their games on the already existing public or private game engines to fasten the production of new games by a lot and highly reduce the cost of development.

The difference between a game and its engine is not always visible. There are two main approaches when it comes to creating a game and a game engine. Some developers create the engine for the game, and some create the game based on the already existing engine. Following the first approach, we can obtain a highly optimized game, but on the other hand, the cost and the time needed to create such a game is much bigger than when using the second approach. In this case, the game is also usually „hardcoded" into the engine itself, which prevents the developers from further modifications of the game and reuse of the game engine without spending resources on modifying the engine. By using the second approach (*data-driven architecture*), it's possible to create a game, where the line dividing the game from the engine is easily visible. This approach allows the developers to quickly develop the game and modify it without even knowing what's happening in the inside of the engine's code.

However, it doesn't mean that the mentioned approaches are the only methods to create a game engine. Some engines cannot be used to build more than one game, as it is the case with many old games (like for example Pacman), where the concept of the game engine practically didn't exist. Other engines can be modified to create very similar games, while others can be "modded" to create any game.

Some may think that the game engine is like a multipurpose software that can play any game possible. Unfortunately, such a solution probably cannot be achieved. Most of the game engines are created to run a specific game on a particular hardware platform. Even the most universal cross-platform engines are designed only to build a game of a specific genre such as first-person shooters, or platform games. That's because the more general the game engine is, the less optimal it is to run a specific game on a particular platform.

Designing of any efficient software requires solving of many different problems, which usually makes the developers to make many compromises. For the game engine to be optimal and efficient, the programmer needs to take into account all possible algorithms for a specific task to be solved. One optimal algorithm used for rendering small, indoors scene can be much better than an algorithm used to create the best algorithm used for rendering big, outdoor worlds. For example, when rendering an indoor scene, the programmer may consider using a BSP tree in order to make sure that no objects that are covered by other objects are drawn, but in case of outdoor world it may not even be needed to use occlusion, but it may be required to use an algorithm that will reduce the number of polygons of objects that are located much further from the camera (level-of-detail techniques) in order to minimize the needed processing power. On the other hand, sometimes, the developer needs to consider the requirements of a game, where for example if it needs to have high resolution, good looking graphics, the developer cannot use algorithms that reduce the number of polygons for the objects.

With the increase of processing power and faster graphical cards together with more optimal algorithms of rendering, the differences of game engines for different game genres are less visible. For example, today, we could use an engine created for a first-person shooter to create a strategy game.

1.5. Problems in game engines

One of the biggest problems of game engines is hardware limitations. The bigger the game, whether it is a larger number of objects, or more extensive game logic, the more calculations the engine must perform, which translates into very high CPU usage. Virtually all modern game engines work only on one processor core, so they are not able to use the full potential of the processor. The most time-consuming components of the game engine are the rendering engine and the physics engine, although the latter is very dependent on the physics that are used in the game. For example, a game where we move a car will require much less computational power of the processor than a game simulating the start and flight of a rocket. Both components are also highly dependent on a number of objects that are needed to both render and calculate real-time physical formulas for.

Game performance may also be affected by the amount and speed of memory on a hard disk. Many objects needed for the game are stored together with the game files in an encrypted form. Files such as object models, textures, or materials are often loaded only when they are needed, for example when the object is to appear on the player's screen. In the case of slower hard drives, depending on the rendering method used, this may lead to longer loading time of objects, loading of the game, or creation of objects without a model or texture.

Another hardware limitation can also be the amount and speed of RAM. Some development teams creating a game, often for smaller games or games that often use the same assets to speed up the reading of, decide to load most of the assets into RAM. This is often done during the game's loading screen. Unfortunately, such a solution for larger games or games using graphics with higher resolution is often impossible, because it requires a huge amount of RAM. For example, some assets of the game may require up to several hundreds of megabytes of memory when loaded.

As mentioned earlier, the definition of game engine states that the engine should allow for the creation of multiple games. However, this is not entirely possible. If the studio wants to develop an efficient computer game, the programming teams will often focus on writing the engine so that it doesn't contain additional functionalities, which will reduce the minimal required hardware resources. This is so-called "writing a game engine for the game". In this case, each subsequent game that is created on such an engine is only slightly different from the previous one, and if the studio decides to develop a different type of game, the engine will require customizations or adding of new functionalities. An example of such an engine is the engine of PacMan game, where the boundary between the engine and the game is practically non-existent. On the other hand, by implementing the engine so that it can be used for many different types of games, the engine is exposed to performance problems, and developers using the engine will often have to adjust their game and style of writing to it. Unreal Engine 4 and Unity are examples of such engines.

To summarize, we often distinguish three types of engine in terms of reusability:

- An engine that cannot be used to create more than one game, eg. PacMan
- An engine that can be adapted to create games that are very similar to each other, eg. Hydro Thunder Engine
- An engine that can be modded to create any game in a given genre, eg. Quake III Engine, Unreal Engine 4, Unity, Source Engine

Many game developers also distinguish a fourth type of game engine, which is an engine that can be used to create any type of game, but such an engine is probably impossible to achieve with today's technology.

The methods of creating game engines as well as games often vary depending on the studio. Publishers, having a massive competition on the game market, are not always willing to share their discoveries with other companies. Many game studios create their own internal tools to help in the creation of game engines, or games, which makes it hard for someone from the outside to learn how the company developed their game. Thus, there arises the problem of the lack of standardization of such methods and practices both because of the invention of internal solutions that are hidden from others, but also because of the lack of the possibility to create a standard that would cover every game genre. As a result, new developers often do not even know how to create game engines or computer games until they try and learn it themselves.

1.6. Similar solutions

1.6.1. Quake engine family (id Tech engines)

The family of Quake game engines (also called id Tech engines) was first created after developers of id Software realized the potential of first-person shooters after their release of very successful game *Castle Wolfenstein 3D*. Id Software quickly created multiple engines together with their next games like *Doom, Quake, Quake II, Quake III,* etc.

Most of the engines from the Quake family had very similar architecture and were released under GPL license, which allowed other studios to use the engine for their games. Nowadays, there are dozens, if not hundreds, of engines that were created based on id Tech engines. It allowed id Software company to gather many fans of both, the engines as well as the games.

Engine *id Tech 1* (*Doom Engine*) is the first game engine created by id Software for a very popular game *Doom* designed for Steve Jobs' NeXT platform and layer ported to DOS environment. In 1997 the engine was published on a non-commercial license as the developers wanted to use it on Linux OS. Two years later, the code was released under GPL license allowing the commercial use of the game engine.

Engine *id Tech 2* (*Quake* and *Quake II Engine*), created for games *Quake,* and later *Quake II* was a first game engine that allowed for displaying of full 3D graphics in the games. The engine was released under GPL license, which allowed for the creation of future versions of the engine.

Engine *id Tech 3* (*Quake III Engine*), created mainly for game *Quake III: Arena,* was developed as a response to the concurrent engine – Unreal Engine. As previous two engines, the code of the engine was released under GPL license.

Engine *id Tech 4* (*Doom 3 Engine*), created for game *Doom 3* in 2004, introduced many new features in comparison to the other engines. The engine used bump mapping technique, for the first time in computer history, whole lighting and shades were generated in real time thanks to the use of stencil buffer, the first version of the engine supported graphics cards using DirectX standards. On the other hand, the engine required high computation power, which at first prevented the creation of great locations and open terrains.

Engine *id Tech 5* (*Rage Engine*), created for game *Rage* in 2011, introduced a mega texture function allowing usage of high-resolution textures.

Engine *id Tech 6,* created for a remaster of original *Doom* game with the same name in 2016, designed to focus on ray tracing to improve graphics of games.

1.6.2. Unreal Engine

Unreal Engine 4 is a video game engine and editor, which won many awards, including the Guinness World Records award for "most successful video game engine". It is created in C++ language by Epic Games company and provides the developers with many different features. The engine was and still is being used to develop many games of various types.

Use of the engine is completely free and provides everything that developer needs to create a game, although when shipping the product, a royalty of 5% of earnings must be paid to the Epic company if the game earns more than $3000 per calendar quarter. Epic Company provides full C++ source code of the game engine. Having access to Unreal Engine's source code gives an opportunity of modifying the game on the engine's level, which is very useful when there is a problem that can't be fixed and need to be changed in the engine's code.

The games made in Unreal Engine 4 can be developed for almost every nowadays used platform including Windows, Linux, macOS, Android, iOS, Nintendo Switch, PlayStation 4, Xbox One, SteamOS, HTML5 and all devices using virtual reality.

The engine provides a multiplayer framework that provides a client/server architecture, which makes it easy to make multiplayer games, where the engine makes all the work to replicate actions made in-game to another player. It also implements an advanced AI library with support of navigation meshes that help in making smart AI characters.

Unreal Engine is known for its possibility of achieving best-looking and realistic graphics thanks to its physically-based rendering, dynamic shadows, realistic reflections, and lighting. There are many videos that were made in Unreal Engine of rooms in houses, apartments, etc. that show the beauty of the engine. Unreal Engine is not only used for developing games, but also cinematics, planning, and in some companies to simulate different life cases.

When it comes to coding, Unreal Engine provides many libraries and functions that can be used while programming the game. It also implements macros that can be used to declare variable specifiers and metadata that transform the classes and variables to be part of Unreal Engine, which otherwise would be very difficult for developers. Declaring the variable specifiers and metadata give us access to the variables inside of Game Editor.

Unreal Engine's game editor is a vast development tool that provides many different functionalities like a level editor, material editor, blueprint editor, behavior tree editor, user interface editor, etc.

The main advantage of the tool is its level editor that is used to construct gameplay levels (adding instances of Actors, blueprints, and meshes to the game). It is the level editor that shows how the game will look. Together with other tools from the editor, it is possible to debug the game from within the window. Thanks to created metadata it's possible to see all the values of variables in the editor and see how and when they change.

Unreal Engine 4 Editor also implements visual scripting called Blueprints, that provide a gameplay scripting system on the concept of using node-based interface to create gameplay elements from within Unreal Editor without the need of coding. Blueprints, while easy to use and prevalent amongst new developers, are a worse choice when it comes to gameplay programming. They're mainly used for parts of the game that need visual design, like meshes, user interface or input controls. It's also possible to communicate between C++ code and Blueprints due to metadata generated during compilation.

1.6.3. Unity

Unity is one of the most popular game engines used today. Its integrated development environment allows the developers to create 2D and 3D games, or other interactive materials such as visualizations, or animations. The engine can be used on Microsoft Windows, macO, and Linux, but thanks to its cross-platform orientation, it allows to develop applications for web browsers, personal computers (Microsoft Windows, Linux, Apple Macintosh), game consoles (Microsoft Xbox 360 and Xbox One, Sony PlayStation 3 and PlayStation 4, Nintendo Wii and Wii U), mobile devices (Google Android, Apple iOS) and even TV platforms (Android TV, tvOS) or virtual reality systems (Oculus Rift, SteamVR, Gear VR).

Unity engine is focused on allowing programmers easy development and cross-platform game distribution. Similarly, to Unreal Engine, it provides the developers with easy to use, built-in environment editor that allows for creation and edition of resources that create the game's world. Unity features a huge analysis toolkit that allows for analysis and optimization of the game on each desired platform.

Initially, Unity allowed to create games in 3 different programming languages: UnityScript (similar to JavaScript), C# and Boo, but in Unity 5 support of Boo language was removed. Thanks to the use of scripting languages there's no need to recompile the code to preview how the game will look and play.

Until version 4.6, the engine was available on a paid license, or a free one that restricted many of the functionalities. With the release of Unity 5, almost all of the engine's functions are free to use for all developers who don't exceed $100.000 revenue annually.

Unity was used in many popular games, like Blizzard's *Hearthstone* (2014), Team Cherry's *Hollow Knight* (2017), or Unknown Worlds Entertainment's *Subnautica* (2018).

When it comes to multithreading, Unity provides a wrapper on C++'s multithreading function allowing developers to use multithreading in games, but doesn't say anything about using multithreading in the game engine itself.

1.7. Game engine architecture[9]

The game engine consists of runtime component and development toolkits. It is built with layers. To avoid coupling between the system, the higher layers are dependent on lower layers, but not the other way around.

1.7.1. Target hardware

Target hardware layer describes the physical parts of a computer or console on which the game will run such as a processor, RAM, hard drive, etc. This layer may also represent whole platforms that will run the game such as personal computers (Microsoft Windows, Linux, and macOS), consoles (Xbox, PlayStation, Nintendo Wii), handheld platforms (Nintendo Switch), mobile platforms (Android, iOS), etc.

1.7.2. Device drivers

Device drivers are low-level programs or program fragments that are responsible for the device and intermediating between it and the rest of the computer system. The device drivers reduce the need for communication between the operating system, or game engine and different types of devices.

1.7.3. Operating system

An operating system is a software responsible for managing a computer system, creating a runtime environment and controlling user's tasks. On a personal computer, the operating system is continuously running. It supervises running of the programs by managing:

- resources of the computer system – resource allocation, synchronization of access to resources, protection and authorization of access to resources, resource recovery, a collection of data on the use of resources
- a process of an application (one of which can be the video game) – creation and deletion of a process, stopping and restoring the process, providing mechanisms that allow for the synchronization of processes and communication between processes
- operating memory – keeping information on which part of the memory is currently being used and by what, deciding which process should be loaded into memory if the memory is free, allocation and deallocation of memory
- files – creation and deletion of files, creation, and deletion of folders, support for end users in file operations, mapping files on a data carrier, creation of file copies
- input/output – buffering and processing of input/output
- data carriers – management of free memory, write allocation, disk planning

The operating system is also responsible for time management to divide the available device resources between many different processes. This process is called *preemptive multitasking* and means that the game engine or any other software can never achieve full control of the hardware and must work well with other applications in the system.

On early platforms, the operating system mostly didn't exist, but if it did, it was just a small library compiled into the game's executable file. On these systems, the game had access to all of the device's resources as it was the only running software on the device. Nowadays, game consoles have operating systems similarly developed to these on personal computers. The operating systems allow users to do multiple operations and run various software at the same time. It's now possible to play one game, exit it and come back without losing any progress thanks to the possibility of running the game in the background. Most of the newest consoles provide a feature that allows to stop the game in any time and open an overlay (eg. XMB interface in PlayStation 4, dashboard in Xbox One) that provides the user with many different features like sending messages to other users, taking a screenshot of the game or recording the gameplay while the player plays the game to share it with friends later. It is also possible to download game patches and DLCs while playing other games, so there's no need to wait for a game to download before starting to have fun. This allows developers to close the gap between personal computers and consoles.

1.7.4. Third-party SDKs and middleware

Interfaces provided by an SDK are called an application programming interface (API).

Most of the game engines use third-party SDKs and middleware written by other developers or studios. There are two main reasons why the usage of SDKs is so popular. First, SDKs are mostly free and already finished, which helps to reduce the need for money spent as well as the time needed to implement such libraries. The second reason is that SDKs are usually developed by teams that specialize in the domain covered by the API, which means that the algorithms provided by the SDK will generally be very well optimized.

1.7.4.1. Data structures and algorithms

Games heavily use different types of containers and algorithms to store various kinds of data in them. A perfect example is the player's inventory which usually is stored as an array of pointers to items that player's character poses.

Examples of libraries providing data structures and algorithms:

- *Boost* – a massive collection of programming libraries that extend the functionality of C++ language. It's very similar to its predecessor, Standard Template Library (STL). Boost provides the developer mainly with general purpose libraries (smart pointers, regular expressions), containers, file system handling, multithreading and tools for the creation of other libraries.
- *Folly* – a library created and used by Facebook. It's firmly focused on efficiency and practicality of its algorithms. The library mostly implements only the data structures and algorithms that C++ Standard Library or Boost didn't and removed them in case of Boost implemented such features.
- *Loki* – C++ software library that makes extensive use of C++ template metaprogramming and implements several commonly used tools like typelist, functor, singleton, smart pointer, object factory, visitor and multimethods.

C++ Standard Library and Standard Template Library (STL)

C++ Standard Library is a standardized in ISO Standard library written in the C++ language and provides the users with collection of classes and functions such as generic containers (vectors, lists, maps, queues, stacks, arrays, tuples), algorithms to manipulate these containers (find, for_each, binary_search, random_shuffle, etc.), function objects, generic strings and streams (iostream), filesystem library, smart pointers, multithreading library, etc.

Standard Template Library is a C++ library containing algorithms, containers, iterators and other constructions in the form of templates. It is often called a generic library, because nearly all of its components are parametrized as templates, which allows the algorithms to work with almost all types, including custom types defined by the user.

1.7.4.2. Graphics

Most of the game engines are using already existing libraries for the rendering of the game, such as:

- *DirectX* – a set of API functions supporting the graphics rendering (two-dimensional and three-dimensional) and sound created by Microsoft. DirectX works only on Microsoft Windows and Xbox consoles. The newest version, DirectX 12, introduces low-level programming allowing to increase the efficiency and performance of graphics in games.
- *OpenGL (Open Graphics Library)* – open and universal API for three-dimensional graphics rendering created by Khronos Group. OpenGL can be used on more platforms than DirectX, like for example on Linux.
- *Vulkan* – low-level, multiplatform three-dimensional graphics API. Vulkan is a competition of DirectX 12. It offers a lower overhead, higher control over the graphics card and lower processor load then in its predecessor, OpenGL.
- *Glide* – three-dimensional graphics API created by 3dfx for their graphics accelerators Voodoo. Its main goal is to increase the performance of games. Due to the popularity and improvements in DirectX and OpenGL, Glide has become redundant.
- *libgcm* – low-level three-dimensional graphics API created by Sony Computer Entertainment used in PlayStation 3 and PlayStation 4. However, there are different libraries for PlayStation consoles, libgcm is preferred due to its efficiency.
- *Edge* – a rendering and animation engine created by Naughty Dog and Sony Entertainment for PlayStation3

1.7.4.3. Collision and physics

Collision detection and physics of different rigid bodies can be implemented in the game using the following libraries:

- *PhysX* – a free set of development tools that allows obtaining special effects in games that closely simulate the behavior of real physical objects. It is developed by Nvidia, which provides it for free for developers.
- *Havok* – a software package that provides physical engine components and other tools useful when creating a computer game or simulations, produced by Havoc.
- *Open Dynamics Engine* – open-source physics engine created by Russell Smith. It's divided into two components: rigid body dynamics and a collision detection engine.

1.7.4.4. Character Animation

SDKs for character animation include:

- *Granny* – a tool suite created by RAD Game Tools. It provides a runtime library for reading and modification of exported model and animation data as well as a large animation system.
- *Havok Animation* – a module of Havok engine, whose purpose is to control the character's animation. It provides effective animation playback and functions such as reverse kinematics. It is also often called Havok Behavior.

1.7.4.5. Biomechanical Character Models

- *Endorphin* and *Euphoria* – plugins for Autodesk Maya that use advanced biomechanical character models to simulate the realistic movement of a character.

Endorphin is used to simulate biomechanical characters by connecting animation with physics. The animator sets physics parameters for the model of a character, like the position of the center of gravity, weight distribution, etc. and defines the movement the character should perform. The rest is handled by the plugin. For example, if two characters were to collide with each other, Endorphin would define how the character will fall by itself without the need to create a fall animation for the characters.

Euphoria is a runtime version of Endorphin to generate the movement of a character in real time. This plugin is handy in games as it allows to create an animation for every, even unpredicted instance of animation.

1.7.5. Platform independence layer

The bigger the audience, the more successful the game can be. That's why most of the more prominent game companies focus their engine on platform independence. The platform independence layer is responsible for wrapping the functions from different operating systems or platforms to create custom functions that will be used by the developer of the game. This reduces the need for using separate functions for each different platform, which makes it possible to develop the game much faster without the need of searching the documentation for platform-specific functions. By creating a custom wrapper, the developer can protect the game engine from the need for adaptation to changes in the used SDKs in the future. The layer also allows the game to be compiled for a specific platform, which reduces the amount of code which would not be used when running the game on a different platform. For example, some functions from older libraries, like C's standard library, can hugely differ from the same functions on another platform.

1.7.6. Core systems

Most of the more significant, complex C++ software programs to create a more reliable software, or video game require the use of additional software utilities:

- *Assertion* – predicate placed at a certain point in the code. Assertion indicates that the programmer assumes that the predicate is real in a given place. If the predicate is false, the assertion stops the execution of the program. It is mostly used when testing the developed software to see if the code is always working as it was designed to work. The advantage of using an assertion is the ability to check in which part of the source code of the program an error occurred. Assertions are usually removed from a final product of the software.

- *Memory management* – most of the game engine developers implement their system of memory allocation and deallocation to achieve better performance of the engine.
- *Math library* – games need to perform a considerable amount of calculations. Most popular in games is linear algebra. Vectors and matrices are used for geometry in the game's rendering component. Many of the mathematical equations are also solved in the physics engine. That's why most of the engines use an efficient, highly optimized math library.

- *Custom data structures and algorithms* – usually there's a need to create custom data structures and algorithms that operate on them. Structures are often used to help the developer gather information about an object in one place instead of saving the data in custom variables of some class. This is highly useful in rendering layer, where to render an object, the developer needs to provide many pieces of information about the object, like what materials is it built from, what's its shape, where in the world it should be placed, what's its scale, information how to render the object, etc.

1.7.7. Resource manager

The resource manager is responsible for allowing the developer access to different assets of the game that are saved on the hard drive, like sounds, models, textures, materials, fonts, collisions, physics parameters, a map of the game, animations. The resource manager also provides functions to allocate the memory and load the resource into it so that it can be used for example in rendering but also handles the deallocation of resources. Some engines' resource managers make use of raw file types, while others use specially archived packages that store the assets of the game.

1.7.8. Rendering engine

The rendering engine is responsible for communication with the graphics card to draw objects on the screen. It's one of the biggest subsystems of a game engine due to the use of many different algorithms that are needed to optimize what is rendered. There is no standardized way to implement a rendering engine. It can hugely differ from engine to engine based on graphics SDK used as well as the number of objects used in the game, or the game world's size in general. However, one of the best methods to implement a rendering engine is to use a layered architecture.

1.7.8.1. Low-level renderer

The low-level renderer is the core of each rendering engine. It is responsible for providing functions for communication with graphics cards as quickly as possible. This component contains the definitions of needed structures to draw a rendering target – an object that is to be rendered. The structures include data about: materials and shaders, static and dynamic lighting, cameras, texts and fonts, primitive geometry, viewports and virtual screens, texture and surface management, debug drawing (lines, raytracing, etc.) and Graphics Device Interface.

1.7.8.2. Graphics Device Interface

Graphics Device Interface is designed to detect any graphics devices, initialize and configure (setting up back-buffer, stencil buffer, etc.) them by using a graphics SDK (DirectX, Vulkan) so that they are ready for the rendering of the rendering targets.

On a personal computer, the rendering component needs to be integrated with a message loop of a Windows operating system. In each iteration, the game loop checks the Windows message queue to find any exciting messages to use, for example, keyboard messages for *human interface devices* layer. If there are no messages, if there's no frame per second limit, the renderer is free to render as many times as possible until a message appears. This can lead to coupling because if there were for some reason too many messages, the renderer wouldn't have any time to render and should be avoided.

1.7.8.3. Other renderer components

Other renderer components are handling previously mentioned structures to convert them into rendering data that can be used by a graphics card. To achieve such data, the components connect the structures into a primitive rendering target, which is connected with a specific primitive geometry, or a mesh that describes its shape, material that it is built from, texture that is applied on the material, information on how the light affects it, etc.

The components are also responsible for the handling of the state of the graphics card. Before rendering a frame, the renderer needs to make sure that the graphics card is not processing something, or already finished rendering the previous frame before being able to handle the rendering of another frame.

1.7.8.4. Culling optimizations

The lower-level renderer is responsible for rendering of rendering targets as fast as possible, regardless if the object is visible or not. That's why a higher-level component is needed to reduce the amount of rendering targets that are not visible for the player.

Culling optimization is one of the first stages preceding the rendering of a scene, in which one determines which of the objects placed on the scene are visible in a given view.

Games can have thousands or even millions of objects, but usually, only a small part of a scene is observed, and therefore a small percentage of all objects are visible. For this reason, there is no need to devote time and memory for processing of invisible objects at all.

There are three main culling optimizations:

- *Viewing frustum culling* – the virtual camera model defines the view block (most often it is a truncated pyramid with a rectangular base or a cuboid), in the volume of which there are objects visible in a given projection. To speed up the process of qualifying objects, the scene is organized hierarchically, "closing" the whole in the octal tree, BSP, etc. This method is usually used to reject objects that are certainly not visible – it is the first stage for other methods.
- *Occlusion culling* – determining which objects (or parts of objects) are obscured by other objects closer to the observer. This method is quite complicated because with the movement of the observer, the dependence of objects obscuring changes all the time. Occlusion culling is ideal for situations when large parts of the scene

are observed (e.g. visualization of a terrain) or when the scene contains a lot of details.

- *Backface culling*[figure 1.1.] – rejection of the rear walls. When the objects on the stage are made of polygons, and it is determined that the "front" of the polygon is outside the solid, while the "back" from the inside, then there is no need to display the "back" surfaces to the observer. The method is straightforward and fast, usually rejects large parts of polygons, but does not give a general guarantee that virtually all invisible polygons will not be displayed. The method is fully effective only for convex solids. Rejecting the back walls is responsible for the common effect in computer games, when by chance the camera will be inside an object, instead of the interior view, the object disappears completely (all visible polygons are at the back). This method is not suitable for displaying translucent objects.

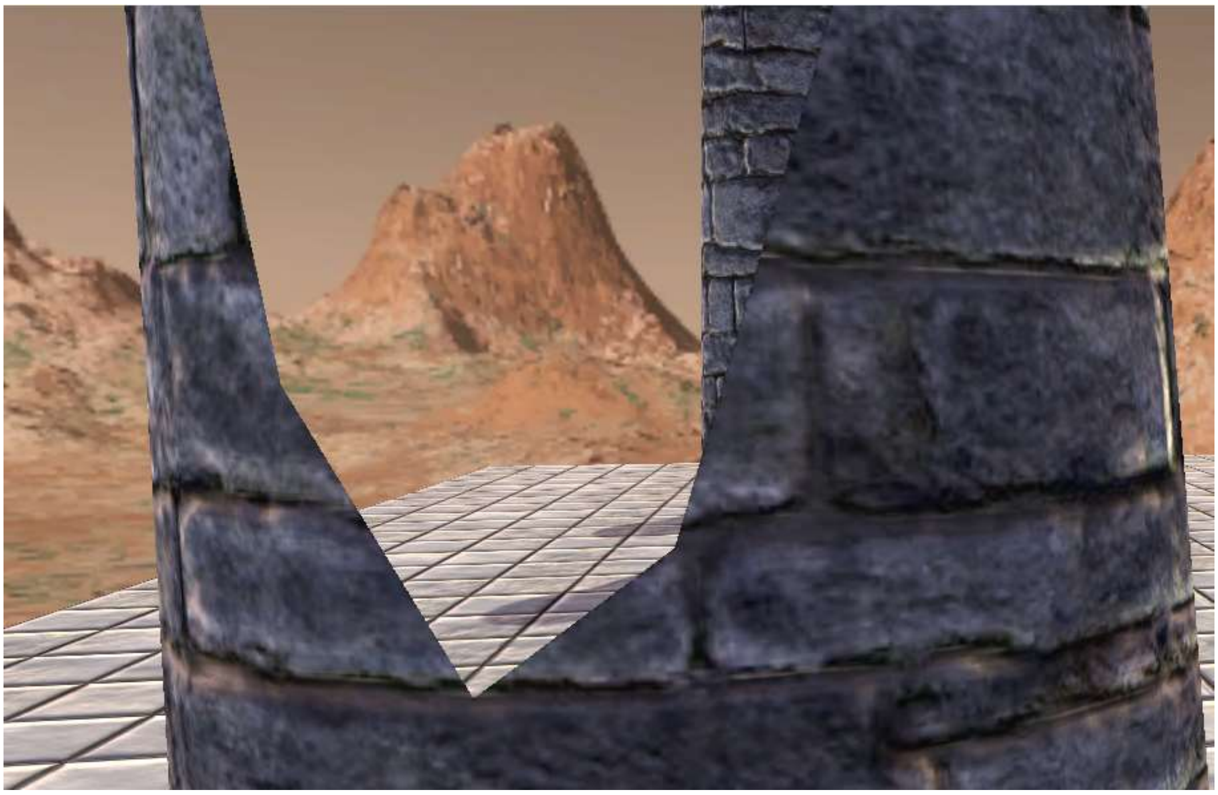


Figure 1.1. Backface culling

1.7.8.5. Visual effects

Nowadays it's almost required for a game engine to have a visual effects system to create astounding graphics. The system is rendered by a rendering component; therefore it is often located inside of the renderer, but in some cases, the visual effects are implemented in a separate component.

Many different visual effects that are implemented in game engines:

- Particle systems – a collection of individual elements called particles are representing a specific effect. It can be, for example, rain, snow, smoke, fire. Particle systems in fantasy games also serve to model special effects such as magic spells

- Decal systems – a decal is imposed on a material to simulate, for example, bullet holes, footprints, etc.
- Light mapping and environment mapping – caching of pre-calculated brightness of surfaces stored in texture maps. They are commonly used in video games to provide global illumination to reduce the computational cost of lighting
- Dynamic shadows – the process of calculation and representation of shadows in the game by testing whether a pixel is visible from the light source
- Full-screen post effects – different types of effects applied after the 3D scene has been rendered

Examples of full-screen post effects:

- High dynamic range (HDR) tone mapping and bloom
- Anti-aliasing (FSAA)
- Color correction

1.7.8.6. Front end

The front end is responsible for displaying different types of information in a graphical form. Nowadays, front end layer is needed in almost all games to implement graphical interfaces that user can use to communicate with the game as well as use them as a part of the game. It is as important in 2D games as in 3D games, as the layer is an integral part of the user experience.

There are multiple types of front-end overlays used in game engines:

- Heads-up display (HUD) – an integral part of most computer games containing additional information about the object, or objects controlled by the player and thus enabling gameplay.
- Graphical user interface (GUI) – different types of forms containing graphical components like buttons, text boxes, sliders, etc. GUIs are mainly used for presenting information about the player (equipment window, inventory window), or allowing a player to send commands related to gameplay.
- Menus – windows or screens providing multiple options that are not always connected with the gameplay. In the menu, a player can change settings of the game, start a new game, save current game, load a previously saved game, view the credits of the game or exit the game.
- Development tools – during the development of a game engine, many game developers include tools that help with the development of the game. The tools may include graphs showing information about current processor load/memory usage, labels displaying information about FPS count, consoles that allow executing previously defined commands. The development tools are usually removed with a final product or kept disabled in the game while giving a player an ability to turn them on.

The overlays are usually implemented in two ways. The first method is to simply draw pairs of triangles visible from an orthographic point of view. The second method is to render a 3D object containing the quads so as they always face the camera.

This layer also includes full-motion video (FMV) system that allows for playing movies that were previously filmed as well as in-game cinematics (IGC) system that is responsible for displaying cinematics in the game. The cinematics may include NPCs speaking when the player comes near them or cutscenes that include a character's model.

1.7.9. Profiling and debugging tools

Game engines are massive software applications, which leads to a significant number of bugs in the code or execution of the code. Profiling and debugging tools help to reduce errors in a game engine by analyzing what happens during the runtime of the game.

Profiling is a form of runtime program analysis (as opposed to static code analysis). It involves examining the behavior of the game engine, or a game using the information acquired during its performance. For example, the developer can profile memory usage of the game, or frequency and the execution time of individual functions. Profiling is usually used to find out which parts of the game engine are optimized to increase its overall speed or reduce memory requirements.

Debugging is a process performed to reduce the number of errors in the software and consists of controlled execution of the program under the supervision of a debugger. The main steps of debugging are error reproduction, isolation of the source of error, identification of the cause of the failure, defect removal and verification of the success of the repair.

Popular tools for profiling:

- *Valgrind* by Julian Seward and Valgrind development team
- Intel's *VTune*
- IBM's *Quantify* and *Purify*
- Parasoft's *Insure++*

Many development teams create their own, custom tools for profiling and debugging, like:

- Displaying profiling statistics (memory usage, processor load, etc.) for each subsystem, or even function of the game engine
- Dumping current state of the game engine (backtrace call stack, memory usage, processor load, etc.)
- Executing the code by calling console commands
- Printing debug messages
- Recording gameplay events to precisely reproduce them

1.7.10. Collision and physics

The collision is an integral element of every game engine. Collision detection allows detecting when objects touch each other and perform actions when they do, like bouncing balls off each other or simply not letting objects fall through each other.

The physics engine is a part of the game engine that deals with the simulation of physical systems, such as systems of rigid bodies (with collision detection), fluid dynamics, deformations of bodies (elastic bodies). The engine performs real-time calculations to simulate physics for each frame of the game.

Collision detection and physics engine are connected, because when the collision is detected, the physics engine usually wants to act like one, or both objects taking part in the collision, like stopping a car that crashed into a wall.

There are many different SDKs providing collision and physics functionalities:

- *PhysX* – a free to use a set of development tools that allow the developer to obtain special effects in computer games that intimately imitate the behavior of real physical objects. PhysX allows objects to be assigned basic physical properties (e.g. mass, position, velocity, acceleration) and to define their interactions between each other (e.g. through collisions, friction, joints) and the environment (e.g. gravity, antigravity, gusts of wind). The library contains advances functions for the detection of real-time collisions of objects, simulation of figures and vehicles in motion, flows of liquids and gases (including turbulent flows), explosions, movement of fabrics (eg. in the wind), as well as deformations of various objects (e.g. compression of balloon, crushing a tin can, tearing a fabric). This engine is an example of middleware, and its primary role is to make it easier for computer game developers to consider the complex physical interactions found in modern computer games without the need to write their code from scratch.
- *Havok* – a paid, closed tool suite providing physics solutions for real-time applications. It is created by Havok development team and currently owned by Microsoft. Havok, unlike PhysX, can be used on almost every platform.

1.7.11. Animation[10]

There are five basic types of animations:

- Sprite/texture animation – animation of a sprite or a texture created by drawing a new sprite/texture for each frame of the animation and displaying the frames in sequence over a specified period.
- Skeletal animation – division of a character's mesh into hierarchical parts of bones.used to animate the character. This animation is done through moving and rotating the bones in different positions to achieve desired poses for the character. It's the most popular animation type used in game engines.
- Morph target animation – a modified version of a mesh is stored as a series of vertex positions. In each keyframe of animation, the vertices are interpolated between these stored positions.
- Vertex animation – animation of individual vertices of a mesh. It is usually used for cloth, hair or water animation.

Animations are sometimes connected with a physics engine to provide more realistic character movement. An example of animation using a physics engine can be a ragdoll animation (ragdoll physics). It's a computer technique based on a realistic simulation of falling bodies. It replaces less realistic specially created animations.

1.7.12. Human Interface Devices (HID)

Human Interface Devices are simply devices for human input. HID include:

- Mouse and keyboard
- Touchscreen
- Console controllers (gamepads, paddles, joysticks, trackballs, steering wheels, motion sensors, etc.)

In some cases, especially on console platforms, the component also handles an output for the devices used by the player. This may include vibrations on gamepads, sounds, or lights generated by controllers.

Human Interface Devices layer is usually developed inside of a game loop, where, for each frame, messages sent to the operating system by different devices are checked and handled by the component. The component operates on a raw message data that provides information about what key was pressed, or released, how much a controller stick was moved from the inside, etc. HID usually also implements functions that allow the developer to check if the keys were pressed in a specific order in a specific timeframe (key sequence), or if multiple keys were pressed in the same time (chords).

1.7.13. Audio[11]

Nowadays audio is one of the most essential features of most large games. However, developers are often forgetting that and don't focus enough on the development of this component.

There are many different development kits that allow for playing audio, as well as many game engines that developed their audio engines, as each game is unique when it comes to sounds and usually requires different modifications of audio based on the environment the player is located in.

DirectX from Microsoft instead of only providing graphics solutions implements also an XAudio2 runtime audio engine that allows for mixing and signal processing. It separates sound data from the voice to provide a developer possibility of filtering the voice using programmable digital signal processing.

When it comes to engines, the Quake engine has a very basic audio engine that most of the developers had to modify to achieve sounds specific to their games. Unreal Engine 4 has an excellent 3D audio rendering engine but is very poor when it comes to functionalities it offers, as the only function it provides is to play a sound in a specific place on the scene, or globally for the player.

1.7.14. Online Multiplayer/Networking

Many games are designed to allow many players to play together. There are at least four multiplayer game modes:

- *Single-screen multiplayer* – two, or more players play using the same screen and more than one set of human interface devices. The game usually provides both players with a controllable character. The camera in this mode is either locked to one player or points to a center between the players and zooms out when one of the players is about to leave the screen. An example of games using this mode is *Smash Brothers.*
- *Hot seat* – used mainly in strategy games, where one set of human interface devices was enough, as the players perform their moves sequentially in turns. An example of games using this mode are *Heroes of Might and Magic* or *Worms* series.

- *Split-screen multiplayer* – type of a multiplayer game, an alternative to online modes consisting in the fact that several players (usually two to four) play on one machine at the same time, and the game is presented on one screen, where each player appears in a separate place on the screen. The mode is often found in case of console games, such as Quake II for Nintendo 64 or PlayStation, or Mario Kart for Nintendo
- *Networked multiplayer* – multiple instances of the game are run on multiple devices (each player has one) connected to the network, e.g., local network or Internet. Thanks to the fact that each player has its device, controller and monitor, this mode is ideal for games that use the graphical capabilities of the device, such as three-dimensional games while requiring minimal response time (the quality of the network connection is also essential). The most popular games that enable networked multiplayer include FPP (*Call of Duty*, *Quake*, *Unreal Tournament*, *Counter-Strike*), TPP (*Uncharted*, *Neverwinter Nights*), RTS (*Warcraft*, *StarCraft*) and many more.
- *Massively multiplayer online game (MMOG)* – type of a multiplayer game played by a vast amount (thousands, or even millions) of players at the same time via the Internet. This type can include more features than networked multiplayer games, such as extensive communication system, a more complex fighting system, trade, and developed economics, or different policies. The game servers are hosted in central servers owned by the developer of the game.

Multiplayer games are very similar to singleplayer games, but with the possibility of playing with many people. In truth, a lot of singleplayer games that allow for any multiplayer mode are developed as an instance of a multiplayer game hosted by the player's device and allowing only the player to play the game. The reason of it is that transforming a singleplayer game into a multiplayer game can be very difficult as it affects most of the game engine's components such as rendering component, human input device system, animation system, etc. On the other hand, transforming a multiplayer game into a single player game is a much easier task.

1.7.15. Gameplay foundation systems[12]

The gameplay is a single or divided into more than one session game player by the player from the beginning to the end. It also consists of elements that the selected computer game offers to the player. The gameplay is implemented on the base of a game engine, usually using the same programming language that was used to develop the engine or a scripting language. The gameplay foundation system is created to distinguish the gap between the game engine and the game. It provides the developer functions that can be used to implement a game logic for the game.

1.7.15.1. Game worlds and objects models

The game world is a virtual world in which the action of a game takes place. It consists of a scene that contains a lot of different static, or dynamic objects, such as:

- Player characters (PC)
- Non-player characters (NPC)
- Static background geometries, like houses, roads, terrain, etc.
- Dynamic rigid bodies, like rocks, furniture, etc.
- Vehicles
- Weapons
- Bullets
- Lights
- Cameras

etc.

1.7.15.2. Event system

Event system allows for communication between game objects. When an event occurs, the object informs another object about the event by sending a message using a function of the receiving object. In an event-driven architecture, the sender creates a message (event) structure, including a type of an event and parameters to be sent. The message (event) is then sent to a receiving object by calling its responsible for message handling function. If there are many messages, the events can be queued for later use.

1.7.15.3. Scripting system

A scripting system allows the developers to use a scripting language to ease the implementation of gameplay. When using a scripting language, there is no need to recompile the whole game to run it. Some engines allow to load the script without the need of restarting the game, and some require the restart for the script to work.

1.7.15.4. Artificial intelligence foundations

Artificial intelligence was first used in games very early. Some of the very old games, like PacMan, had already implemented it. However, it wasn’t always recognized as a part of a game engine. AI was usually developed into a game, not an engine because it varied from game to game. With the newest game engines, artificial intelligence started to be implemented into the engines and SDKs, as the developers started to find patterns that can be used in many different games.

For example, a company Kynogen created a middleware artificial intelligence development kit called Kynapse. It consisted of a complete 3D pathfinding, an automatic AI data generation tool, optimizations for multicore/multiprocessing/cell architectures, spatial reasoning, streaming mechanisms to handle vast terrains and the management of dynamic and destructible terrains. This technology was bought by Autodesk and replaced by a new artificial intelligence development kit called Gameware Navigation that allowed designers to create game lighting, character animation, low-level pathfinding, high-level AI and advanced user interfaces. However, the project was discontinued in 12th July 2017 and removed from the online store of Autodesk.

1.7.16. Game-specific subsystems

Game-specific subsystems are components that vary from game to game. These include gameplay systems that can be connected with a player's character movement, in-game cameras, the artificial intelligence of NPCs, weapon systems, vehicle systems, etc. For example, a real-time strategy game can implement a small inventory system for a character, where one slot represents one item, while an RPG game may require an inventory system, where one item needs to be represented by more than one slot in order to restrict the player from carrying too many items at once.

2. Project

As the game engine is a very complex software application, it was designed with the use of layered architecture to separate the components responsible for individual tasks. The layers are mainly stored in a game loop's class, which allows for quick access and helps to monitor which component is dependent on which to render a frame of the game.

The layered architecture, thanks to the decomposition of the system into independent components, allows for easier profiling and analysis of each component's performance. It also opens the possibility of introducing the concurrency in order to save resources.

The project includes definitions and rules for both types of engines (single and multithreaded) as both engines will be implemented in order to compare the efficiency and performance of the engines. Most of the subchapters include common parts of both types of engines. Subchapters specific to a specific architecture will have a distinct division in the name of the chapter.

2.1. Engine support systems

The main goal of developing a game engine is to make it as efficient as possible in order to meet the increasing needs of players. Besides different algorithms that need computing power, the approach of how the developer utilizes the memory can have a huge impact on the performance of the game engine.

2.1.1. Dynamic memory allocation

Dynamic memory allocation is a very slow operation as the system needs to switch from the user context to the kernel's context in order to allocate the memory and then switch back to user context, back to the program. It also needs to find a big enough block of free memory in order to save the provided data on the heap. In order to increase the performance of a game engine, the dynamic memory allocation should be avoided, or the developer should provide custom memory allocators to reduce the costs of allocations of data.

2.1.2. Memory access patterns

In memory due to memory access patterns (the pattern of reading/writing memory on storage), hugely impacts the performance of an application. If the data is located close in small blocks of memory, the code will execute much faster than if the data is stored in big chunks across the whole range of memory addresses.

2.1.3. Memory fragmentation

After launching an application, the heap memory is empty, but the memory is quickly filled with data as objects are allocated. Some of the allocated objects stay longer before being deallocated than others, which creates small gaps of free memory between memory blocks. The longer the applications run, the more allocations and deallocations happen, which may eventually create so many gaps that the allocating system will not be able to find a big enough contiguous memory block in order to allocate an object in it and may cause the software to crash. This introduces the problem of memory fragmentation, which should be avoided by carefully identifying and planning on how the data will be stored for each custom data type defined in the game engine.

2.1.3.1. Stack and pool allocators

Stack allocators are used in order to avoid fragmentation. They first allocate a specified amount of memory, while holding a pointer pointing to the first occurrence of free memory in it. The allocation takes place through putting a data onto a stack of the free memory and relocating the pointer to an end of an allocated object. The objects need to be deallocated from the end of the stack in the opposite direction they were added. The developer can place a marker in a position of an allocated part of the memory (by default it is the beginning of the stack). When deallocating, the memory is freed from the end of the stack to the place of the marker. Stack allocators are usually used for data structures that can vary in sizes, mostly for in-game objects created for a specific scene, so that after the level ends, they all can be deallocated from memory.

A pool allocator divides the memory into blocks of equal size. It allocates a specified portion of memory, often based on the number of elements the developer wants to store. For example, if the developer wants to store a 4x4 matrix, it will require 16 elements * 4 bytes = 64 bytes of memory. The memory can be allocated and deallocated in any order, as it will always be possible to find free memory for the object of 4-byte size (provided that the memory is not fully occupied). This solution allows for memory fragmentation, as it will never cause an out-of-memory error due to missing memory block big enough to hold the object.

2.1.3.2. Defragmentation and relocation

Defragmentation is needed when stack and pool allocators cannot be used, that is when the objects of different sizes are allocated and deallocated in random order. Defragmentation consists of moving already allocated memory data from higher addresses into lower addresses of memory in order to remove the gaps of free memory blocks.

The relocation is a method used together with defragmentation in order to change the addresses pointed by pointers, so that they point to the same objects, but which are located under a different memory address after defragmentation. Such a solution requires developers to manually track used pointers as there is no general rule or a possibility to find and relocate all pointers of moved objects. The relocation is not always possible, as when using a third-party library that creates pointers to its object, it is possible that the pointers to its data structures will not be relocatable.

2.1.4. Engine configuration

Engine configuration should allow the player to change the configuration of the game engine, such as changing keybindings, changing the volume of sound and music, and so on. It should also provide development settings for the developer, like setting the maximum framerate, changing the speed of the character, or the game. The configuration should be stored in a memory in global variables or singleton class, but also provide the option to save it when some option is changes as well as load all options on launching of the game.

Possible methods to store configuration include:

- Text configuration files – the configuration is stored in a flat file with custom pattern matching in order to save/load an option, INI file (key=value), XML or JSON format
- Compressed binary files – the configuration is compressed into a binary file in order to reduce the needed memory. It is used mainly in console games that require an additional memory card for game saves
- Windows registry – the configuration is stored in the registry key of a predefined registry key. This solution can be problematic as the registry can easily become corrupted.
- Command line options – command line allows to execute and set different options of the game engine by typing them in. It often implements a mechanism of suggesting option names when typing.
- Environment variables – the configuration is stored as an environmental variable under a PC system
- Online user profiles – an online profile for a player is created and the configuration is stored in a cloud. This method may also include storing other information about the player like achievements, or progress in the game.

2.2. Resources and file system

As games require many different assets like sounds, 3D mesh data, materials, textures, animations, game world layouts, etc., there arises a need for efficient resource handling. Due to a limited amount of RAM, game engines need to load shared resources only once in order to minimize the amount of needed memory. Most of the time it is impossible to load all needed assets into the memory at the same time, as that would be problematic in case of big games that have tens of gigabytes. To solve the problem, a *resource manager* (or *asset manager*) will be introduced to handle and manage the loading of resources used in a game.

The resource manager will wrap existing, native file system libraries to provide a programming API that will allow the developer to use cross-platform games. It will also allow for data streaming to load the data whenever it is needed.

2.2.1. File system

The file system should allow to:

- create, rename and move files
- open, close, read and write to a file
- scan the contents of a directory
- handle asynchronous file input/output requests (used in streaming)
- support different path formats (Windows and UNIX)

Streaming is a *process of loading data in the background while the main program* is running. It is mostly used for audio and texture files but can be used to load any resource needed.

The asynchronously loaded resources must have a priority. For example, if the engine is streaming audio and playing it from a hard drive, loading the next buffer of audio data should have a much higher priority than loading a texture for some distant object in the game's world. It should provide a possibility to suspend lower-priority requests, so the higher-priority resources can be loaded before they are used.

Asynchronous file handler should run in a separate thread. The main thread sends a resource load request that is put onto a queue and continues execution of the code. *The handler's thread picks the requests from the queue and handles them sequentially.* When the request is completed, a callback to the main thread's function is called, notifying that the operation was finished with a result (succeeded/failed).

The system can also be reused to transform any synchronous operation *into an asynchronous operation by moving the code into a separate thread, or by running it on a separate processor.*

2.2.2. Resource manager

The resource manager is responsible for managing the previously mentioned resources like textures, materials, light data, etc. The resources should be managed outside of the game engine with the use of offline tools as well as during the runtime of the engine.

In the project, the resource manager is a subsystem that will be used by all classes that require access to resources. It shall work with every type of resource used in the game and provide functionalities allowing to handle them optimally.

2.2.2.1. Revision control for resources

There are two main possibilities to store resources:

- Source control – files are kept under source control, like git
- Custom resource management tools – simple wrappers around source control, or a new implementation of it

The project uses already existing source control – git – that allows for storing even large data sizes. The engine's revision control subsystem should allow for automatic recognition of new files and help the developer control the resources that are required by the game engine while discarding files that were never referenced. It may also allow the developer to download only the resources needed for a specific part of the code to work to minimize needed bandwidth when the project is stored in an external repository.

2.2.2.2. Resource database

Not every data in a resource is needed by the game engine. For example, when using a file with animation data, the engine might need only specific frame ranges in order to display the animation correctly. To accomplish it, before resources can be used by the engine, they need to be preprocessed through an asset conditioning pipeline to be converted into a binary form that can be loaded by the engine.

The resource database shall hold metadata describing how the resource should be processed, or what game object uses given resource. The metadata can be stored inside a resource, but it can also be stored in an XML file, custom GUI, or even relational database like SQLServer.

Functionalities of a resource database:

- Handle many types in the same way
- Create, delete, inspect and modify existing resources
- Move resources from one path to another
- Allow cross-references to other resources
- Maintain referential integrity of all cross-references
- Log the changes in the database
- Search/query for resources

2.2.2.3. Asset conditioning pipeline

Resources created in external tools almost always contain data that is not interesting for the game engine. Asset conditioning pipeline preprocesses the resource's data to extract only the needed information before loading it in the engine.

The asset conditioning pipeline should consist of:

- Exporters – custom plugin, custom program, or part of a game engine that's responsible for exporting needed data from digital content creation (DCC) and save it in a file of custom, recognized by the engine type.
- Resource compilers – compilers are used to preprocess the exported data so it can be used optimally by the game engine. Implementation of resource compilers is not always needed as most exporters provide resource data that is game-ready.
- Resource linkers – linkers combine multiple resources into a package that can be used by the game engine. It is useful to link assets when creating custom meshes that are created by connecting more than one mesh or including a package of multiple animations for the character. Not all resources require to be linked.

The game assets may have interdependencies which creates a need to preprocess the resources in a specific order starting with a resource that does not have any dependencies and ending with a resource that has the most dependencies. Additionally, when an asset that has dependencies needs to be rebuilt, then all assets it depends on, also need to be rebuilt.

2.2.3. Runtime resource management

Features of runtime resource manager:

- store only one copy of each resource in memory
- manage the lifetime of each resource
- load needed resources
- unload resources that are no longer needed
- load composite resources
- maintain referential integrity
- manage the memory usage of resources
- allows for custom processing of loaded resource
- handle data streaming

2.2.3.1. Resource file formats

Most of the resources shall be stored in a standardized format, like Windows Bitmap files (BMP), Joint Photographic Experts Group files (JPEG), or in standardized compressed formats like DirectX's S3 Texture Compression. A custom file type may be needed if the standardized formats do not allow to store all necessary data for the game engine.

2.2.3.2. Resource GUIDs

GUIDs (globally unique identifier) is a unique identifier, responsible for mapping a resource to a file on a hard drive. The game engine will use the most commonly used GUID, which is simply the file's file path.

2.2.3.3. Resource lifetime

There are different resource lifetime needs. Some objects need to be loaded together with the game and kept in memory until the game's process closes. This need introduces the definition of *global resources*, which are usually connected to the player's character, such as the character's model, textures, materials, animations. Every object that will be used for the entire gameplay should be loaded as a global resource.

Some of the resources should be loaded only for a particular level of the game. After the player's character leaves the level, the memory allocated for the object should be freed.

In case of resources that are used for a shorter time than the existence of a level, e.g., resources used for a cutscene (animations, voices) should be loaded before the cutscene begins and freed after the cutscene ends.

Resources like background music and videos should be streamed, which means that parts of the resources will be loaded into the memory's buffer only for a very short time (even fraction of a second) and released right after being played.

2.3. Game loop

2.3.1. Rendering loop[13]

In 2D games, static graphics are not drawn by the renderer with the use of a *rectangle invalidation*. However, in 3D games, such an approach is not possible, due to the use of a camera that constantly changes, moving all the contents of the game scene on the screen. The constant change of objects can be represented by a rendering loop.

Steps of a rendering loop:

- update the camera
- update positions of objects
- render the scene
- swap back buffer with a front buffer in order to render the calculated scene

2.3.2. Singlethreaded game loop[14]

Game loop is the hearth of a game engine. It controls every component in the engine and keeps track of *game time*. The game time is a class object that stores the last time a frame was rendered in milliseconds and provides functions to check how much time passed in comparison to it.

In a single-threaded engine, one iteration of a game loop is often equal to one frame that is rendered. To limit frames per second, developers reduce the number of iterations per second by monitoring game time and waiting for a specific period before the next iteration. For example, for a 60-fps game, the time between frames equals about 16.66ms.

Typical steps of a game loop in single-threaded game engine:

- handle inputs
- handle game logic
- check collisions
- render the graphics
- play sounds

The game loop should be located and executed in the main thread.

2.3.2.1. Windows message pumps

Windows message pump should handle all messages submitted by the user with the use of human interface devices (mouse clicks, keyboard buttons), as well as system messages (paint, print message). The messages should then be translated to an instruction, the game engine can understand and dispatch to a proper component for further handling. After all, messages are processed, the engine is ready for execution of next game loop iteration.

2.3.2.2. Callbacks (optional)

In order to provide users a system that will allow for running additional code on the low-level of the game engine, callbacks and listeners should be created. Use of callbacks makes the game engine more of a framework than just a library, allowing the user to add functionality to, usually simple, modest game loop.

The user should have a possibility of creating listeners, adding a functionality, that are executed from inside the game loop. For example, the user extends a FrameListener class that allows him to override its functions: onFrameStart, onFrameEnd, which will later be called by the game loop.

2.3.2.3. Events

An *event* represents an occurrence of a change in the game's state, or its world (ex. Input from hid, the collision of objects). An *event system* detects such an occurrence and allows the developer to write a code that will handle an event by creating delegates.

2.3.3. Multithreaded game loops[15]

2.3.3.1. Task decomposition

Instructions from the game loop need to be decomposed so that they can run in parallel. The decomposition should transform the game engine from a sequential to a concurrent program.

Two main approaches of task decomposition:

- task parallelism
- data parallelism

Task parallelism should be utilized when multiple different operations need to be executed. For example, collision detection can be run in parallel with an update of the game's state.

Data parallelism should be used when a single operation works on big data to divide the data into smaller batches and run algorithms on them in parallel to significantly increase the calculation speed.

2.3.3.2. Each subsystem on a separate core

In this approach, each subsystem is assigned to a separate core. If there are more cores available than there are subsystems, then chosen operations can further be decomposed onto all remaining cores.

The main thread would manage and synchronize the calculations of each subsystem. The game loop would still exist and work as in a single-threaded approach.

2.3.3.3. Scatter/Gather

Scatter/gather approach consists of data parallelism. It uses a divide-and-conquer approach, which consists of dividing the data into batches and running operations on them on multiple processor's threads (one thread per batch). After performing all needed calculations, the results are then gathered, combined and ready for further processing.

In the game loop, the main thread, for each frame, would divide a dataset into batches and run operations on it on multiple threads and cores. The thread workers will update the data, or produce an output that will be stored in a buffer for later use after all threads finish their work.

After division of the data to threads, main thread should continue performing its tasks while waiting for the results from its created threads. When all threads return the output before main thread finishes its work, the main thread shall gather all the data and, if needed, perform additional calculations on it. On the other hand, if the main thread finished its work before worker threads, the main thread should be put to sleep and wait for the threads to finish before handling the next frame.

The scatter/gather approach should employ *single instruction, multiple data* (SIMD) architecture, which consists of single instruction operating on multiple data streams (executing same series of instructions on multiple, independent datasets at the same time).

2.3.3.4. Job system

A *job system* is a system consisting of multiple thread workers that are running continuously. Each thread contains a queue of jobs, which can be submitted to a worker. A job is a small, independent, well grained task, which can be performed without any need of synchronization between another tasks, or synchronization between common objects.

Jobs should be independent from each other in order to maximize the processor's capabilities. There should exist a possibility to assign a priority to a job. Jobs with higher priority should be executed first.

In theory, each iteration of the game loop should be divided into many small jobs and distributed among thread workers for execution by adding them to a worker's job queue.

For each CPU available on used system, only one thread should be created and locked with the use of affinity to one core.

2.3.3.5. Job system with dynamic thread worker allocation

The job system with dynamic thread worker allocation should monitor the time of execution of each of the subsystems/all of its jobs in order to decide which of the subsystems requires the most of processing power. It allocates then more thread workers to the job that needs the most in order to try making all the subsystems finish at the same time.

2.3.4. Time

2.3.4.1. Real time

Real time is measured based on the CPU's timer register represented in units of CPU cycles counter from the moment, the CPU was turned on. The unit can be converted to seconds by multiplying it by the frequency of the timer.

2.3.4.2. Game time

Game time is a separate, independent timeline created by the developer. It should reflect the time in a game with the use of *ticks*. A tick is a predefined fraction of time (ex. 1 tick = 1 second).

The game time shall provide functions to pause the game (stop updating the game's ticks), slow/speed the game (extending/reduce the length of a tick).

Operations on game time should not affect the game loop's execution to allow some elements of the game run while the game is paused (ex. launching the game's pause menu should not prevent the player from interacting with the game's menu),

In order to enable debugging, game time should provide functions allowing the developer to move the camera when the game is paused and stepping the game's time by a custom value in order to precisely check what is happening in that time.

2.3.4.3. Frame rate

The *frame rate* is a number representing how fast the game's scene is being refreshed. The unit of frame rate is Hertz (Hz), which defines the number of cycles per second. The equivalent of Hertz in games is called *frames per seconds* (FPS).

2.3.4.4. Delta time

The *delta time* is a time between the last two frames. It should be measured with each iteration of a game loop. For 60 FPS game, the delta time equals 1/60 of a second, which is 16.6ms.

2.4. Human interface devices

The game engine needs to support various human interface devices, like keyboards, mice, game pads, and so on. It should also provide functionalities allowing to detect different combinations of buttons, detect long presses, chords and sequences.

All human interface devices provide input to the operating system, which should be intercepted and handles within the game engine. Some of the devices can also handle an output from the game engine and provide it back to the user (vibration, sounds).

Types of inputs:

- digital buttons (keyboard, gamepad buttons) – buttons that have only two states: pressed and not pressed (down and up)
- analog axes and buttons (analog sticks, joysticks) – provides *analog input* which can take values in a specific range instead of only presenting states like digital buttons. It allows the developer to check how hard the button is pressed/moved

- relative axes (mice, mouse wheels) – devices of this type return 0 if the state of the device is not changing. When the state changes, the returned value is relative to the previous position from the last input's value.
- Accelerometers – provides relative analog input in a three-dimensional space (x, y, z)
- 3D orientation (Wiimote, DualShock) – built with multiple accelerometers (usually 3) in order to represent the exact position of a device in the players hand
- cameras (Sony's PlayStation camera, Microsoft Kinect) – infrared sensor capturing infrared signals from the player's controllers in order to determine the location of the controllers in space

Types of outputs:

- rumble – allows the controller to vibrate usually by the use of motors
- force-feedback – a motor adding resistance to the player's controls
- audio – allows the game engine to play a sound using controller's audio system
- other

2.4.1. HID requirements

- event detection (key down, key up) – game engine should detect changes in the states of the devices
- sequence and chord detection – game engine should detect if multiple buttons were pressed at the same time (chords) or in a specific order (sequences)
- dead zone handling – dead zone is a specified threshold for analog devices' input. If the value of the input is inside the threshold, it is treated as 0 to reduce the input's noisiness
- analog signal filtering – analog signal needs to be filter in order to make it more natural. The most common filter is a *low-pass filter*
- multiple human interface devices handling – game engine should map a controller to a specific player's character
- cross platform support – game engine should make use of abstraction in order to avoid hardware specific details
- input remapping – game engine should allow players to modify button to action mappings
- input disabling – game engine should provide a possibility to disable and enable input from a player. It can be used when playing cutscenes, so the player can't move during them
- context-specific controls – game engine should allow the developers of a game to specify different actions based on the object the player interacts with.

2.5. Debugging

To ease the process of development of a game and a game engine, different debugging tools are usually developed in order to help reduce the number of bugs, speed up the development and display useful information.

2.5.1. Logging and tracing

The game engine should have a system allowing the developer to print different types of messages with specific verbosity levels (e.g. info, debug, warning, error).

2.5.1.1. OutputDebugString

As the game engine is written in Visual Studio, the debug log should be printed using an OutputDebugString function, which prints messages to Visual Studio's Debug Output window. The function should be wrapped into specific functions to keep the log's consistency and allow for formatted output.

2.5.1.2. Verbosity

The game engine should allow to specify a verbosity level on which debug logs with different verbosity levels should be printed. On the lowest verbosity level, only the info, warning and error logs, on the debug verbosity, debug logs should be printed additionally.

2.5.1.3. Channels

Debug messages should be categorized using channels. Each subsystem should have its own channel allowing the developer to choose (using filters) what debug messages are printed by the game engine. Each channel can have its own color in order to distinguish from which component each message comes.

In order to save memory, channel flags used for filtering should be stored inside a 32, or 64-bit mask, where each channel occupies only one bit of the mask.

2.5.1.4. Output logs to files

All printed debug messages should be stored inside of files for further analysis. Files should be created per channel with a timestamp of each message. All the logs should be saved to a file, independently of the verbosity and filtering settings to allow the developer to store every event that happened even in case of a crash.

The debug logs should be *flushed* (saved from a buffer to a file) right after the event that is logged happens, so no information is lost in case of unexpected behavior.

2.5.1.5. Crashes

In case of a crash, the information about it should be gathered and saved to a file as well as current state of the game engine should be dumped into a dump file for debugging.

Typical crash informations:

- stack trace – shows last called functions that lead to the crash
- state of memory allocators – shows the amount of memory used by the game engine, free memory space, values stored in an allocator
- location of a player
- screenshot of the game
- game-specific informations

2.5.2. Debug drawing

Debug drawing is a set of functions allowing the developer to draw lines, primitives and text in a 3D space. It is very useful in development as it allows to visualize all the mathematical formulas used in the game engine.

The debug drawing allows the developer to save much time as instead of manually calculating why the code he wrote doesn't work, he can just draw a line and see what the problem is.

Debug drawing features:

- drawing lines
- drawing simple geometry
- drawing points
- drawing 3D text
- drawing wireframe instead of whole object
- drawing collision
- provide ability to change color, line width, etc.

2.5.3. Debug menus

A debug menu is an in-game menu that allows for real-time configuration management. The developing team can change any option added to the game engine, which significantly reduces the debugging time as different features can be tested without the need of compilation and linking of the game engine.

It should be possible to open the debug menu with a single button on a keyboard (usually a function key). Opening of the menu should pause the game, so the developer doesn't miss the window of frames that he wants to debug.

Debug menu options:

- enable/disable debugging features
- modify values used in algorithms
- modify properties of in-game items
- call game engine's functions
- arrange the menu by grouping of its functions

2.5.4. Debug console

The debug console works on a very similar concept as the debug menu, but instead of providing a clickable graphical interface, it allows the developer to enter predefined commands that change the game's behavior.

Example commands:

- set <OPTION> <VALUE> - sets the value of an option
- get <OPTION> - gets the value of an option
- execute <FUNCTION> - executes game engine's function

2.5.5. Debug camera

A debug camera is a camera that can be either detached from the player or a completely new instance of a camera that allows the developer to fly around the game's world outside of the character's body. The debug camera can be used when the game is paused, but also when it's running.

2.5.6. Time manipulation

Time manipulation is very useful when debugging visual elements of the game. The developer should have a possibility to slow/accelerate the game's time, pause it and single-step the game by specified number of frames.

2.5.7. Cheats

Cheats allow the player to play against the rules of the game. Most common practice is to add cheat commands to a debug console. Cheats are very useful when debugging some types of games. For example, in an RPG game, where the character can be attacked, the developer can enable an invincibility cheat, so the character never dies and continue debugging without being disturbed.

Other frequently used cheats include:

- giving item to the player's character
- ability to fly
- infinite ammunition
- healing player's character

2.5.8. Screenshots

In order to take a screenshot of the game, the game engine should make a call to Windows' graphic library that will copy the contents of video RAM to the main RAM, which can then be saved into a file. The file should include a timestamp when the screenshot was taken and placed into a predefined folder.

2.5.9. Profiling

Profiling is used to measure the game engine's performance. The profiler provides functions allowing the developer to mark parts of code that should be analyzed and name them. It measures the execution time of each marked block and stores the results in memory. The data is then displayed in a various form on the game's screen.

The profiler should consider the hierarchy of function calls. If a function calls another function, the profiler should be able to differentiate the time needed to perform the function including its children (*inclusive* execution time) from the time needed only for the function (*exclusive* execution time). The exclusive execution time of a function can be measured by computing inclusive times of each of the function's children and subtracting it from inclusive time of the function.

Profiler's functionalities:

- analyze CPU usage
- analyze GPU usage
- analyze memory usage
- provide call stack
- export data to Excel spreadsheet

2.5.10. Memory statistics

The game engine's debugger should provide information about how much memory is used by a subsystem. The information can be displayed in a form of a table, or a graph.

As the game engine can't track and manage the whole memory i.e. memory used by external libraries, there is a need to secure the engine from out-of-memory errors. This introduces a need to specify default actions that should occur when the game tries to load a resource, but there's no more available memory.

Example actions in case of out-of-memory error:

- if an animation fails to load, the default pose of a character should be used
- if a texture fails to load, the texture should be replaced by a pink texture
- if a model fails to load, a red 3D text should be displayed in the place of the object

2.6. Rendering engine

Steps required to render a 3D scene:

- describe the virtual scene
- position and adjust the camera
- define light sources
- describe the visual properties of the surfaces
- calculate color and intensity of each pixel's light rays (rendering equation)

2.6.1. Scene description

The scene is created of different objects, like solid blocks, fluids, smokes, or particles.

Types of objects based on light:

- *opaque* - the light can't pass through the object
- *transparent* – the light can pass through the object, but isn't scattered
- *translucent* – the light can pass through the object, but is scattered into multiple directions

To render an opaque object, only its surface's parameters are needed as it doesn't allow the light to pass through. Light for transparent and translucent objects on the other hand needs to be calculated. This includes calculation of reflection, refraction and absorption of the light by the objects. In order to simplify the implementation of the light, an alpha value will be used to specify how much the object is transparent.

2.6.1.1. Triangle mesh

Triangles are the simplest form of polygons that can be used to create a surface. The surface of an object should be divided into triangles using the process of *tessellation* (division of a surface into polygons), or to be more specific *triangulation* (division of a surface into triangle polygons). The closer the object is, the more triangle polygons should be used to increase the smoothness of an object.

Triangle list

Triangle list is used to describe the object's mesh by creating a list of vertices, where each set of three consecutive vertices create a triangle polygon.

Indexed triangle list

Indexed triangle list is a structure similar to triangle list, but without duplicated sets of vertices. It helps to reduce the need of transforming the same vertex multiple times by the GPU by creation of *indices* stored in an *index buffer*, while the vertices are stored in a *vertex buffer*. The buffers are used by the graphics library (DirectX) and later in a GPU to finally render an object in the scene.

2.6.1.2. Textures

Textures, also called a *texture maps* are bitmapped images containing information about the color. In order to apply a texture, the developer needs to load it into the video memory and projects it onto a triangulated mesh.

Types of textures:

- *diffuse map* – describes the surface color at each texel (texture pixel)
- *normal map* – stores unit normal vectors for each texel
- *gloss map* – describes the brightness for each texel

Textures should be applied based on a two-dimensional coordinate system (*texture space*) represented by a normalized pair of numbers, ranging from (0,0) to (1,1). The pair (0,0) denotes a bottom-left corner of a texture, while (1,1) denotes the top-right corner.

2.6.1.3. Materials

A *material* describes the properties of a mesh like used textures, shader-specific properties and other properties used by the graphics acceleration. Together with a mesh information, a material contains all information that is needed to render an object. Such a pair of structures is called a *render packet*.

2.6.1.4. Shading[16]

Shading is a connection of lighting and visual effects. It is a part of rendering engine that is responsible for preparing the surface of an object for rendering. Shading includes deformation of vertices, simulation of water or hair.

2.6.1.5. Lighting[17]

Local lighting

Local lighting affects only single objects. It is used to provide realistic impression of light falling on an object. The local lighting is created by the use of a *direct light*, which is emitted onto an object, bounces from it and is reflected to a virtual camera.

Global lighting

Multiple object take part to create a global lighting. In case of global lighting, an *indirect lighting* is used, where the light bounces multiple times from multiple objects before reaching the virtual camera.

2.6.1.6. Virtual camera

The virtual camera is a focal point consisting of a *imaging rectangle* (a rectangular virtual sensing surface) positioned in front of it. Each pixel on a screen is thought as a light sensor that determines the color and brightness of the pixel.

2.6.2. Rendering pipeline

The rendering pipeline is a set of ordered procedures responsible for transformation of input data into an output data for the rendering.

Each of the procedures can be executed independently from another, which allows for a parallelization of the tasks. When the first procedure is busy calculating a result for on of the input data elements, the second procedure can start working on the first procedure's output, decreasing the overall execution time of the rendering pipeline.

In addition of parallelly executed procedures, the data computation inside each of the procedures can also be run in parallel. The procedure should be run onto N threads, where each thread is processing one data element.

To measure the performance of a rendering pipeline, its *throughput* will be calculated. The throughput defines how many data elements were processed in a time of a one second. Additionally, a *latency* of the pipeline will be taken into consideration. The latency indicates how long does it take to process one data element within the entire pipeline. The latency will be also calculated per each of the pipeline's procedure allowing to measure an execution time of each procedure.

Procedures of a rendering pipeline:

- *tools procedure* – defines geometry and surface (material) properties
- *asset conditioning procedure* – processes the properties with an asset conditioning pipeline
- *application procedure (CPU)* – identifies visible objects and submits them to GPU together with material data
- *geometry processing procedure (GPU)* – transforms vertices and processes triangles
- *rasterization procedure (GPU)* – converts triangles into shaded fragments that are passed through different tests, and later into a frame buffer

2.7. Animation

Animation system is responsible for giving the game's characters life. The character animations allow a character to perform specific movements that often reflect movements of real-life personality. Animations can also be used for different objects to push them into movement.

Types of animations:

- *sprite animation* – derived from a cel animation, also known as *traditional animation,* that is created by drawing each frame of the animation on a cel and displaying them in a sequence, which gives the impression of movement. The sprite animation replaces the cells with sprites (small bitmap image). The sequence of sprites is drawn so that it can be looped indefinitely (*looping animation*)
- *rigid hierarchical animation* – used in first 3D games, a rigid hierarchical animation consists of dividing an animated object into multiple, connected with each other rigid pieces. The connections simulate joints of a character, which allows to make the animation more realistic to a humanoid. The rigid hierarchical animation produces an animation, where the joints are "snapping", which on some models can look unrealistic

- *per-vertex animation* – the animation is created by an artist, usually in an external tool, and exported to the game engine. This animation type is memory-heavy, because of the need of storing the data before its use
- *morph targets animation* – created based on per-vertex animation. The animator defines a small set of poses for a character. The animation is created by calculating the movement of the character from one pose to another (linear interpolation between the vertices)
- *skinned animation* – similarly to a rigid hierarchical animation, consists of a skeleton built from bones (rigid bodies). The bones themselves are not rendered. Instead, a *skin* is created between the joints. The skin is represented as a triangle mesh and has an ability to stretch with the movement of the joints to simulate a natural skin behavior.

2.7.1. Skeleton

A skeleton is created by a hierarchy of joints connected with bones. The joint are modified by an animator and are identified by an index. Each joint in the hierarchy can have only one parent, where the root joint has none.

The skeleton should be stored as a data structure containing an array of data structures representing joints, where the joints are stored in a hierarchical order (the next element is always a child of previous one).

The joint structure should contain a name of the joint, an index of its parent and an inverse position of the joint in a skeleton.

2.7.2. Pose

A *pose* is a position of a joint at a given time relative to its original position. The game engine should display one pose per frame, which is 60 poses in a 60 FPS game.

A skeleton's pose is created by transforming (positioning, rotating and scaling) all of its joints, which means it is a collection of poses for each joint that created the skeleton.

2.7.2.1. Blind pose

A *blind pose*, also called as a rest pose, or a reference pose, is a skeleton's original pose. The blind pose for most humanoid characters takes a form of a t-pose, where the skeleton looks like a letter T.

2.7.2.2. Local pose

A *local pose* of a join is a pose relative to the joint's parent. It allows for better join management, as when the joint is moved, every its descendant will also be moved.

2.7.2.3. Global pose

A *global pose* of a join is a position of the joint in the world.

2.7.3. Skinning

Skinning a mesh requires additional informations are needed: *indices* of the joints and a *weighting factor* of a joint. The indices describe what joint or joints a vertex of a skin should be bound to. The weighting factor is used to calculate a final position of a vertex, which is represented as a weighted average of the positions of joints indexed to the vertex (if only one joint is bound, then the final position of the vertex is equal to a position of the joint).

The number of maximum joints that can be bound to a vertex should be limited in order to reduce the workload of the CPU and memory load.

2.7.4. Animation blending

Animation blending is a process allowing a use of multiple animations to create a unique, final pose of an object. The animation blending is omitted in this work to simplify the overall process of result gathering.

2.7.5. Animation pipeline

An animation pipeline, similarly to a rendering pipeline is a sequence of procedures used to process an input animation data into an output data that can be used for rendering.

Procedures of an animation pipeline:

- *pose extraction* – extracts a pose for a current frame
- *pose blending* – blends the animations together to determine positions of joints. Only executed if there's more than one animation
- *global pose generation* – connects are joints together to produce a final pose for a frame
- *post-processing* – modifies the pose using a specified technique. The procedure is optional and used mostly when the physical forces act on the object
- *recalculation of global pose* – required if post-processing was used to calculate a global pose from local pose outputs produced by it
- *matrix palette generation* – converts the global pose into an inverse bind pose matrix used by the game engine

2.8. Physics

Features of a physics system[18]:

- collision detection
- simulation of forces
- rag dolls
- trigger volumes
- vehicles
- machines
- fluid simulations
- cloth simulations

2.8.1. Collision detection[19]

The collision detection system is responsible to detect if any of the game's objects (both dynamic and static bodies) touch each other. The detection is based on simple geometric shapes (spheres, boxes and capsules) that are imposed onto an object. The shapes act as an invisible shell of an object, which can be used for collision detection by checking if the shapes of two objects are intersecting with each other.

The collision detection system should provide a detailed information about how the object intersect, which can be used in order to correct the positions of objects that are interpenetrating each other before rendering. It should also provide a possibility to create objects that are not affected by a collision, but instead allow the developer to decide what will happen if the objects collide. It can be useful if creating different types of pickups in a game, such as health packs, or weapons.

2.8.2. Rigid body dynamics

Rigid body dynamics, also called a *physics system* has a purpose of simulating physics, as well as a motion of objects in a game. The physics system should provide a set of function allowing to determine dynamics (forces that act on an object, changing its movement).

3. Implementation

3.1. Tools

3.1.1. Programming language – C++

C++ is a general-purpose programming language that was designed and implemented by Bjarne Stroustrup in 1985, which provides object-oriented, generic and imperative programming features, while also giving possibility of low-level memory manipulation. The C++ programming language is usually used when it comes to programming embedded systems, operating systems, drivers and artificial intelligence, but also sometimes to make applications for phones. The language is also very popular when it comes to developing game engines, because of the possibility of memory manipulation, Windows, DirectX and OpenGL libraries.

It is a compiled language, which means that to run the program the source code must be compiled with a compiler, which produces object files and later connected with a linker to produce an executable file, which can be launched. The file is produced for a specific platform or hardware and it is impossible to run it on another system, however C++ language is considered to be a universal programming language, which means that it can be used widely on many platforms. It's because the portability comes from the possibility of easy transfer and compilation of source code on any platform and for any system.

The C++ is a statically typed language, which means that the type of each variable must be known to the compiler when being used. Depending on the type we can use a specific set of methods or operations. In comparison to soft-typed programming languages, where there are not types and there is no need of converting the type to another one, in C++ there is such a need, which can be achieved by using different types of casting, the language provides. The advantage of it is that we can always be sure we are using a proper type.

Although C++ is an extension of C, it can be faster than it. That's because of tight data structures, which when cached are much faster and more scalable. It also provides us with template metaprogramming, which allows to calculate values of functions in compilation time instead of execution time (from C++11 standard we can use constexpr to achieve the same). The use of pointers is also why the language is very fast. That's because pointers are very close to the machine and performing operations on them is the same as operating on the platform's memory. However, using pointers may have its disadvantages due to ease of making a mistake, which can cause memory leaks and segmentation faults. As of C++11 new smart pointers were introduced that help with the problem by automatically allocating and deallocating memory when needed.

The main part of C++ are classes, which were the main reason why the C++ language was developed, hence the first name of C++ was “C with classes”. The classes made programming easier to understand, use and more efficient, because of the possibility of representing the idea of program in them. With classes come inheritance and polymorphism, which is also a feature that this programming language developed.

3.1.2. Development Environment – Visual Studio 2017[20]

Visual Studio 2017 is an integrated development environment that was developed by Microsoft. It can be used to developed many different applications and programs for Windows, but also for different platforms, as well as web sites, web applications, services and mobile applications, databases and games. Visual Studio supports many programming languages, but it’s mostly used for C languages, due to the support that Microsoft provides.

Visual Studio has many different, useful features that are very helpful when it comes to programming. It allows us to develop, debug and test applications, but also to collaborate with other members from team, due to Git support and can be fully customizable with the use of different plugins and addons. This development environment also supports game development, but only for chosen game engines and technologies. The support includes DirectX, Unity, Unreal and Cocos, but that doesn’t mean that other engines can’t be integrated with it.

When it comes to programming, Visual Studio uses IntelliSense, which provides a very powerful assistance when writing a code. IntelliSense is an intelligent code completion tool that processes the existing code and, when writing a code produces auto completion suggestions. It also helps the developer with reducing typos and other common mistakes, which speeds the programming process drastically. Another feature of Visual studio is that it helps us to navigate in projects that have a lot code in them. It provides us with many different types of searching, filters and other functions, like finding references of functions, easy switching between header and source files. The next aspect of editing in Visual Studio is that it shows us the list of each error there is in a code without the need of compilation, which eliminates the need of recompilation to see if there are any errors. Visual Studio gives us also the possibility of refactoring the code, providing quick actions that allow us to rename all instances of variables and methods.

Debugging in Visual Studio is easy to perform due to many different components built in the environment. It’s possible to debug code written in different programming languages supported by Visual Studio, even if the developed application uses more than one language at a time. Also, Visual Studio’s debugger allows for performing debugging not only on the local platform, but also on another platforms, servers, or emulators, where the application is running. The debugger displays many kinds of information for the developer, like values of the variables, exceptions and their location in code when something goes wrong. The biggest advantage over another debugger is that Visual Studio’s debugger allows to go deeper into debugging than only looking at the code. It allows us to go closer to the hardware level, where we can see values of the memory, registers and disassembly window, which can help the developer to find a very hard to locate bug. Another advantage is that it allows us to debug multi-threaded programs, by showing a graphical representation of each thread which shows what is being performed by that thread and what are the values within it.

Visual Studio is also very useful tool when it comes to testing an application. It supports unit testing and has a tool called IntelliTest, which helps to generate unit tests by automatically testing the functions with different values. The generated tests can then be exported as a unit tests. Visual Studio's testing feature allows us to test the application live, which displays which line of code is passing unit tests and which is failing, allowing to write a properly working functions without the need of running tests multiple times.

3.1.3. Version Control – GIT[21]

Version control is very useful when it comes to development of any kind of application, including games. The version control is used as a file storage as well as a tracking tool for changes made in those files. It allows also for merging of changes without breaking the code, so the developers may work together.

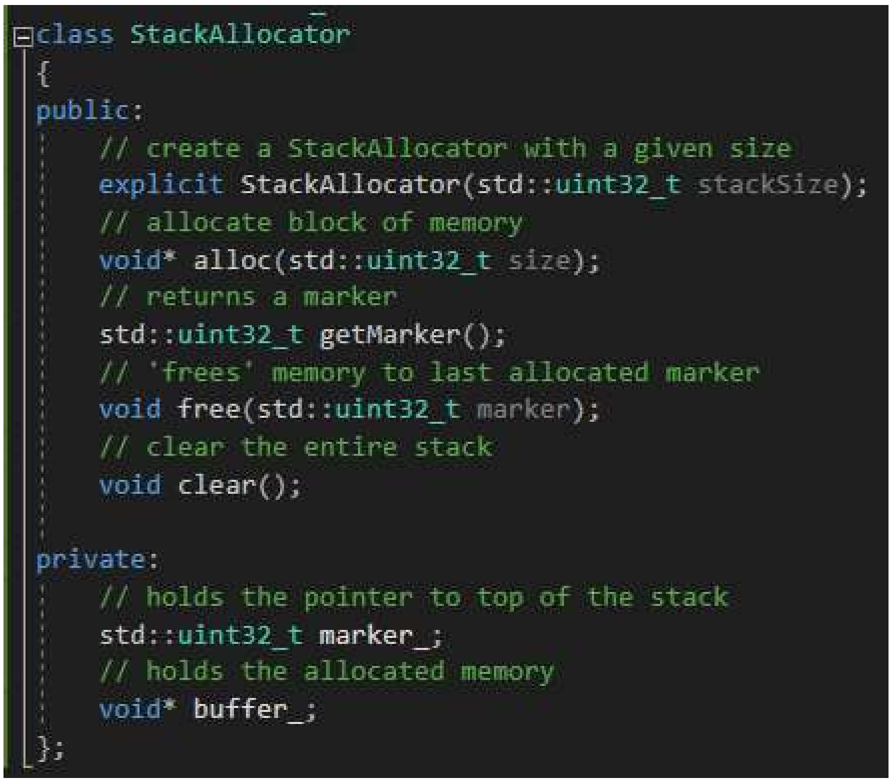
In the project, the version control stores source code and assets of a project and helps to track any changes made in a project, which can be later used for bug fixing purposes. Thanks to tracked changes, the developer can easily locate a date when the problem started and use it to check which parts of code might have broken the application.

Git is a free, popular tool that implements the features of version control. It allows for easy storage of changes made in implemented system and for simple reverting of modifications without the need of removing the code manually. Git stores the files in local repositories, which are placed on developer's computer, but allows for synchronization of data with remote servers (an example of such servers can be GitHub or BitBucket), which allows multiple members of a team to work on one project at the same time. Due to Git being integrated inside of Unreal Engine's Editor and Visual Studio, which provide various built in options to use Git without the need to call command from the command prompt, the version control in the project is much easier to handle.

3.2. Engine support systems

3.2.1. Stack allocator[figure 3.1.]

Stack allocator is implemented by allocation of a big chunk of memory with the use of *new* function. The StackAllocator class holds a pointer to a marker indicating the top of the stack of memory. All memory addresses above the marker are free to use and the memory under the marker is used. When the memory is allocated, the pointer is moved to the end address of the allocated memory. When the memory is freed, the stack pointer moves to the beginning of last allocated memory block.



```
class StackAllocator
{
public:
    // create a StackAllocator with a given size
    explicit StackAllocator(std::uint32_t stackSize);
    // allocate block of memory
    void* alloc(std::uint32_t size);
    // returns a marker
    std::uint32_t getMarker();
    // 'frees' memory to last allocated marker
    void free(std::uint32_t marker);
    // clear the entire stack
    void clear();

private:
    // holds the pointer to top of the stack
    std::uint32_t marker_;
    // holds the allocated memory
    void* buffer_;
};
```

Figure 3.1. Declaration of StackAllocator

3.2.2. Pool allocator

Pool allocator, similarly to stack allocator, is also initialized by allocating a huge amount of memory. Free memory addresses are stored inside a linked list and during allocation, the next free address is removed from the list and returned. When the allocated element is freed, its address is added back to the linked list of free addresses and is ready for allocation for another object. This allows to achieve the complexity of O(1) for both allocation and deallocation of a memory.

3.2.3. Memory defragmentation

As the game engine is using mostly stack and pool allocators there's no need to introduce a memory defragmentation.

The stack allocator frees the memory in reverse order of allocation, which means that in order to free a memory of an object in the middle of the allocated memory, all the above memory addresses also must be freed, which makes it impossible for memory fragmentation to occur.

The pool allocator stores elements of a fixed size, which makes it possible to always allocate a memory space without worrying on fragmenting the memory.

3.2.4. Engine configuration

The runtime engine configuration is stored in a singleton class' (*GameEngineConfiguration*) member variables, which allows for easier access to the configuration and helps debugging the game with different parameter values.

The engine configuration is stored in a file called settings.ini, which is used to initialize variables of *GameEngineConfiguration* class on its creation.

3.3. Resources and file system

3.3.1. File system

Basic file types like .log, .txt and other similar files are loaded using C standard library and Microsoft's low-level IO APIs. The file is buffered using the API's functions[table 3.1.], like *_open()*, *_read()* and stored as a stream of bytes.

Table 3.1. Functions used for operations on files

Operation	**C standard library**	**Low-level I/O**
Open a file	fopen()	_open()
Close a file	fclose()	_close()
Read from a file	fread()	_read()
Write to a file	fwrite()	_write()
Seek to an offset	fseek()	_lseek()
Return current offset	ftell()	_tell()
Read a single line	fgets()	n/a
Write a single line	fputs()	n/a
Read formatted string	fscanf()	n/a
Write formatted string	fprintf()	n/a
Query file status	fstat()	n/a

3.3.2. Resource manager

The *ResourceManager* class provides a wrapper, which uses file system functions in order to provide functions responsible for performing operations on different types of files used in the game engine, like textures, shaders, models, etc.

3.3.2.1. Revision control for resources

Resources are stored inside a git repository on GitHub together with the code, which allows for easy and fast project replication on different devices and keeping the project up-to-date without the need to store it on external storage devices.

3.3.2.2. Resource database

The files are stored in the file system divided into directories. Each directory contains a different type of file or a group of similar files.

Directory hierarchy:

- Game engine – root directory
    - Resources – root directory for all resources
        - Models – models of characters
        - Shaders - shaders
        - Textures – textures of objects
    - settings.ini – configuration file

3.3.2.3. Asset conditioning pipeline

The asset conditioning pipeline is built in the *ResourceManager*. When loading a file, specific features of it are extracted.

Asset conditioning pipeline for:

- .m3d files (model files):
  - read materials
  - read subset table
  - read skinned vertices
  - read triangles
  - read bone offsets
  - read bone hierarchy
  - read animation clips
- .dds files (texture files):
  - convert data to DXGI format
  - load:
    - dimension
    - alignment
    - width
    - height
    - depth
    - mip levels
    - format
    - layout
  - create texture

3.3.3. Runtime resource management

3.3.3.1. File formats

Following file formats were used in the game engine:

- .m3d file – contains data about models
- .dds file – contains data about texture
- .hlsl file – contains data about shaders
- .ini file – contains game engine configuration

3.3.3.2. GUID

GUIDs of resources are stored as a *string* and are represented by the resource's file system path, which guarantees that the file's GUID will be unique.

3.3.3.3. Resource lifetime

In the game engine, resources are loaded right before they are used and removed when they are no longer needed. It was achieved by the usage of shared pointers together with constructors and destructors of the object. If all the references to a resource are freed, then the resource itself is destroyed.

3.4. Game loop

3.4.1. Rendering loop

As previously mentioned, the rendering loop consists of the following steps:

- update the camera
- update scene elements
- render scene
- swap rendering buffers

However, to avoid creation of multiple loops and save execution time, the rendering loop was integrated into a game loop.

3.4.2. Single threaded game loop[figure 3.2.]

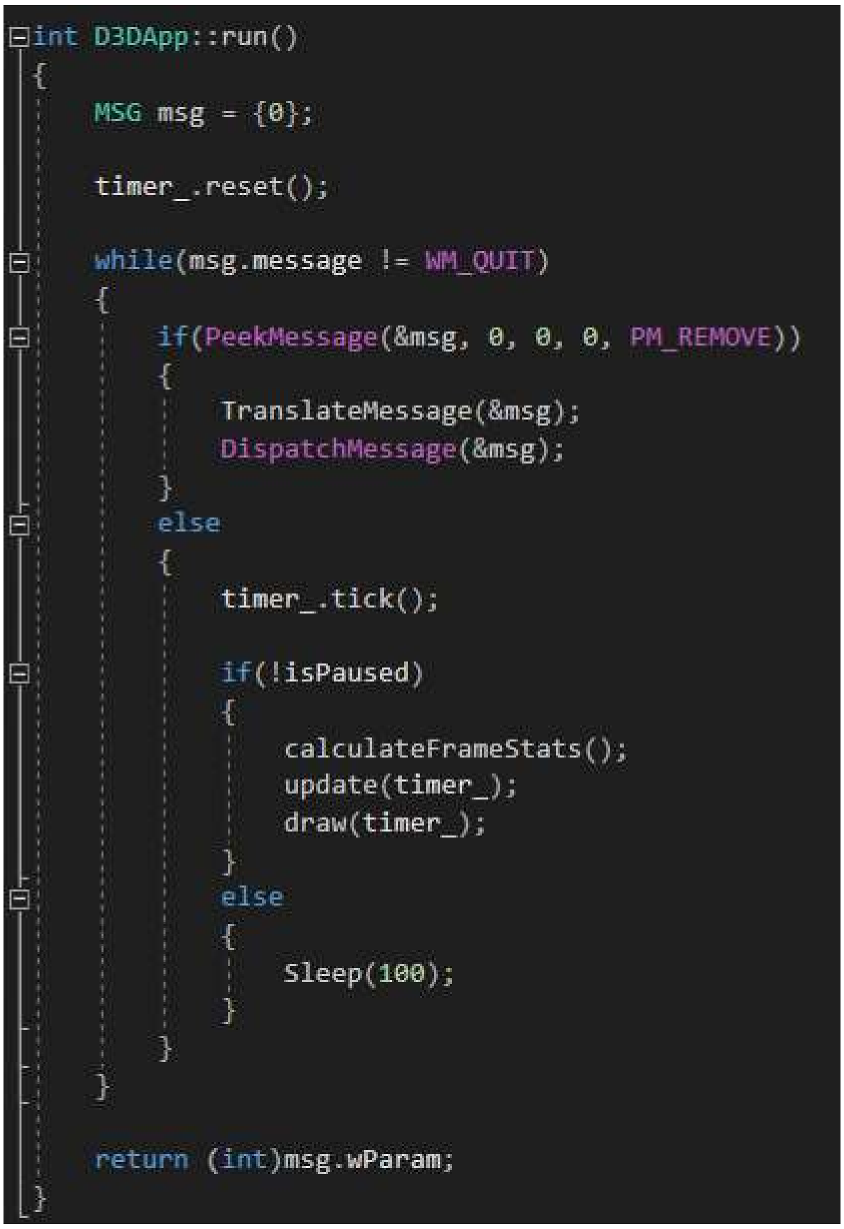
Steps of a game loop[22]:

- handle input
- update camera
- check if the GPU finished processing commands
- wait for GPU if not finished
- animate lights
- animate materials
- update object buffer
- update skinned buffer
- update material buffer
- update main pass buffer
- update shadow transform
- synchronize positions of objects with physics engine
- simulate physics forces
- step physics engine
- render graphics

3.4.2.1. Windows message pumps

The windows message pump uses Windows API to peek the messages submitted to Windows OS by human interface devices. The messages are then translated into character messages and dispatched to a window procedure. Such a modified message can be used by the game engine.

The input handling is the first step of the game loop and continues execution until all messages are handled.



```
int D3DApp::run()
{
    MSG msg = {0};

    timer_.reset();

    while(msg.message != WM_QUIT)
    {
        if(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            timer_.tick();

            if(!isPaused)
            {
                calculateFrameStats();
                update(timer_);
                draw(timer_);
            }
            else
            {
                Sleep(100);
            }
        }
    }

    return (int)msg.wParam;
}
```

Figure 3.2. Game loop implementation

3.4.2.2. Callbacks

Callbacks are implemented by creating std::function objects bound (std::bind) with a class function. Such an object is then passed to another class or function. The class has then a possibility of executing a callback, which will result in a call to the bound function.

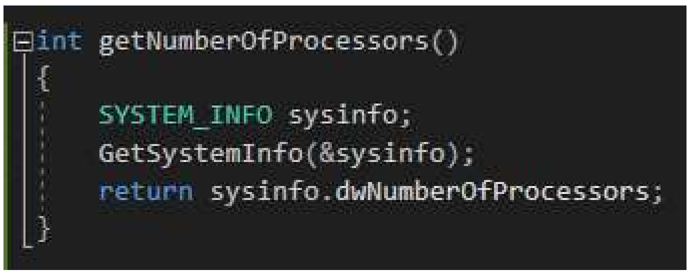
3.4.2.3. Events

Events work very similar to callback, but instead of keeping only one function object of one class, multiple functions (called delegates) are stored. When an event occurs, for example one tick of the game time's passes, the class responsible for an event calls all delegates that subscribed for the event.

3.4.3. Multithreaded game loop

3.4.3.1. Each subsystem on a separate thread

Each of the bigger subsystem's functions are ran on a separate thread with the use of *CreateThread* function (located in processthreadsapi.h header). Additionally the game engine fetches a number of available cores/processors with the use of *GetSystemInfo()* function[figure 3.3.]. After the number of cores is determined, the main thread of the game engine tries to set the affinity (*SetThreadAffinityMask()* function) of a thread to run on a separate core in order to avoid time-slicing for threads running on the same core. The cores are assigned in order from 0 to *numberOfProcessors*, where the number of currently used cores is incremented with *InterlockedIncrement()* function to allow all threads to use actual value of the variable, avoiding data races.



```
int getNumberOfProcessors()
{
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    return sysinfo.dwNumberOfProcessors;
}
```

Figure 3.3. Definition of a function returning number of processors/cores

3.4.3.2. Scatter/gather[23]

The data is divided into equally sized arrays and operations on the arrays are performed on separate threads, ran similarly to threads used for running subsystems on separate threads.

Before the data is divided and threads are executed, an output array is preallocated on a main thread, which allows to avoid allocations inside the threads. The threads can safely save the output data into the array without the need of synchronization between the threads, as each thread calculates an index[formula 3.1., formula 3.2] that is guaranteed to be free for use only for the thread.

$$startIndex = \frac{numberOfElements}{numberOfThread} * coreId \tag{3.1.}$$

$$endIndex = \frac{numberOfElements}{numberOfThreads} * (coreId + 1) \tag{3.2.}$$

3.4.3.3. Job system

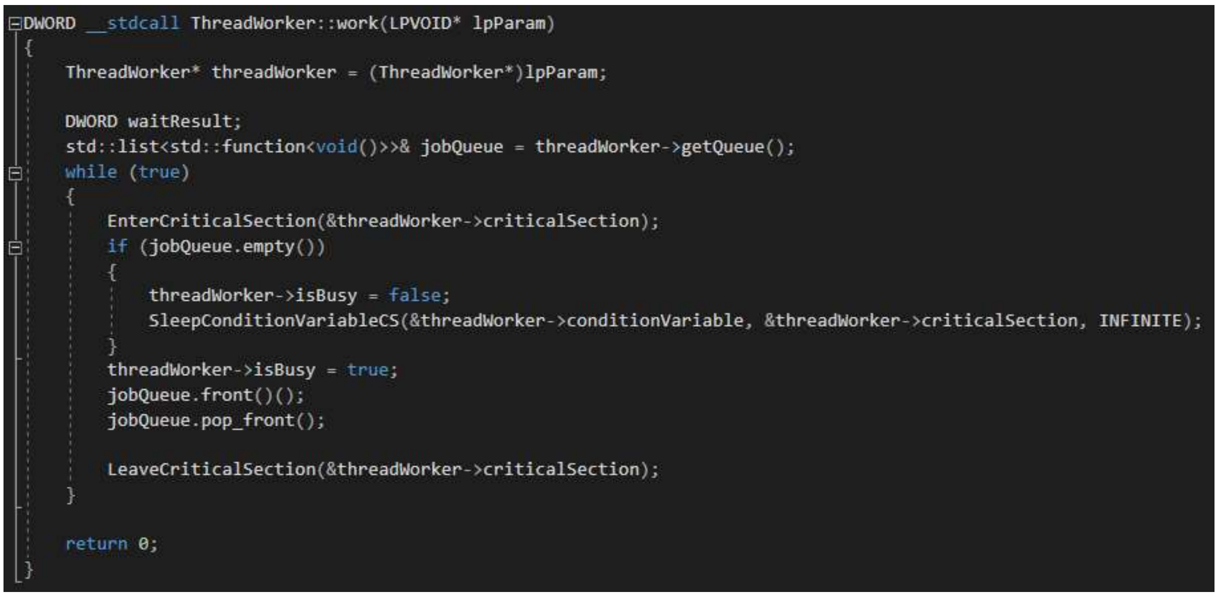
The job system consists of a thread pool and thread workers. A thread pool, also known as an executor, is a class that stores thread workers and manages them by allocating jobs and keeping track of execution times and problems that may appear. A thread worker is a class containing a queue of jobs, which are represented by a function object that is executed in the thread[figure 3.4.].

Each worker thread is located on one of the computer's cores and runs infinite loop that is designed to process all submitted jobs to the worker's job queue, which are added by other threads or jobs. As a first step in the infinite loop, the worker thread goes to sleep using a condition variable and waits for jobs to be added to the job queue. When a job is added, the thread is woken up and calls the job's function. When the job is executed and the function returns, the thread worker goes back to the beginning of the loop and checks if there are more jobs to be executes. If there's no more jobs, the thread goes back to sleep and waits for another job to be added to the queue.

Job workers use also a critical section mechanism, which indicates that the worker is currently using the job queue and function adding the job should wait for the worker to leave the critical section before a job can be added. This allows to avoid referencing of the queue variable by two threads simultaneously, which removes a possibility of crash/data races.

3.4.3.4. Dynamic job system

Dynamic job system is an extension of a job system. It monitors execution time of each of the game engine's subsystems and based on the times, allocates a different number of thread workers to each of the subsystems. The time is gathered from the beginning of inserting the jobs to the thread worker, till all thread workers executing the specific job finish executing the jobs. The amount of thread workers allocated to a subsystem is calculated as a fraction of time spent by subsystem per overall time of execution of every job in a frame.



```
DWORD __stdcall ThreadWorker::work(LPVOID* lpParam)
{
    ThreadWorker* threadWorker = (ThreadWorker*)lpParam;

    DWORD waitResult;
    std::list<std::function<void()>>& jobQueue = threadWorker->getQueue();
    while (true)
    {
        EnterCriticalSection(&threadWorker->criticalSection);
        if (jobQueue.empty())
        {
            threadWorker->isBusy = false;
            SleepConditionVariableCS(&threadWorker->conditionVariable, &threadWorker->criticalSection, INFINITE);
        }
        threadWorker->isBusy = true;
        jobQueue.front()();
        jobQueue.pop_front();

        LeaveCriticalSection(&threadWorker->criticalSection);
    }

    return 0;
}
```

Figure 3.4. Definition of worker's main function

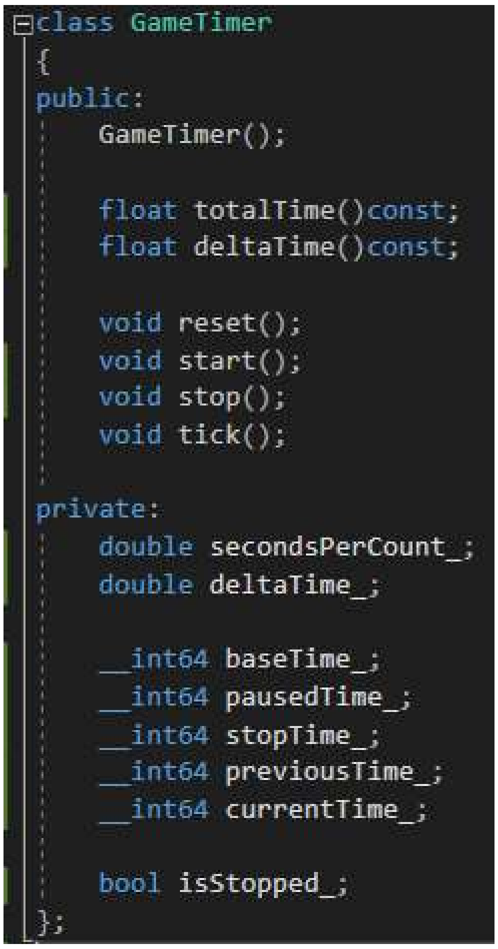
3.4.4. Time

3.4.4.1. Real time

The real time is obtained using *std::chrono::system_clock::now()* function.

3.4.4.2. Game time[figure 3.5.]

The game time is stored in a GameTime class and is based on a value of the performance counter. The time is stored in seconds, which is calculated by taking a frequency of the performance counter (*QueryPerformanceFrequency* function) and dividing the value of the counter (*QueryPerformanceCounter* function) by it.



```
class GameTimer
{
public:
    GameTimer();

    float totalTime()const;
    float deltaTime()const;

    void reset();
    void start();
    void stop();
    void tick();

private:
    double secondsPerCount_;
    double deltaTime_;

    __int64 baseTime_;
    __int64 pausedTime_;
    __int64 stopTime_;
    __int64 previousTime_;
    __int64 currentTime_;

    bool isStopped_;
};
```

Figure 3.5. Game timer declaration

3.4.4.3. Frame rate

The frame rate is calculated by checking how many frames were rendered in a time span of a one second.

3.4.4.4. Delta time[formula 3.3.]

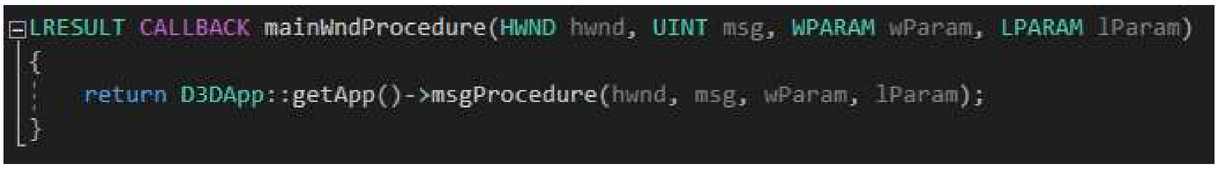
The delta time is calculated by subtracting a time of previous tick from a current time and multiplying the result by the number of seconds required for one performance counter's tick.

$$\Delta t = (currentTime - previousTime) * secondsPerCount \qquad (3.3.)$$

3.5. Human interface devices

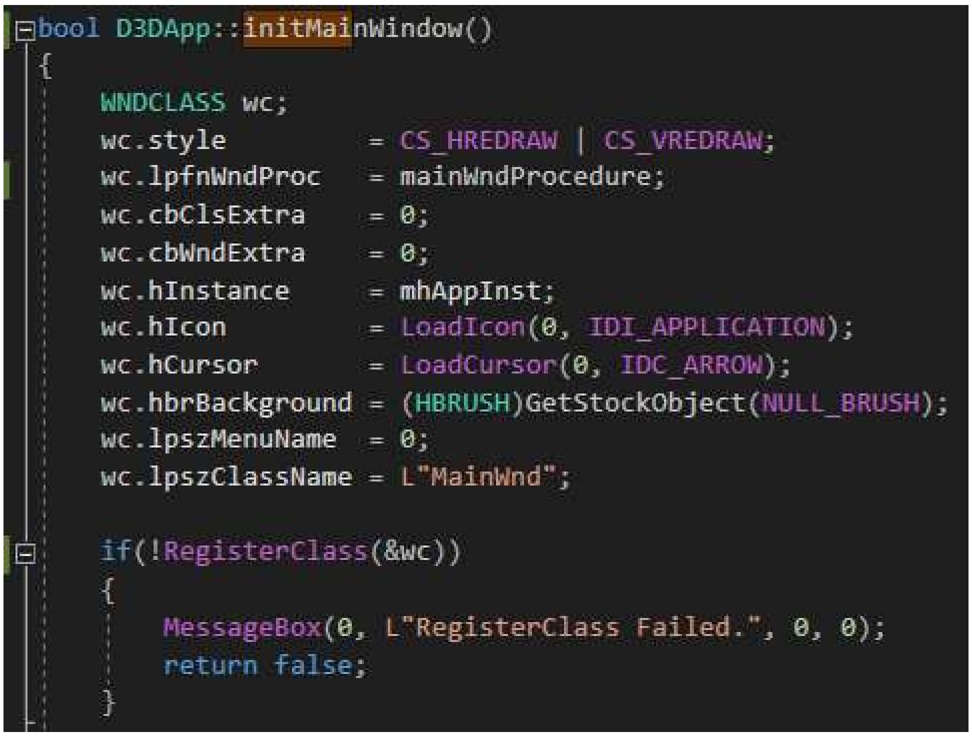
To handle human interface devices, first a window procedure[figure 3.6.] must be defined. The procedure is defined together with an initialization of a game engine's main window[figure 3.7.].

All window related messages are captured using window's message procedure[figure 3.8.] and, if needed, using *PeekMessage* function. The messages are then translated and dispatched to the game engine's input handler. The procedure uses a callback function that is designed to handle all kinds of messages needed by the engine. This includes window activation message (*WM_ACTIVATE*), window resize message (*WM_SIZE*), window move message (*WM_MOVE*), but most importantly keyboard and mouse messages. When a message is successfully recognized, a corresponding handler function is called with the message's parameters.



```
LRESULT CALLBACK mainWndProcedure(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    return D3DApp::getApp()->msgProcedure(hwnd, msg, wParam, lParam);
}
```

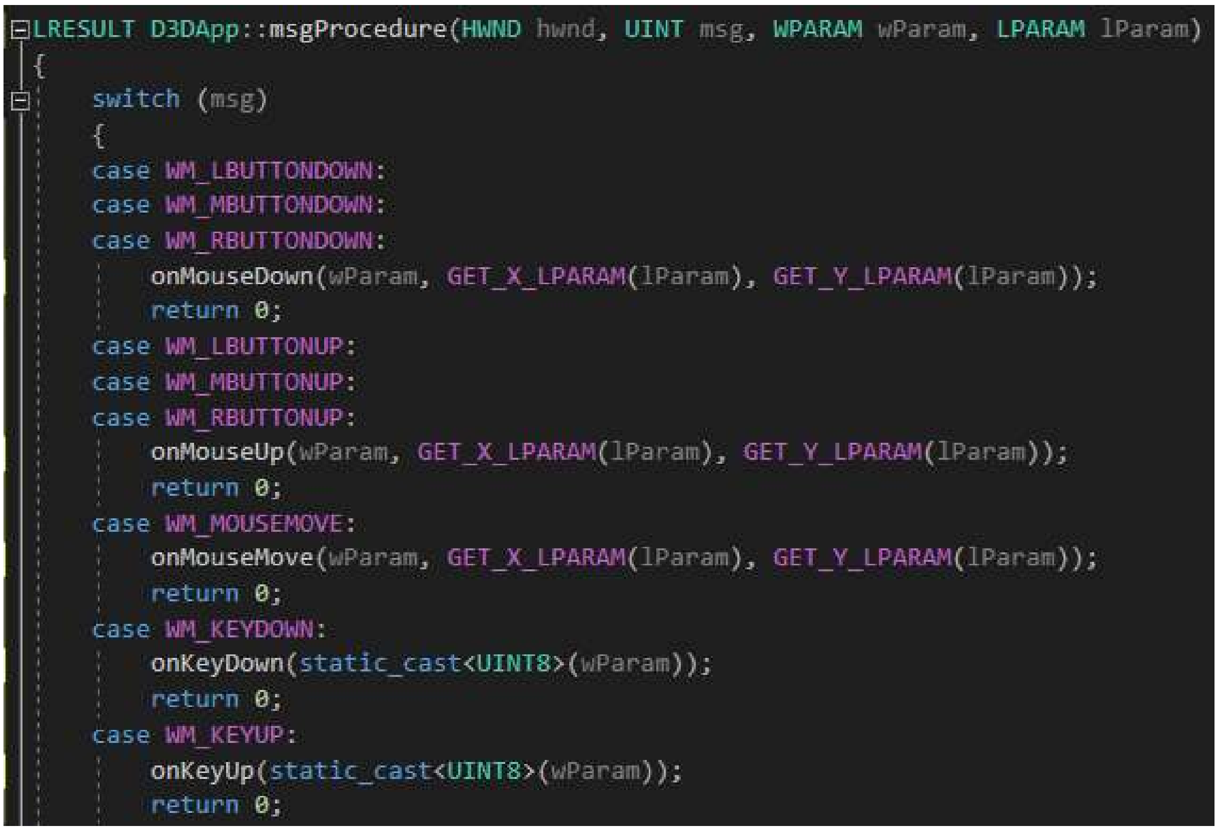
Figure 3.6. Definition of main window procedure callback



```
bool D3DApp::initMainWindow()
{
    WNDCLASS wc;
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = mainWndProcedure;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = mhAppInst;
    wc.hIcon         = LoadIcon(0, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(0, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(NULL_BRUSH);
    wc.lpszMenuName  = 0;
    wc.lpszClassName = L"MainWnd";

    if(!RegisterClass(&wc))
    {
        MessageBox(0, L"RegisterClass Failed.", 0, 0);
        return false;
    }
```

Figure 3.7. Creation of game engine's window



```
LRESULT D3DApp::msgProcedure(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_LBUTTONDOWN:
    case WM_MBUTTONDOWN:
    case WM_RBUTTONDOWN:
        onMouseDown(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
    case WM_LBUTTONUP:
    case WM_MBUTTONUP:
    case WM_RBUTTONUP:
        onMouseUp(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
    case WM_MOUSEMOVE:
        onMouseMove(wParam, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
        return 0;
    case WM_KEYDOWN:
        onKeyDown(static_cast<UINT8>(wParam));
        return 0;
    case WM_KEYUP:
        onKeyUp(static_cast<UINT8>(wParam));
        return 0;
```

Figure 3.8. Definition of input handling function

3.6. Debugging

3.6.1. Logging and tracing

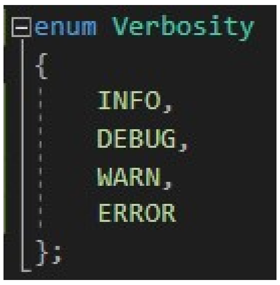
3.6.1.1. OutputDebugString

The *OutputDebugString* is called in order to print logs to a console.

3.6.1.2. Verbosity

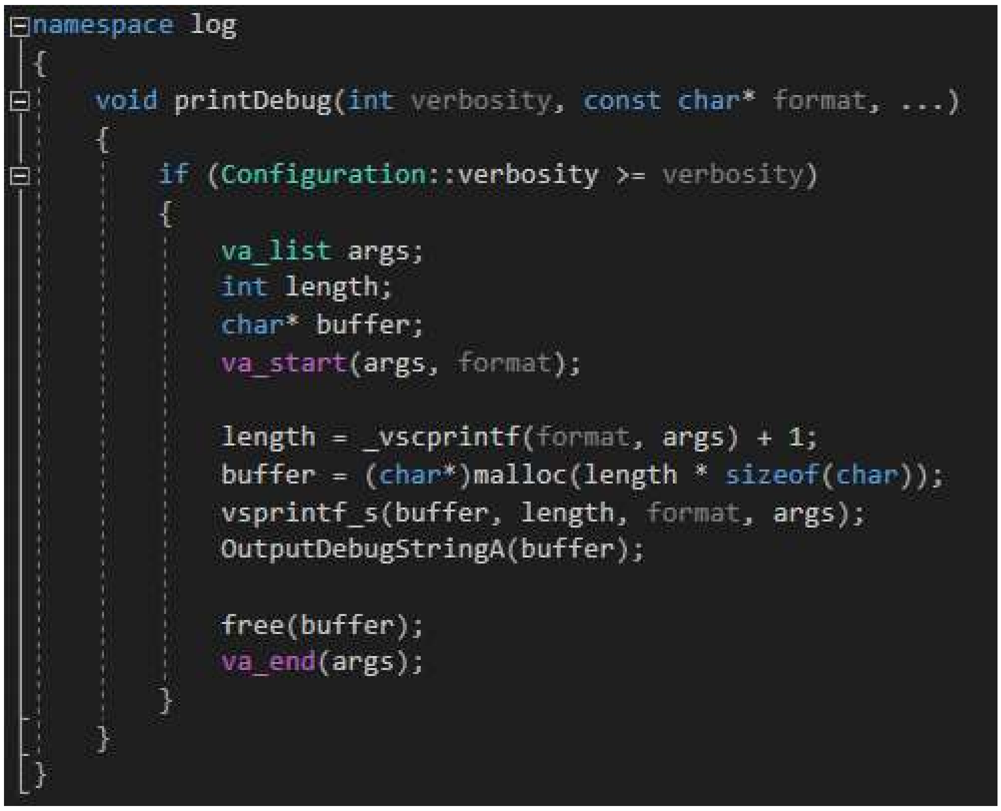
The verbosity is declared as an enumeration type[figure 3.9.]. It is used as a parameter to logging functions to provide an additional information of log type in resulting logs. The debug logs are only displayed if the executed game engine was compiled with a _DEBUG_ flag (checked with #ifdef).

The verbosity level is defined in the game engine’s configuration. When calling a debug printing function[figure 3.10.], if the verbosity level is high enough, then all messages having the verbosity lower than a verbosity defines in configuration will be printed. This was achieved with a simple *if* function.



```
enum Verbosity
{
    INFO,
    DEBUG,
    WARN,
    ERROR
};
```

Figure 3.9. Verbosity declaration



```
namespace log
{
    void printDebug(int verbosity, const char* format, ...)
    {
        if (Configuration::verbosity >= verbosity)
        {
            va_list args;
            int length;
            char* buffer;
            va_start(args, format);

            length = _vscprintf(format, args) + 1;
            buffer = (char*)malloc(length * sizeof(char));
            vsprintf_s(buffer, length, format, args);
            OutputDebugStringA(buffer);

            free(buffer);
            va_end(args);
        }
    }
}
```

Figure 3.10. Definition of logging function

3.6.1.3. Channels

In the game engine, the channel type is represented by an enumeration type with values corresponding to different types of subsystems.

When printing a message with a channel type as a parameter, a name of the channel is appended to the message, which later can be used in a log parsing tool to filter out for better readability.

3.6.1.4. Output logs to files

Logs are saved to files using an std::ofstream stream that allows to open a file and stream data to it, saving it.

3.6.2. Debug drawing

There are two different debug drawing tools implemented in the game engine:

- PhysX Visual Debugger (PVD)
- Game engine’s debugger

PhysX Visual Debugger provides a graphical user interface for visualizing and debugging of physics created with the PhysX SDK. It doesn’t require any code changes in order to work. The initialization of PVD is later described in chapter 3.8.2.

The game engine provides different functions allowing for debug drawing during the gameplay:

- drawing object wireframe
- drawing shadow map[figure 3.11.]
- drawing text
- drawing lines

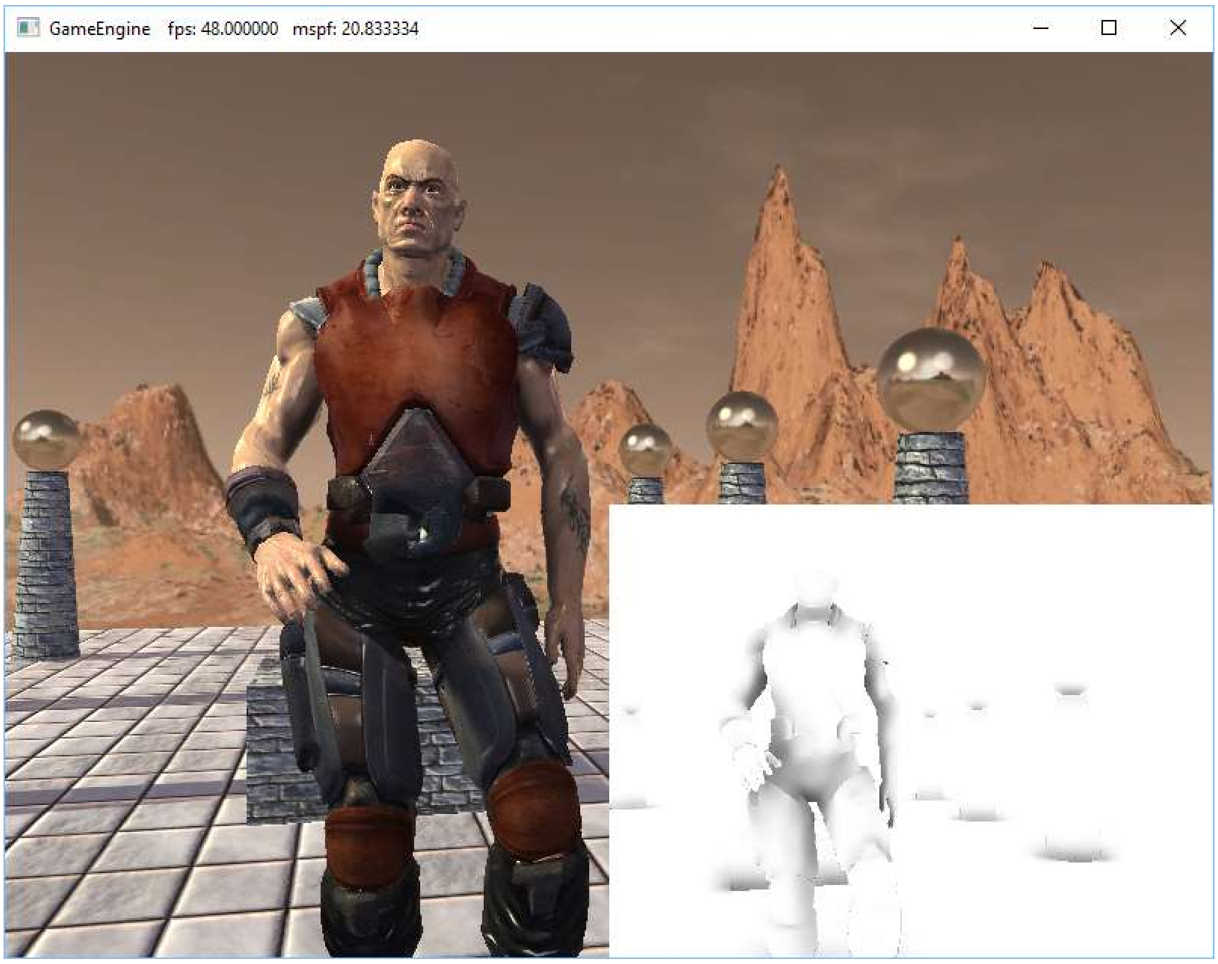


Figure 3.11. Shadow map (in the corner)

3.7. Rendering engine

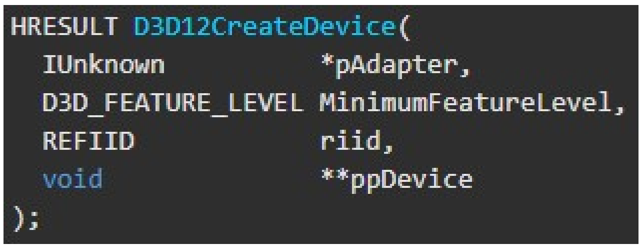
3.7.1. DirectX initialization[24]

DirectX initialization is a long process and consists of the following steps:

- creation of *ID3D12Device*[figure 3.12.]
- creation of *ID3D12Fence* object[figure 3.13.]
- query of descriptor sizes
- check of 4X MSAA quality level support
- creation of command queue, command list allocator and main command list
- creation and description of a swap chain
- creation of descriptor heaps
- resize of back buffer
- creation of render target view to the back buffer

- creation of depth/stencil buffer and view
- setting the viewport

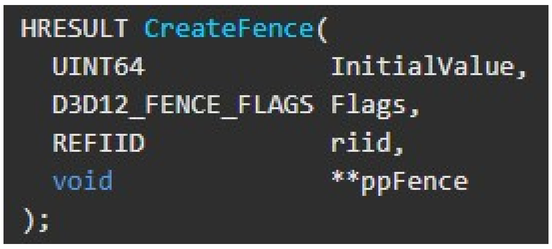
The *ID3D12Device* represents a display adapter. The device is needed in order to check if the desirable DirectX version is supported. It also provides functions allowing for creation of other DirectX objects.



```
HRESULT D3D12CreateDevice(
  IUnknown          *pAdapter,
  D3D_FEATURE_LEVEL MinimumFeatureLevel,
  REFIID            riid,
  void              **ppDevice
);
```

Figure 3.12. D3D12CreateDevice function declaration

The *ID3D12Fence* object is used for synchronization between CPU and GPU. It is created with a function provided by the device (*CreateFence*).



```
HRESULT CreateFence(
  UINT64            InitialValue,
  D3D12_FENCE_FLAGS Flags,
  REFIID            riid,
  void              **ppFence
);
```

Figure 3.13. CreateFence function declaration

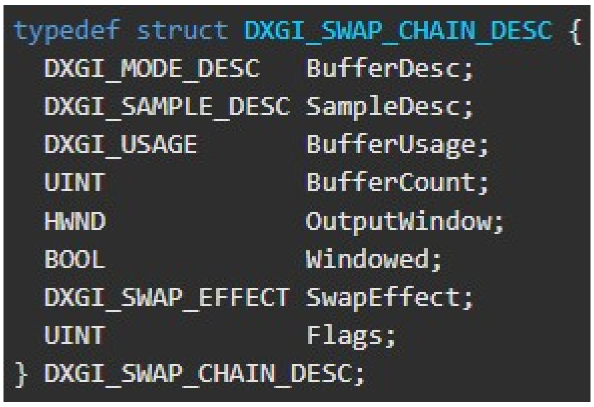
Descriptors are used to describe a resource to a GPU. As descriptor size can be different based on the used GPU, there's a need to first extract the size. In order to get that information, a *GetDescriptorHandleIncrementSize* function is used for each type of needed descriptor (provided as a function parameter).

The quality support is checked using a *CheckFeatureSupport* function and providing parameters used for the sampling, like sample count and the flag describing what type of sampling is used (4X MSAA).

To create a command queue and command list, there's a need to initialize following interfaces:

- *ID3D12CommandQueue*
- *ID3D12CommandAllocator*
- *ID3D12GraphicsCommandList*

A swap chain is initialized by filling an instance of DXGI_SWAP_CHAIN_DESC[figure 3.14.] structure. The structure defines the parameters that are to be created for the swap chain.



```
typedef struct DXGI_SWAP_CHAIN_DESC {
  DXGI_MODE_DESC   BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE       BufferUsage;
  UINT             BufferCount;
  HWND             OutputWindow;
  BOOL             Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT             Flags;
} DXGI_SWAP_CHAIN_DESC;
```

Figure 3.14. DXGI_SWAP_CHAIN_DESC declaration

Descriptor heaps are needed in order to store the needed descriptors and views. A single heap is created with a use of *CreateDescriptorHeap* function.

Depth and stencil buffers are used to determine the depth of visible objects and to provide stencil parameters used for stenciling. The buffers are represented as a 2D texture, which is described by a *D3D12_RESOURCE_DESC* structure and created with the use of *CreateComittedResource* method.

The viewport is a back buffer used for storing information of what should be rendered on the screen. It is described by a *D3D12_VIEWPORT* structure.

3.7.2. Scene description

3.7.2.1. Triangle mesh

The triangle mesh is built with triangle plates. Each of the triangles is defined by specifying all three of its vertices.

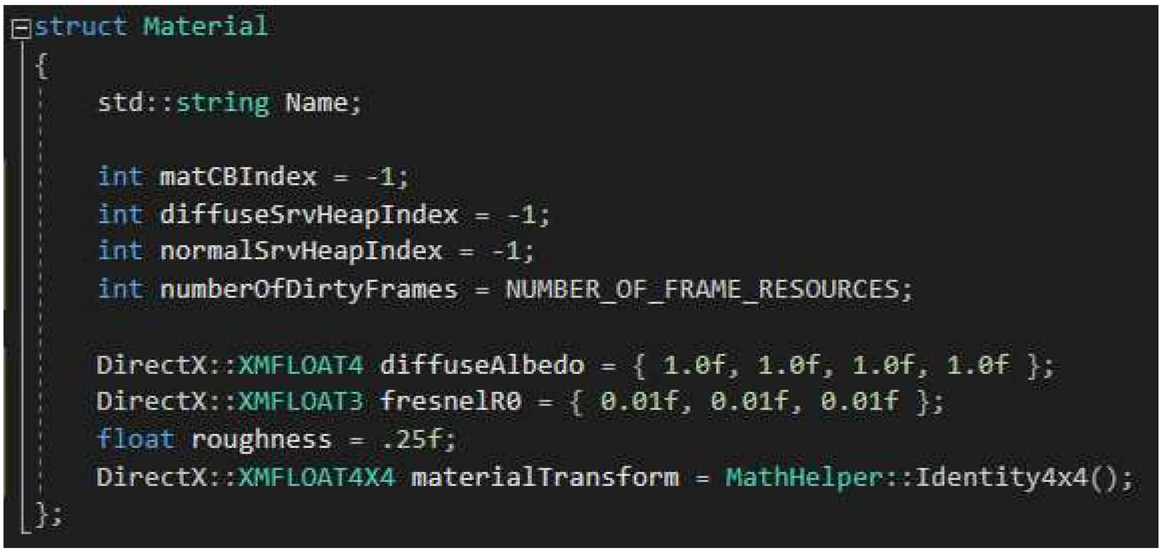
3.7.2.2. Textures

Textures are stored in a DirectDraw Surface file format, which is compressed using a S3 Texture Compression algorithm. The compressed texture can be later decompressed by the GPU.

Textures are loaded with the use of Microsoft's DDSTextureLoader, which is a group of helper function allowing for easier loading of .dds files.

3.7.2.3. Materials

Materials are implemented as a structure[figure 3.15.] containing all information needed by the GPU to process it.



```
struct Material
{
    std::string Name;

    int matCBIndex = -1;
    int diffuseSrvHeapIndex = -1;
    int normalSrvHeapIndex = -1;
    int numberOfDirtyFrames = NUMBER_OF_FRAME_RESOURCES;

    DirectX::XMFLOAT4 diffuseAlbedo = { 1.0f, 1.0f, 1.0f, 1.0f };
    DirectX::XMFLOAT3 fresnelR0 = { 0.01f, 0.01f, 0.01f };
    float roughness = .25f;
    DirectX::XMFLOAT4X4 materialTransform = MathHelper::Identity4x4();
};
```

Figure 3.15. Material declaration

3.7.2.4. Shading

The shaders are defined inside .hlsl files and compiled using a *D3DCompileFromFile* method, which is wrapped inside a helper function to support error handling.

3.7.2.5. Lighting

Lighting is stored inside *PassConstants* structure, which describes constant variables in the game engine.

Ambient light is stored as a *XMFLOAT4* type, which is a 4D vector. The vector describes x, y, z and w coordinates.

All other lights are stored as a *Light* structure, which consists of strength, direction and position of the light as well as a spot power and falloff range.

3.7.2.6. Virtual camera

The player's camera is defined as a vector of three single-precision floating-point values, which represent the camera's position in the world.

The camera system uses a spherical coordinate system consisting of three values:

- radial distance (*r*) – describes a distance from the target, camera points at
- polar angle (*θ*) – describes vertical position of the camera relative to the target
- azimuthal angle (*φ*) – describes horizontal position of the camera relative to the target

The camera's position is calculated as indicated by the formulas 3.4., 3.5. and 3.6.

$$eyePosition.x = radius * \sin(\varphi) * \cos(\Theta) + \mathrm{lastPosition.x} \quad (3.4.)$$

$$eyePosition.z = radius * \sin(\varphi) * \sin(\Theta) + \mathrm{lastPosition.z} \quad (3.5.)$$

$$eyePosition.x = radius * \cos(\varphi) \quad (3.6.)$$

where lastPosition is a position of the target the camera is following.

3.8. Animation

3.8.1. Skeleton

The skeleton is stored as a set of skinned vertices, which are extracted from a .m3d file

3.8.2. Skinning

All skinned vertices are saved in the character model's file. To render a skinned skeleton, first all the needed data is loaded from the model file. The data is used to create an instance of a *SkinnedModel* as well as a *MeshGeometry* by initializing the structure's vertex and index buffers for both CPU and GPU. Later, submeshes are created that are used in creation of *RenderItem* that is later needed for object rendering.

3.8.3. Animation blending

The resulting position of a vertex of a skin can be influenced by the character's bones. The position is calculated as an average weight of the positions of the bones.

The character's skin is represented as a single mesh. Each vertex has a vector of four indices that point to the indices of influencing bones in a final transformation matrix. The final transformation matrix holds a final position for each bone. The vertex can use up to four bones to calculate its final position. Additionally, the vertex contains a vector of weights of each of the bones. If the weight of the bone is equal to zero, then the bone's influence is not considered.

3.9. Physics

3.9.1. PhysX initialization

To initialize PhysX, an instance of *PxFoundation* is needed to be created. The *PxFoundation* class is required to access different modules of PhysX SDK. To initialize the class, following parameters are required:

- version id – version of the foundation
- allocator callback – a function used for memory allocation
- error callback – a function called in case of an error occurring

The second step of initializing physics in the game engine is to create instance of *PxPhysics* class, which is responsible for handling of the physics used in the game engine and providing methods for physics manipulation. *PxPhysics* class is created with the use of *PxCreatePhysics* function, which takes physics version, a foundation, a tolerance scale, a flag telling if memory profiling should be enabled and an optional instance of PVD.

3.9.2. PhysX Visual Debugger (PVD)

To use PhysX Visual Debugger, an instance of a *PxPvd* needs to be created and connected with the debugger by using a socket connection. The socket connection allows to use the visual debugger remotely.

3.9.3. Collision detection[25]

To achieve collision detection in the game engine, a collision mesh needs to be created for each object. A collision mesh is a simplified object, which can be a sphere, box, capsule, etc., that is imposed on the object for which we want to simulate a collision.

As the physics engine runs separately of a rendering engine (there are separate instances of objects on the physics engine and on the rendering engine), there is a need to synchronize the objects in order to properly render them. To achieve that, the main thread creates synchronization jobs that take the positions of objects from the physics engine, translate it to DirectX's left-handed coordinate system (PhysX uses right-handed coordinate system) and set the position of all objects in a rendering engine.

3.9.4. Rigid body dynamics

PhysX provides many methods that can be used in order to push an object into motion. The game engine uses a wrapper that wraps these methods by extending the functionality of PhysX classes allowing to use the game engine's objects as parameters.

4. Experiments[26]

4.1. Load testing (performance)[27]

Load testing is performed to see what the limits of the system are and if the application is running correctly while the system is under load. The load testing was performed by generating a specific number of objects each frame and simulating physics for objects' generated shape and position.

The testing was performed on a version of the game engine, where all operations related to graphics card were extracted to reduce the number of external factors that can influence the results of the experiments.

The goal of these tests is to check how big is the improvement of using multithreading and to choose the best architecture.

As of the day of writing this thesis, the desired minimum FPS number among the gaming community is 60, the load testing is not performed for values giving results below the minimum 60 FPS.

The frame statistics were gathered every one second, which in some cases can create an impression of bigger FPS spikes. An average number of frames per second was introduces to make such cases more readable, making the tests more reliable.

4.1.1. Singlethreaded architecture

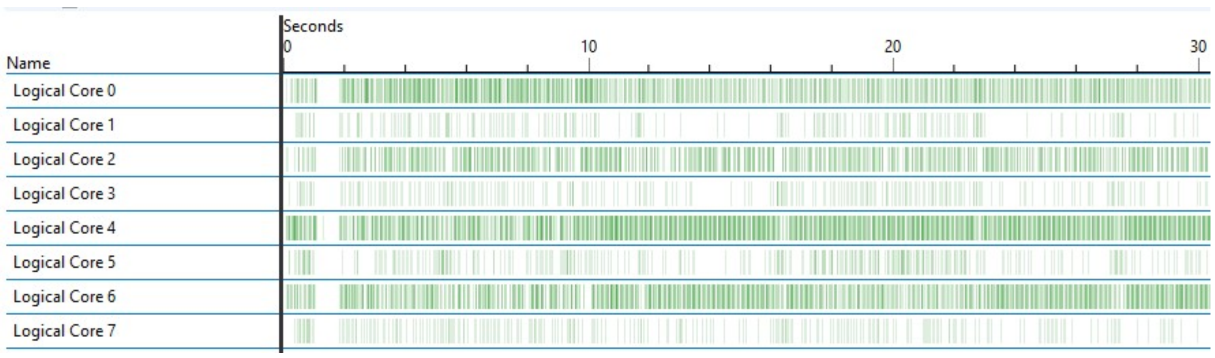


Figure 4.1. Load of each logical core for single threaded system

From the function load graph[figure 4.1.], it is seen that the game engine is executed on all the logical cores. That means that that there are many context switches between cores. The context switches are expensive for the processor as it requires to save and load all the registers and memory maps.

4.1.1.1. 10 objects

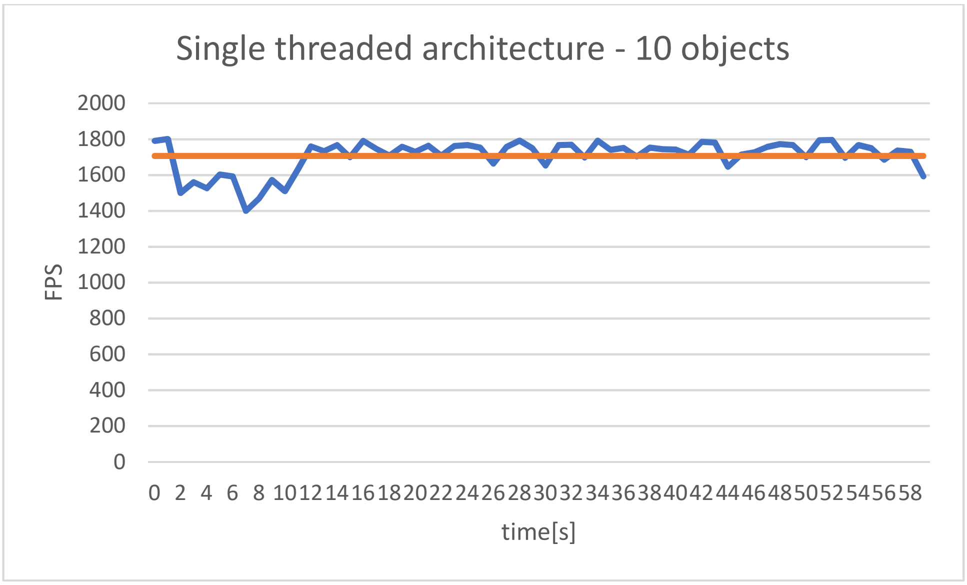


Figure 4.2. FPS number in time for 10 objects in single threaded architecture

Table 4.1. Thread statistics for single threaded architecture with 10 objects

Cross-core context switches	**Total context switches**	**Percent of context switches that cross cores**
72034	128346	56.12%
Execution time	**Synchronization time**	**Sleep**
56%	42%	2%

The average number of FPS for 10 constantly generated objects was 1706[figure 4.2.], which is a very good result compared to desired 60 FPS.

Looking at the amount of context switches that cross cores[table 4.1.], it is seen that there's a very huge possibility of improvement if those were eliminated.

4.1.1.2. 100 objects

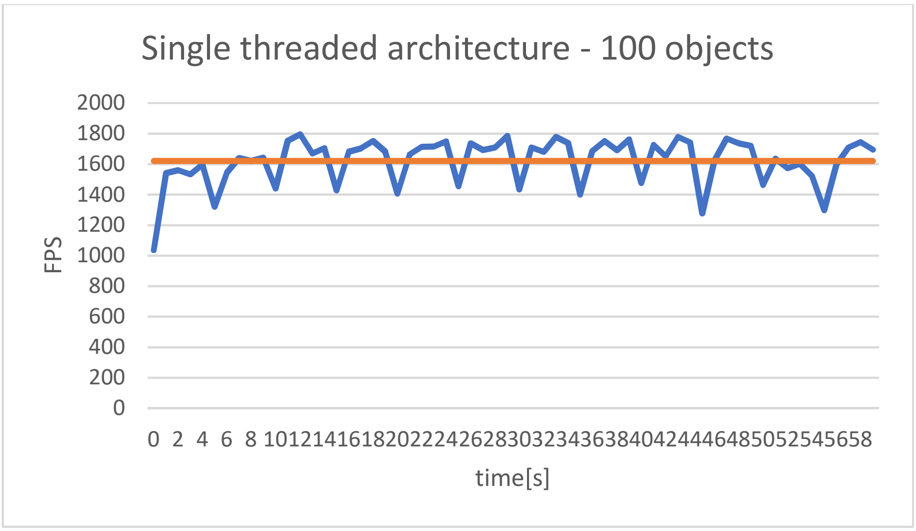


Figure 4.3. FPS number in time for 100 objects in single threaded architecture

Table 4.2. Thread statistics for single threaded architecture with 100 objects

Cross-core context switches	Total context switches	Percent of context switches that cross cores
64759	120988	53.53%
Execution time	**Synchronization time**	**Sleep**
56%	40%	4%

For 100 randomly generated objects, the average number of FPS was 1621[figure 4.3], which indicates roughly 1 FPS loss per object.

The percentage of context switches[table 4.2.] remained very similar to the previous experiment, as only one thread is being used, which was expected due, as all the cores are equally distributed for all the systems tasks.

4.1.1.3. 1000 objects

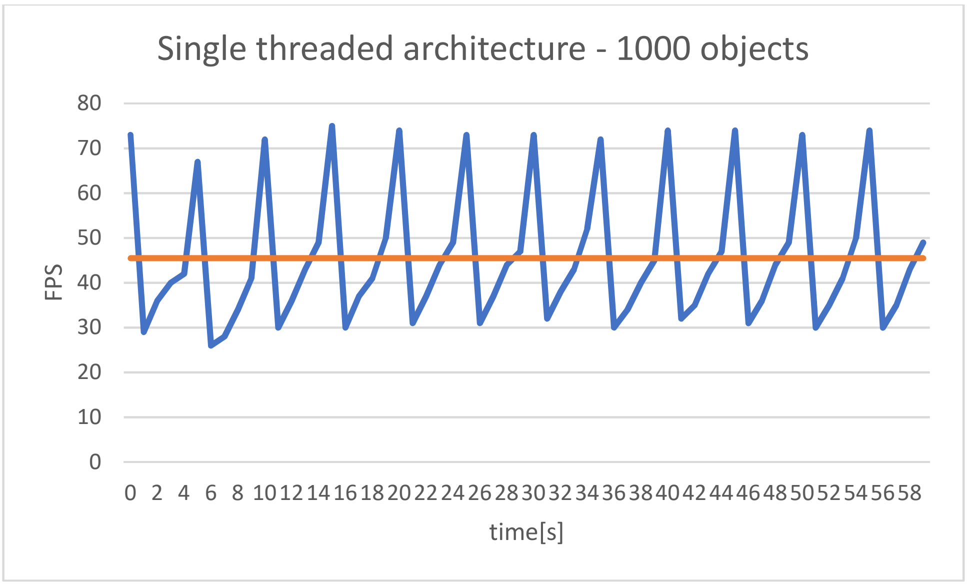


Figure 4.4. FPS number in time for 1000 objects in single threaded architecture

Table 4.3. Thread statistics for single threaded architecture with 1000 objects

Cross-core context switches	Total context switches	Percent of Context Switches that Cross cores
2736	4608	59.38%
Execution time	**Synchronization time**	**Sleep**
6%	88%	6%

In comparison to the previous experiment, the loss of frames was equal to 1.75 FPS per object, which is 0.75 FPS per object bigger than in previous experiment[figure 4.4.]. This, together with a very big synchronization time[table 4.3.] indicates that there was no enough computational power to optimally support execution of both, the game engine as well as other programs ran on the computer at the time of the experiment.

4.1.2. Job system

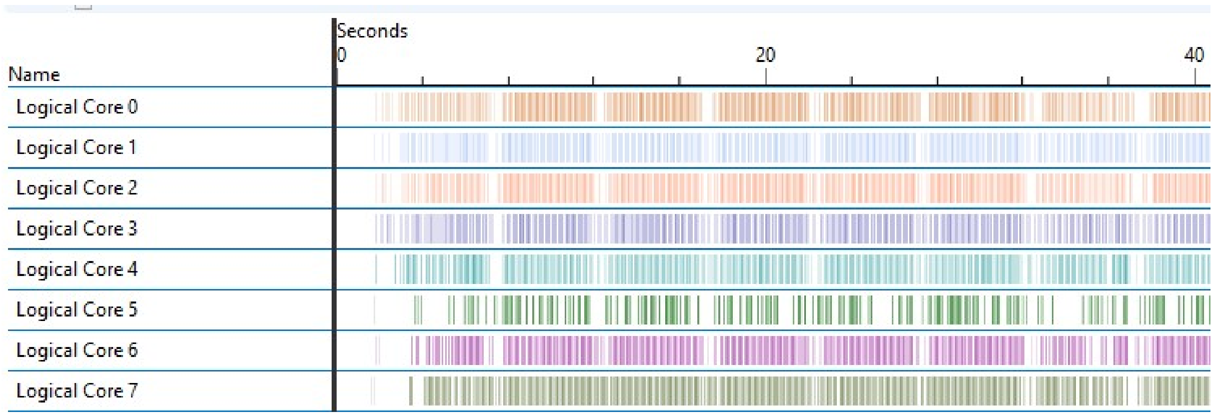


Figure 4.5. Load of each logical core for job system

From the concurrency view[figure 4.5.] for each of the cores, it is seen that thanks to the affinity set for each thread, each thread worker is situated on a specific core and executing the jobs only on that core. However, there are visible bigger gaps, meaning the thread was not executing any jobs at the time and gives a space for even bigger improvement.

For the job system, the context switching was reduced to 0% thanks to the use of setting each thread worker on a separate core.

4.1.2.1. 10 objects

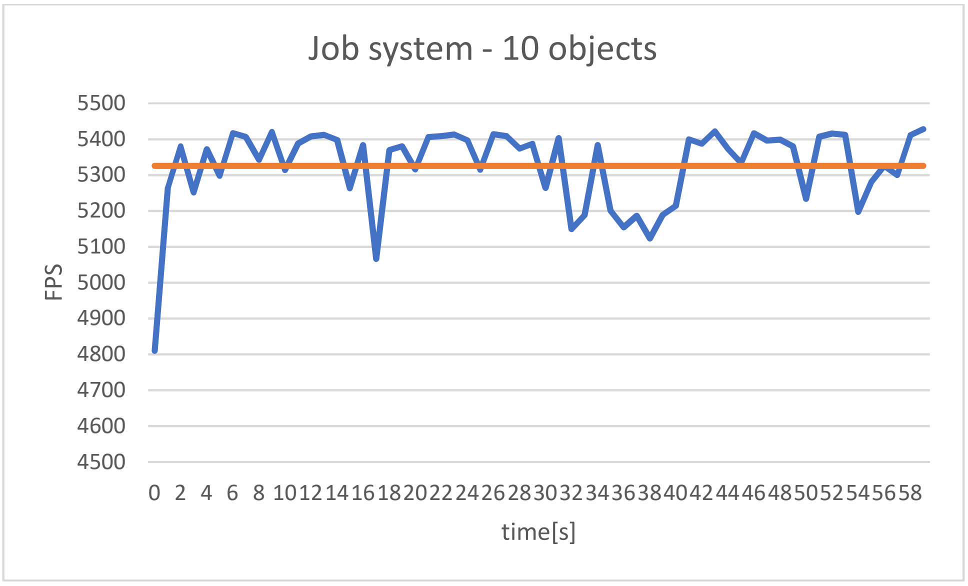


Figure 4.6. FPS number in time for 10 objects in job system

An average number of FPS for 10 objects was 5326 FPS[figure 4.6]. It means that the game engine utilizing a job system is more than 3 times faster than the one without any multithreading.

Thanks to setting each thread worker on a separate core, context switches between core were completely removed and are equal to 0 for each of the threads.

4.1.2.2. 100 objects

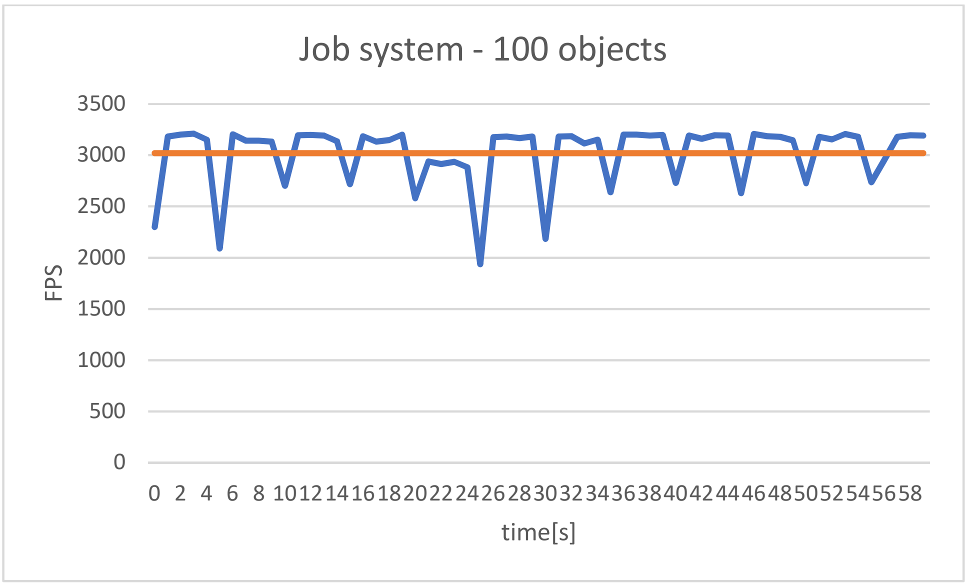


Figure 4.7. FPS number in time for 100 objects in job system

For 100 objects, the average number of FPS was 3020[figure 4.7.], which is a huge decrease (25FPS per object) for only additional 90 objects. This is due to the fact that not all thread workers are used to theirs full potential as the jobs are equally assigned to a thread worker without any differentiation, so some workers may finish sooner.

4.1.2.3. 1000 objects

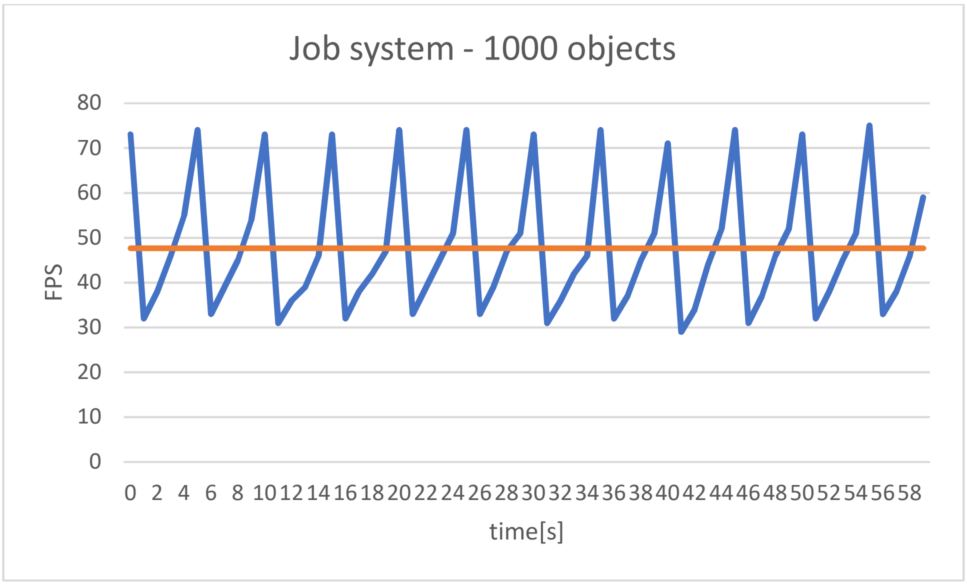


Figure 4.8. FPS number in time for 1000 objects in job system

The chart[figure 4.8.] looks almost identical to the graph of the game engine using 1000 objects with a single threaded architecture. The average number of FPS is 48, which indicates a decrease of 3FPS per object and raises a need of improving the allocation of threads (thread workers) used by each of the subsystems in order to use the full potential of the workers.

4.1.3. Job system with dynamic thread allocation

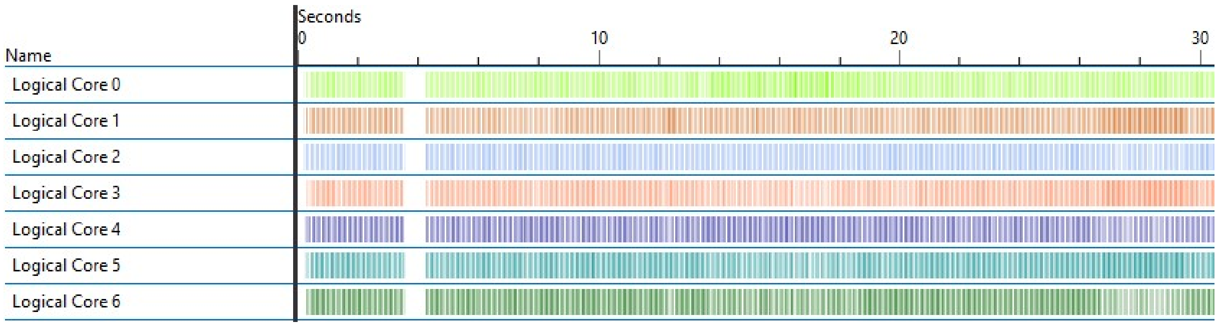


Figure 4.9. Load of each logical core for dynamic job system

From the load distribution[figure 4.9.], it is seen that the thread workers work at a very similar load, using all the available processing power of the CPU. There is also a noticeable 1 second gap, which indicates that the workers weren't executing any jobs at the moment.

The dynamic job system, similarly to job system without dynamic thread allocation, also had 0 context switches for each of its threads.

4.1.3.1. 10 objects

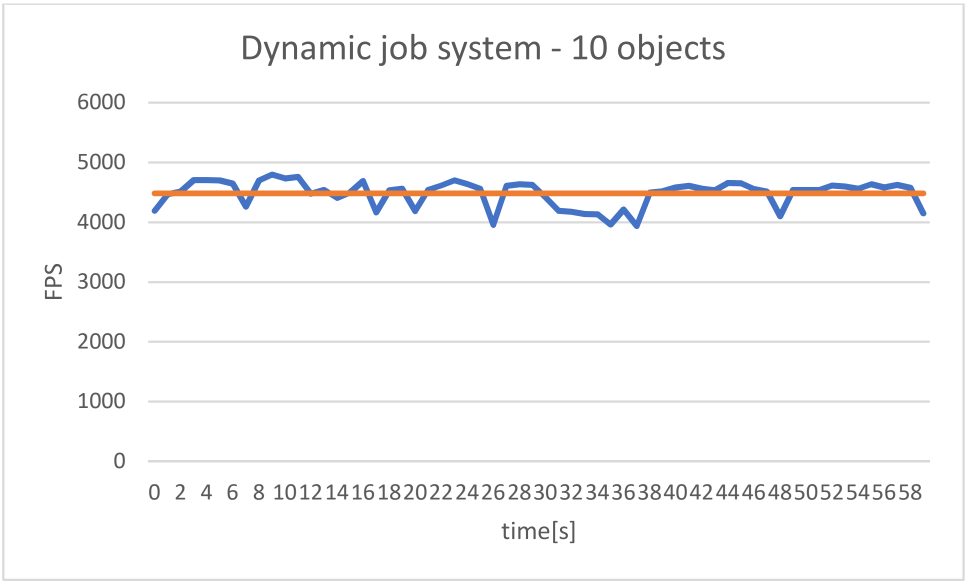


Figure 4.10. FPS number in time for 10 objects in dynamic job system

The chart[figure 4.10.] shows that the average number of FPS was 4483, which is smaller than the number of FPS in job system without the dynamic thread allocation. This is, because additionally just executing the jobs, there's a need to calculate the timers of executions of each subsystem, as well as to correctly allocate the thread workers.

4.1.3.2. 100 objects

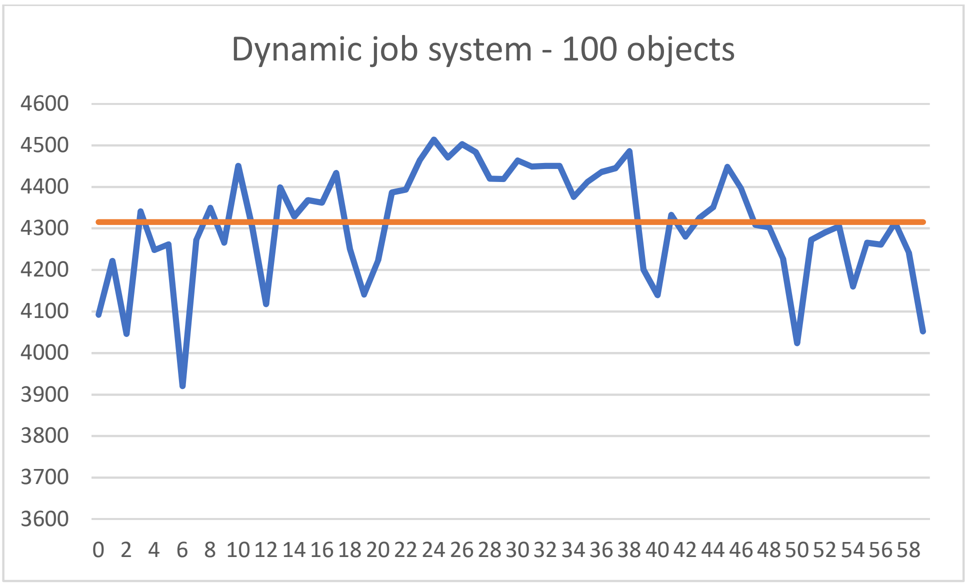


Figure 4.11. FPS number in time for 100 objects in dynamic job system

The chart[figure 4.11.] shows that the average number of FPS was 4315, which means that the number of FPS decreased about 2 FPS per object, while it seems to be a bigger decrease per object, we still need to remember that overall number of FPS is nearly 3 times higher from the single threaded approach.

Compared to the static job system, this solution is nearly 1.5 times faster in this phase of testing.

4.1.3.3. 1000 objects

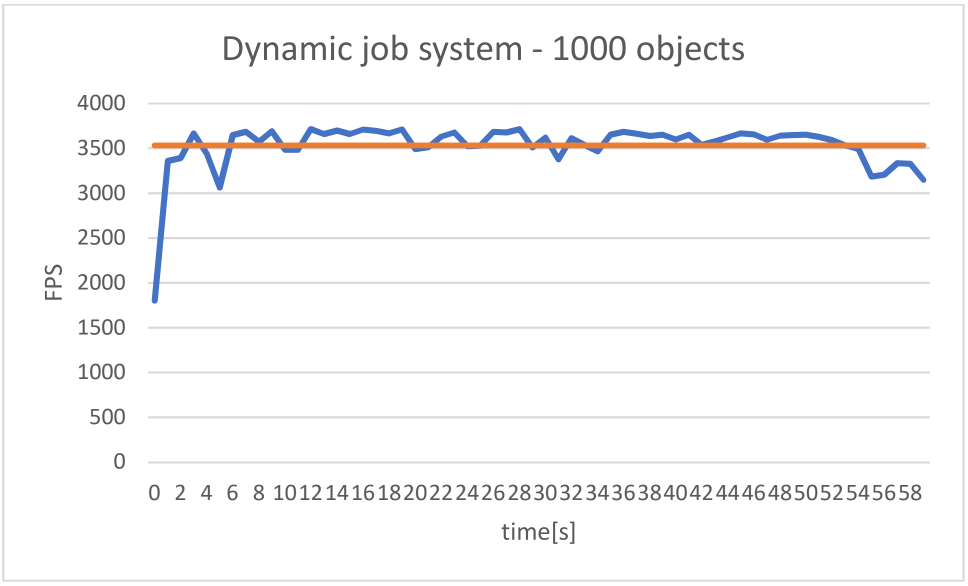


Figure 4.12. FPS number in time for 1000 objects in dynamic job system

The chart[figure 4.12.] shows that the average number of FPS was 3532, which compared to the previous attempts allows the usage of 1000 objects at an FPS rate higher by almost 74 times, which is a very huge improvement.

4.1.3.4. 5000 objects

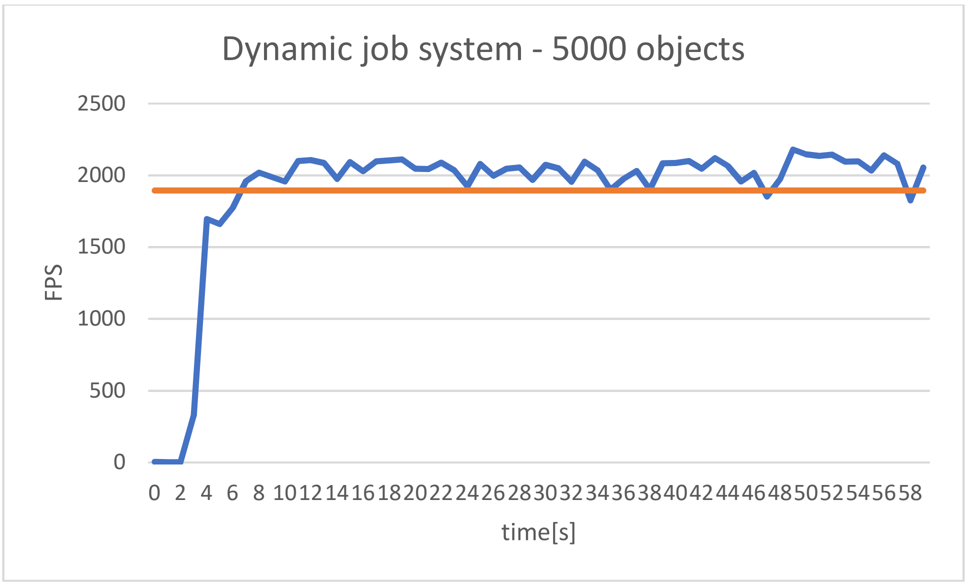


Figure 4.13. FPS number in time for 5000 objects in dynamic job system

For 5000 objects, the average number of FPS was equal to 1894[figure 4.13.], which indicates a loss of 0.49 FPS per object.

There is also a noticeable 4-second gap at the beginning of the experiment, which tells that the dynamic job system requires a longer initialization time for a large number of objects.

4.1.3.5. 8000 objects

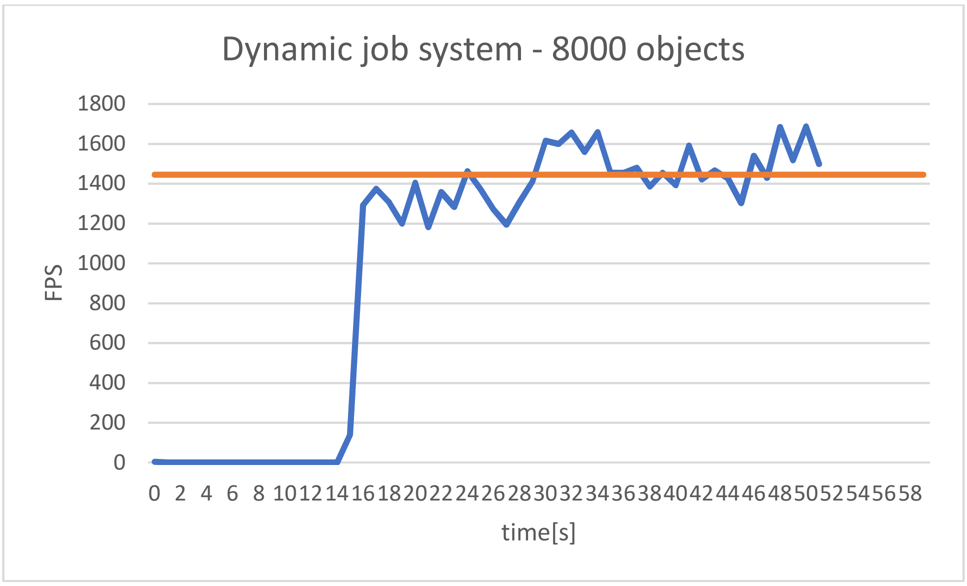


Figure 4.14. FPS number in time for 8000 objects in dynamic job system

The more object there is, the time needed for initialization is longer. For 8000 objects it was equal 16 seconds to fully initialize the whole game engine.

The average number of FPS without taking initialization in the consideration was 1444[figure 4.14.], which is still a huge number, only a bit smaller than the result for single threaded approach for 100 objects. The decrease of FPS compared to the previous experiment is equal to 0.15 FPS per object, which is still a small enough number, allowing to state that the approach is the best for large, as well as a small number of objects.

4.1.4. Summary of load testing

4.1.4.1. Statistical tests[28]

Statistical tests were performed to confirm the reliability of the results gathered during load testing. The tests were performed for single threaded architecture and architecture using job system with dynamic thread allocation for 1000 objects, as for this number of objects, it was noted as the biggest performance improvement.

The data used for statistical tests is a number of FPS for each of the architectures, which was previously presented as the load tests' results.

The tests were carried through using a MATLAB software, which is an interactive environment for performing scientific and engineering calculations, and for creating computer simulations.

First, statistics about each of the data sets were gathered[table 4.4.].

Table 4.4. Statistics for single/multi-threaded number of FPS data

Statistic	Singlethreaded	Multithreaded
Average	47.6	3531.7
Median	45	3620.5
Minimum	29	1802
Maximum	75	3714
Standard deviation	14.7	272
1. Quantile	36.5	3493
3. Quantile	53	3663

From the statistics, it is seen that a multithreaded architecture reaches much better performance than a singlethreaded architecture. However, further tests were performed to confirm this statement.

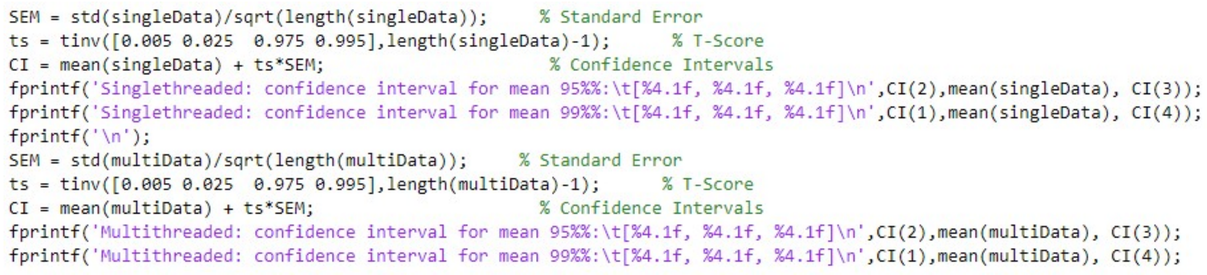
Next, a Lilliefors test was performed, which tests a decision for the null hypothesis that the data comes from a normal distribution. The alternative hypothesis tells that if the null hypothesis is rejected, the data doesn't come from a normal distribution. The test is used to decide what next tests can be used for the comparison of the sets.

For both data sets, the Lilliefors test returned a decision that the sets don't have a normal distribution, which means that there's a need to rely on mean and median values.

To confirm that the multithreaded is faster, a two-sided Wilcoxon rank sum test was used. The test, for null hypothesis checks if the data from both data sets come from a continuous distribution with equal means. The alternative hypothesis tells that they are not. However, in a right-sided test, the alternative hypothesis is that the median of multithreaded data is higher than a median of singlethreaded data, what is a goal that will confirm the reliability of performed experiments.

The result of Wilcoxon rank sum test allowed to reject the null hypothesis and accept the alternative hypothesis, which confirms that the multithreaded architecture is indeed faster from the singlethreaded architecture.

To get a closer look, how much faster is the multithreaded architecture, confidence intervals[figure 4.15.] were calculated.



```
SEM = std(singleData)/sqrt(length(singleData));     % Standard Error
ts = tinv([0.005 0.025  0.975 0.995],length(singleData)-1);     % T-Score
CI = mean(singleData) + ts*SEM;                         % Confidence Intervals
fprintf('Singlethreaded: confidence interval for mean 95%%:\t[%4.1f, %4.1f, %4.1f]\n',CI(2),mean(singleData), CI(3));
fprintf('Singlethreaded: confidence interval for mean 99%%:\t[%4.1f, %4.1f, %4.1f]\n',CI(1),mean(singleData), CI(4));
fprintf('\n');
SEM = std(multiData)/sqrt(length(multiData));     % Standard Error
ts = tinv([0.005 0.025  0.975 0.995],length(multiData)-1);     % T-Score
CI = mean(multiData) + ts*SEM;                          % Confidence Intervals
fprintf('Multithreaded: confidence interval for mean 95%%:\t[%4.1f, %4.1f, %4.1f]\n',CI(2),mean(multiData), CI(3));
fprintf('Multithreaded: confidence interval for mean 99%%:\t[%4.1f, %4.1f, %4.1f]\n',CI(1),mean(multiData), CI(4));
```

Figure 4.15. Confidence intervals calculation in MATLAB

A confidence interval informs how much the calculations can be trusted, for example regarding the average.

A level of significance α is the maximum probability that there is a mistake of the first type (i.e. the probability of rejection of the null hypothesis, even though it was true). It is the probability that the confidence interval does not contain the parameter sought, even though it contained it.

A level of confidence is a percentage that tells how accurate the confidence interval is, or how often the calculations are right. The higher the confidence level, the more often the parameter estimation is correct. The used levels of confidence were: 95% and 99%

Confidence interval for singlethreaded architecture for mean with 95% confidence:

[43.8, 47.6, 51.5]

Confidence interval for singlethreaded architecture for mean with 99% confidence:

[42.6, 47.6, 52.7]

Confidence interval for multithreaded architecture for mean with 95% confidence:

[3461.4, 3531.7, 3601.9]

Confidence interval for multithreaded architecture for mean with 99% confidence:

[3438.2, 3531.7, 3625.1]

$$worstCaseImprovement = \frac{3438.2}{52.7} = 65.24 \tag{4.1.}$$

$$averageImprovement = \frac{3531.7}{47.6} = 74.2 \tag{4.2.}$$

$$bestCaseImprovement = \frac{3625.1}{42.6} = 85.1 \tag{4.3.}$$

As it is seen from the calculations, the worst improvement of using a multithreaded architecture with a dynamic job system over a singlethreaded architecture is equal to 65.24[formula 4.1.], an average 74.2[formula 4.2.], and the best 85.1[formula 4.3.].

4.2. Functionality testing

Functionality testing is focused mainly about playing the game and noticing if something happens differently than it should. This type of testing doesn't require much knowledge about the game or any knowledge about the structure of the game's code, but usually requires knowledge about the games in general to be able to see any differences, or bugs.

The functionality testing was the most frequently used type of testing when it comes to this project. After changing a functionality or adding a new one it is advisable to check if the game is responding in the way we want by simply running the game and playing it to find out if everything is working like it should.

Functionality testing is very helpful when it comes to evaluation of game's usability, because many people, like friends, family or even strangers can help to test it and tell the developer what's wrong, and what should be changed or fixed.

There weren't any problems found while testing the functionalities of the game engine for any of its versions.

4.3. Compatibility testing

Compatibility testing was done to find out if there are any problems with the game engine on computers with different version of Windows Operating System and different software running on it. The testing also consisted of testing the game engine on computers with different graphic cards and processors to find out what are the minimal system requirements for the game.

Hardware used for the compatibility testing:

Test I – Custom built PC

- CPU - Intel® Core™ i7-7700K
- GPU – MSI GeForce GTX 1060 GAMING X 6G
- RAM – 16GB

Test II – Laptop Lenovo ThinkPad T480

- CPU - Intel® Core™ i7-8650U
- GPU – NVIDIA GeForce MX150
- RAM – 16GB

4.4. Soak testing

Soak testing is performed by leaving the game open, minimized or by performing the same operation, like mouse clicking, multiple times and is used to find memory leaks, rounding errors or float-point errors.

During soak testing of any of the game engine's versions, there were no problems found. The memory stayed constant, which indicated that there were no memory leaks, average number of FPS also didn't change by much.

4.5. Regression testing

After fixing a bug there is a possibility that the fix broke something in the game engine. The regression testing is performed to find any new bugs, memory leaks or other problems after fixing a previously found bug. If a new problem was found, it was immediately fixed, and the regression repeated until there were no remaining bugs.

5. Conclusions and future work

5.1. Conclusions

As part of the work, multiple versions of a three-dimensional graphics engine were created. Their code was used as an example to describe the architecture and implementation of such engines. The work is the most complete and effective project, which can theoretically be used in the creation of computer games and not only.

The greatest difficulty of implementation of the game engine was to properly render object that needed to be synchronized amongst many different subsystems, like rendering, physics and animation components and to synchronize the data between different, constantly running threads without breaking the engine's execution.

Based on the experiments, the best version of the game engine is a multithreaded job system with dynamic allocation of work threads based on execution time of all used subsystems. At the best, possible to compare with other solutions output, the dynamic job system provided a huge, almost 74 times faster execution time than other systems.

Additionally, the job system allows to generate a much larger number of objects. Calculating, at 60 FPS, the game engine using dynamic job system would be able to continuously generate and handle up to 17 225 objects per second. Without the need of generating the objects, which is a costly operation, the game engine could simultaneously handle much larger number of objects, achieving even much higher framerate.

However, the job system has its downsides. In the job system, every job must be fully executed before a thread worker can execute a next job, which means that there's no possibility of including jobs that require waiting for something to happen (ex. waiting for raycasting results). Putting a job to sleep would require a context switch to another job, which would involve saving the job's call stack and registers and then overriding the worker thread's call stack with the incoming job's call stack, which will definitely have impact on the game engine's performance.

Multithreading can be used as an answer to the long-lasting operations used in game engines. An example of using such solutions is the problem of dynamic loading of resources during the game, or generating many of different objects, or particles for graphic-heavy games. It happens that the limited capacity of the operating memory doesn't allow to store all elements of the game, therefore the game engine might provide the developer with an interface for performing long operations asynchronously, such as loading 3D models from the disk.

To summarize, the goals of the thesis were achieved as there are multiple 3D game engine architecture models projected. All the proposed models were successfully implemented, and the resulting applications tested.

Gathered results, confirmed with statistics tests, were enough to tell that the resulting multithreaded 3D game engine achieved a better performance than a singlethreaded counterpart.

5.2. Future work

In the future work, there will be an attempt to, using CUDA[29] (parallel computing platform and programming model for general computing on graphical processing units), utilize a graphics card in order to parallelize the game engine even more, with more processing power than is provided by the processors.

As the job system using thread workers is not ideal, there's a plan to introduce a few more approaches of upgrading the job system in order to choose the best one:

- jobs as coroutines[30] – an approach that has an ability to stop execution of a current coroutine in order to run another one with a possibility to continue from where it left off when another coroutine finishes execution or yields back the control. It is due to the fact that coroutines swap the callstacks of jobs within the thread, which gives a possibility of putting one job to sleep and continue execution of other jobs at the same time
- jobs as fibers[31] – the first step of fiber system is to convert one of its threads into a fiber. The thread executes the fiber until a method is called that switches the control to another fiber. Similarly, to coroutines, the fiber system saves the entire call stack of a job when switching fibers. This approach allows to transfer fibers between different threads

The next goal of the implementation of the game engine is to introduce networking to see how the engine will perform when in a need to additionally send many requests to a server at the same time.

6. Bibliography

[1] B. Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[2] S. Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[3] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Use of the Standard Template Library*. 1996.

[4] S. Meyers. "*Specific Ways to Improve Your Use of the Standard Template Library.*" Addison-Wesley, Reading, MA 142, 2001.

[5] J. Lakos. *Large-scale C++ software design.* Reading, MA 173, 1996.

[6] D. Kodicek. *Mathematics and Physics for Game Programmers.* Hingham, MA: Charles River Media, 2005.

[7] E. Lengyel. *Mathematics for 3D game programming and computer graphics.* Cengage learning, 2012.

[8] A. Grama. et al. *Introduction to parallel computing*. Pearson Education, 2003.

[9] J. Gregory. *Game engine architecture*. AK Peters/CRC Press, 2019.

[10] R. Williams. *The animator's survival kit: a manual of methods, principles and formulas for classical, computer, games, stop motion and internet animators*. Macmillan, 2012.

[11] R. Stevens and D. Raybould. *The game audio tutorial: A practical guide to creating and implementing sound and music for interactive games*. Routledge, 2013.

[12] M. A. DeLoura. *Game programming gems 2*. Cengage learning, 2001.

[13] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2018.

[14] W. F. Engel. *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*. Wordware Publishing, 2004.

[15] A. Williams. *C++ concurrency in action*. Manning, 2019.

[16] D. Eberly. *3D game engine architecture: engineering real-time applications with wild magic*. CRC Press, 2004.

[17] B. T. Phong. *Illumination for computer generated pictures.* Communications of the ACM 18.6 (1975): 311-317.

[18] D. H. Eberly. *Game physics*. CRC Press, 2010.

[19] G. Van Den Bergen. *Collision detection in interactive 3D environments*. CRC Press, 2003.

[20] *Visual Studio*. Microsoft[Online]. Available: https://visualstudio.microsoft.com/ [Accessed 13 June 2019]

[21] J. Loeliger, and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[22] D. H. Eberly. *3D game engine design: a practical approach to real-time computer graphics*. CRC Press, 2006.

[23] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[24] F. Luna. *Introduction to 3D game programming with DirectX 11*. Stylus Publishing, LLC, 2012.

[25] I. Millington. *Game physics engine development*. CRC Press, 2007.

[26] B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[27] G. J. Myers. *The art of software testing.* Vol. 2. Chichester: John Wiley & Sons, 2004.

[28] L. M. Moore. *The basic practice of statistics.* 1996.

[29] Nvidia, C. U. D. A. *Nvidia cuda c programming guide.* Nvidia Corporation. 2011.

[30] G. Kahn, and D. MacQueen. *Coroutines and networks of parallel processes*. 1976.

[31] S. Dolan. *Effective concurrency through algebraic effects*. OCaml Workshop. 2015.

List of drawings

Figure 1.1. Backface culling, p. 18

Figure 3.1. Declaration of StackAllocator, p. 48

Figure 3.2. Game loop implementation, p. 52

Figure 3.3. Definition of a function returning number of processors/cores, p. 53

Figure 3.4. Definition of worker's main function, p. 54

Figure 3.5. Game timer declaration, p. 55

Figure 3.6. Definition of main window procedure callback, p. 56

Figure 3.7. Creation of game engine's window, p. 56

Figure 3.8. Definition of input handling function, p. 57

Figure 3.9. Verbosity declaration, p. 57

Figure 3.10. Definition of logging function, p. 58

Figure 3.11. Shadow map (in the corner), p. 59

Figure 3.12. D3D12CreateDevice function declaration, p. 60

Figure 3.13. CreateFence function declaration, p. 60

Figure 3.14. DXGI_SWAP_CHAIN_DESC declaration, p. 61

Figure 3.15. Material declaration, p. 62

Figure 4.1. Load of each logical core for single threaded system, p. 64

Figure 4.3. FPS number in time for 100 objects in singlethreaded architecture, p. 65

Figure 4.4. FPS number in time for 1000 objects, p. 66

Figure 4.5. Load of each logical core for job system, p. 67

Figure 4.6. FPS number in time for 10 objects in job system, p. 67

Figure 4.7. FPS number in time for 100 objects in job system, p. 68

Figure 4.8. FPS number in time for 1000 objects in job system, p. 68

Figure 4.9. Load of each logical core for dynamic job system, p. 69

Figure 4.10. FPS number in time for 10 objects in dynamic job system, p. 69

Figure 4.11. FPS number in time for 100 objects in dynamic job system, p. 70

Figure 4.12. FPS number in time for 1000 objects in dynamic job system, p. 70

Figure 4.13. FPS number in time for 5000 objects in dynamic job system, p. 71

Figure 4.14. FPS number in time for 8000 objects in dynamic job system, p. 71

Figure 4.15. Confidence intervals calculation in MATLAB, p. 73

List of tables

Table 3.1. Functions used for operations on files, p. 49

Table 4.1. Thread statistics for single threaded architecture with 10 objects, p. 65

Table 4.2. Thread statistics for single threaded architecture with 100 objects, p. 66

Table 4.3. Thread statistics for single threaded architecture with 1000 objects, p. 66

Table 4.4. Statistics for single/multi-threaded number of FPS data, p. 72