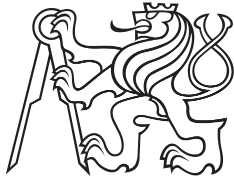
Bachelor Project



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Comparison of Unity and Unreal Engine

Antonín Šmíd

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: STM, Web and Multimedia
May 2017

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Antonín Šmíd

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: Srovnání Unity a Unreal Enginu

Pokyny pro vypracování:

Nastudujte možnosti vývojářských platforem pro tvorbu počítačových her Unity a Unreal Engine. Porovnejte tyto platformy pomocí realizace stejné hry přiměřené složitosti na obou platformách (např. PacMan). Zaměřte se především na zobrazovací možnosti srovnávaných nástrojů, jako jsou podporované materiály, světla, osvětlovací mapy, reflektivní mapy, podpora shaderů a postprocessing. Vyhodnoťte vizuální kvalitu výstupu subjektivním srovnáním a srovnáním s výstupem z offline fotorealistické simulace. Zhodnoťte vliv jednotlivých efektů na rychlost zobrazování na čtyřech platformách: notebook, herní PC, mobilní telefon a GearVR. Identifikujte úzká hrdla zobrazovacího řetězce pro srovnávané nástroje na všech testovaných platformách.

Seznam odborné literatury:

[1] Adam Watkins. Creating Games with Unity and Maya, Focal Press, 2011.
[2] David H. Eberly. 3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic, Morgan Kaufmann, 2004.
[3] Jesse Schell. The Art of Game Design: A book of lenses. CRC Press, 2008.
[4] Tomas Akenine-Moller, Eric Haines, Naty Hoffman. Real-Time Rendering. A K Peters, 2008.

Vedoucí: doc. Ing. Jiří Bittner, Ph.D.

Platnost zadání: do konce zimního semestru 2018/2019

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4.4.2017

Acknowledgements

I am grateful to Jiri Bittner, associate professor, in the Department of Computer Graphics and Interaction. I am thankful to him for sharing expertise, and sincere guidance and encouragement extended to me.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used. I have no objection to usage of this work in compliance with the act §60 Zákon c. 121/2000Sb. (copyright law), and with the rights connected with the Copyright Act including the amendments to the act.

In Prague, 25. May 2017

Abstract

Contemporary game engines are invaluable tools for game development. There are numerous engines available, each of which excels in certain features. To compare them I have developed a simple game engine benchmark using a scalable 3D reimplementation of the classical Pac-Man game.

The benchmark is designed to employ all important game engine components such as path finding, physics, animation, scripting, and various rendering features. In this thesis I present results of this benchmark evaluated in the Unity game engine and Unreal Engine on different hardware platforms.

Keywords: Game Engine, Benchmark, Unity, Unreal, 3D Pac-Man reimplementation

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
GCGI, FEE, CTU in Prague
Praha 2, Karlovo náměstí 13

Abstrakt

Současné herní engine jsou důležitými nástroji pro vývoj her. Na trhu je množství enginů a každý z nich vyniká v určitých vlastnostech. Abych srovnal výkon dvou z nich, vyvinul jsem jednoduchý benchmark za použití škálovatelné 3D reimplementace klasické hry Pac-Man.

Benchmark je navržený tak, aby využil všechny důležité komponenty herního enginu, jako je hledání cest, fyzika, animace, scriptování a různé zobrazovací funkce. V této práci prezentuji výsledky benchmarku, který jsem implementoval a vyhodnotil v enginech Unity a Unreal na různých platformách.

Klíčová slova: Game Engine, Benchmark, Unity, Unreal, 3D Pac-Man reimplementace

Překlad názvu: Srovnání Unity a Unreal Enginu

Contents

1 Introduction 1

2 Game Engines 3

2.1 Brief introduction into the game development ... 3

2.2 Components of the game engine ... 5

2.2.1 Rendering ... 6

2.2.2 Animation ... 7

2.2.3 Physical engine ... 9

2.2.4 Artificial intelligence ... 9

2.2.5 Scripting ... 10

2.2.6 Audio ... 10

2.3 Contemporary game engines ... 11

2.3.1 Unity 3D Engine ... 11

2.3.2 Unreal Engine 4 ... 12

2.3.3 Cry Engine 3 ... 12

3 The Pac-Man Benchmark 13

3.1 Game Concept ... 14

3.2 The Maze ... 14

3.3 Pac-Man Movement ... 15

3.4 AI characters ... 17

3.5 Maze generator ... 18

3.6 Scaling the problem ... 20

3.7 VR adjustments ... 21

4 Unity implementation 23

4.1 Game architecture and components ... 23

4.2 Ghost navigation ... 25

4.3 Visuals ... 25

4.3.1 Models and animations ... 26

4.3.2 Materials ... 26

4.3.3 Lighting ... 29

4.3.4 User Interface ... 29

4.3.5 Screen-space effects ... 29

4.4 Profiling ... 30

5 Unreal implementation 31

5.1 Game architecture and components ... 33

5.2 Ghost navigation ... 34

5.3 Visuals ... 35

5.3.1 Models and animations ... 35

5.3.2 Materials ... 36

5.3.3 Lighting ... 38

5.3.4 User Interface ... 38

5.3.5 Screen-space effects ... 38

5.4 Profiling ... 40

6 Results and Comparison 41

6.1 Features 41

6.2 Benchmarks 45

6.3 Visual quality 50

6.4 Subjective developer's opinion . . 58

7 Conclusions 61

References 63

Figures 65

A CD Contents 69

Chapter 1

Introduction

Game engines are complex, multipurpose tools for the creation of games and multimedia content. They offer an environment for an efficient development, sometimes even without the knowledge of scripting. Game engines should cover many different areas of the game development process such as rendering, physics, audio, animation, artificial intelligence, and the creation of the user interface.

Part of the development team are usually the artists (level designers, modelers, animators) who are unable to work with code and their tasks require a visual environment. The team can either develop a custom environment or licence an existing middleware solution. The choice of the game engine in the early phase of the project is crucial. That is why I have decided to compare two of the major engines[Web] on the today's market: Unity3D and Unreal Engine 4.

The definition of the game engine itself is a complicated issue whereas the game engine's architecture may vary greatly. Monolithic systems provide complete out of the box solutions while *the modular component engines* [Ac] may offer just API and the developer has to code all the logic including the game loop.

Anderson et al. [Ac] have covered the problematics of defining an engine and proposed a Whiteroom Benchmark for Game Engine Selection that compares the engines according to the features. The Whiteroom demonstrates basic gameplay features such as doors that can be opened, player's interaction with objects, stairs/steps, and elevators. The benchmark is designed to use the technical features considered standard in modern games. They have evaluated four engines: Source, Unity, Unreal, and CryEngine and provided the implementation notes.

Another engine selection methodology proposed by Petridis et al. [PDdFP10] empathizes the rendering features. However, those papers do not mention any performance measurements and I would like to include those as well.

Game engines are complex tools and comparing them is a problematic task. It is possible to realize a subjective comparison if we have a common experience with implementing the same project on both platforms or an objective comparison where we evaluate both implementations from the perspective of measurable criteria. For comparison, it is important to have a project of adequate complexity, which can be implemented on both platforms in a very similar way. That is the task I aim to accomplish in this thesis.

I have developed a simple benchmark using a scalable reimplementation of the classic Pac-Man game [Pit]. The benchmark is designed to employ all important game engine components including various rendering features, path finding, physics, animation, and scripting. I have prepared three versions of the benchmark to run on different hardware platforms. Apart from the full benchmark for PC, I have evaluated an Android build with touch controls and simplified GearVR [Par15] build, to test the virtual reality performance.

In this bachelor's thesis, I will breathy cover the game development process and I will describe the game engine's components in Chapter 2. Then I analyze the Pac-Man game characteristics and explain how I use them in the benchmark. In Chapters 4 and 5 I will cover the implementations of the benchmark in Unity and Unreal. I will go through the individual tools that I have used during the implementations, explain their purpose and behavior.

Finally, in Chapter 6 I present the measured data and show images from the game compared to the offline render from Blender Cycles [FS15]. I will point out the differences and reasons for them. In the very end I will present my humble opinions and recommendations from a perspective of the benchmark developer.

Chapter 2

Game Engines

2.1 Brief introduction into the game development

Game development is a complex collaborative discipline [Sch08]. Considering AAA titles, the process might take even five years (Grand Theft Auto V). The game studios have tens to hundreds of employees in various fields. Examples of the most common professions are the concept artists, scriptwriters, the game designers, the engine programmers, the game programmers. In every game studio, there are 3D artists such as the modelers, character designers, the material designers, riggers, people who care about the lighting setup, particle effects, simulations, post process effects, then there are animators, motion capture actors... and directors who manage the whole process. Creation of an interactive game is comparable to the shooting of a movie. It is very expensive; it employs many different professions, it takes a lot of time, and there is one more similarity: The success of the final project is unsure. When the game is released, the gamer community may or may not like it.

In the beginning, there has to be an idea. Something that makes the game innovative, something that attracts the players. It might be completely new gameplay concept or just an old well-established concept with a few innovative elements. Usually, the basic game types are distinguished based on the camera position. Common game types are 2D platformer (Mario), 2D from top view (Bulánci), 3D First person shooter (Counter strike), 3D third person shooter (GTA), 3D strategies from the top view (Civilization), MMO(Massive multiplayer online games like World of Warcraft).

Let's look closer at the First person shooter. This game concept simulates the view from eyes of a character. The player uses the mouse to look around and WSAD keys to walk. He can see his arms, his weapon, but

usually, he can not see his body. This game concept is so well approved, that most of the shooter games just use it. The players are used to controlling it, for most of them it feels natural. By using the first person shooter controls, the developers know it will surely work, and they can focus on other gameplay features. Fig. 2.1 shows the typical first person view.

Figure 2.1: Example of a first person view. Player is looking through the characters eyes, he can see his arms and weapon. Image courtesy Crytek.

Then every game needs a story, characters that live the story and an environment. The authors have to make decisions about the visual style of the game. Is it supposed to look realistic, cartoonish, hand-drawn? It is also important to specify the target audience to answer these questions.

When the story, stylization, game type and environment are clear, the process continues by prototyping the gameplay and creation of the assets which fill the environment. Usually, the concept artists draw images based on the art directors ideas and the modelers then create characters or environments based on these concept images. Level designers take those 3D models and compose them into the playable game environment. They create scripts to make the level interactive and add the AI characters.

Until now, we have not mentioned the core component of the entire development. There needs to be a common environment for coders and artists to put the game together. There has to be a component that renders the assets, takes care of the game logic, the AI, the sound and the network multiplayer. This core component is called the game engine.

As we may anticipate, there are high expectations from the game engine. Firstly, the engine needs to render objects in real-time. It communicates with the graphics card and creates an environment for the artists to display and work with their assets without any knowledge of programming. The models have materials and textures, there are lights in the scene, there may

be particles, but most importantly there is a camera, that looks at the scene.

Secondly, the engine needs to handle the animations. There are various types of animations: simple change of parameters over time (position, rotation, color, transparency), skeletal animations (moving characters with bones) or morphing animations for facial expressions (changing the positions of vertices inside an object). Game engine should be able to play, edit, script or blend those animations together.

The game engine should provide an environment for scripting the game logic. This may involve scripting languages or visual scripting solution, prepared structure, and interfaces for the objects, messaging systems, event systems, debugging solution, error handling and optimization tools.

There are many other functions that the engines have, it depends on the particular engine whether or how it implements them. Most of the games need audio handling, some physics solution, support for an artificial intelligence or building of the User Interface. An engine can have a terrain builder, foliage tools or cinematic tools. In these areas, the engines differ greatly.

The game engine is a key component of the game development process. The choice of a game engine is crucial. Large game studios build upon their proprietary engines, however, for mid-size studios, it is better to use some of the already existing middleware solutions. It is difficult to choose an engine based on the engines propagation materials or official documentation, that is why I have compared two of them in this thesis.

2.2 Components of the game engine

Contemporary game engines build upon the components architecture. Game engines are divided into several components [Ebe04], each one providing special functionality (see Fig. 2.2). I will briefly go through the most important ones.

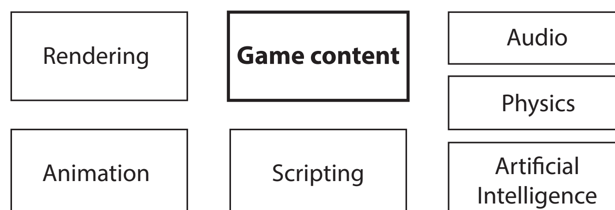


Figure 2.2: Game engines components overview.

2.2.1 Rendering

When we want to render 3D objects on the screen, our program calls the graphical API such as OpenGL or DirectX. This interface enables us to control the graphics card and make it display our objects. These APIs are low-level, for simple game development, we do not want access them directly because we would spend much more time on setup and low-level programming.

Render engine provides rendering functionalities literally at the press of a button. We just set the objects in the scene, set up the camera, lighting, materials and textures. When we compile the game, the render engine takes care of rendering our scene properly. It usually comes together with a number of shaders which we can use to simulate different materials or effects.

Render engine is the most important part of the game engine. Apart from the gameplay, the visual presentation of the game is rated. The rendering takes the majority of the update time. It is not unusual that over 90% of the whole calculation is just rendering. Game logic, scripts, physics, sound have to share the rest. Gamers expect better visual quality every year. At the same time, the game has to run on the low-end machines as well. This is pressuring the render engines to be as optimized as possible with visual quality configurable during the gameplay.

In the scene, there are objects at certain positions and camera. The camera has a configuration like real world camera lens. The most important parameter is the view angle, in first person games, it is usually set between 60 to 90 degrees, which is equivalent to 18-28mm lens on Full Frame. Near plane and far clipping planes define the distance from the camera, where the objects are visible. With this information, the game engine can calculate the matrix to transform the objects from the world coordinates to the viewport of the camera. Then the process of occlusion culling [AMMH08](page 661) cuts off all the objects, which are out of the viewing frustum (Fig. 2.3). The remaining objects are sorted by the depth and rendered.

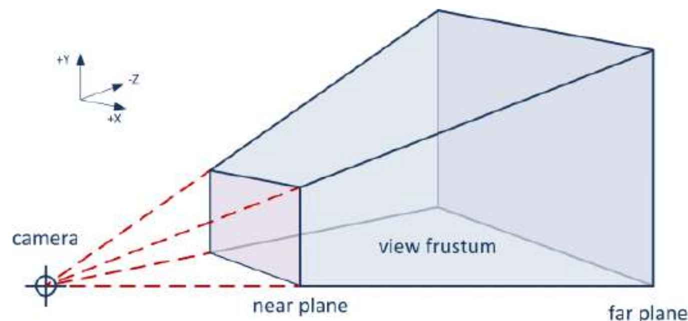


Figure 2.3: Viewing frustum of the camera.

There are different approaches to the object rendering. The traditional Forward rendering calculates the lights and materials for every visible geometry and after that solves which one of them is nearest to the camera, this one is displayed. The current approach that Unity and Unreal use is called Deferred rendering [AMMH08](pages 279-283). Multiple passes of the geometry are rendered (Depth, Normal, Color...). The shading is the calculated based on these passes. Only the visible fragments are shaded. Deferred rendering can efficiently handle large amounts of lights, however, it does not help with the shadows rendering there is no anti-aliasing and we can't render transparent objects using this technique.

The game engine usually provides materials or ways to create and configure them. The materials can be unlit (just color, lights have no effect), lit (shaded based on the lights), transparent or translucent. Textures are images that help to define the material. There are different kinds of textures that describe the color, roughness, reflectiveness or displacement of the material.

When the scene is rendered, the engine applies post process effects. These effects improve the visual quality of the final result. The most common effects are anti-aliasing (smooths the jagged edges), ambient occlusion (darkens the corners) and color corrections of the final image.

2.2.2 Animation

Not every object in the scene is a static one. Objects can be moved by physics or animated. Almost in every game, there is some animation needed. Here I consider animation a predefined change in certain parameters over time. We can animate position; the whole characters can be animated with skeletons or the vertices inside the mesh can be formed into various poses and animated.

Simple animations are usually created inside the engine. Let's say we want the door to open. That means we trigger an opening animation when the player comes close to the door and presses the action button. Door opening animation is a change in the rotation. Because each door may open in a different way, we do not want to animate it in the 3D software; it is better to do so inside the engine itself. Game engines offer an environment for animation. It usually looks like a graph. On X axes there is time and on the Y axes there the value we want to animate. It may be any possible value that can be set in the engine. Usually, it should be possible to change the shape of the curve to edit the movement.

More complex types are skeletal animations. To make a character move, we need to rig it. A rig is essentially a skeleton inside the mesh and connection

of bones to certain parts of the mesh. So the bones inside arm really move the vertices on the arm. Rigged characters can be animated by hand, or by using motion capture techniques. The final result is, for example, the walk cycle.

The game engine is not supposed to provide an environment for the creation of skeletal animations. However, it should be able to work with them. The engine should be able to play and blend those animations together so that they create continuous fluid motion. Player then does not perceive discrete playback of animations, but a living character that can do various tasks. For controlling the character animations, there are state machines which decide what state the character is currently in a what animation to play. The systems can mix more animations together and modify them to achieve most fluid results.

Lastly, the shape of the object can be animated as well. That is useful for creating facial expressions (Fig. 2.4). The different poses are imported with the object into the engine. One of the poses is always the base pose. The other poses are handled as relative distortions of the main pose. If the distortion is set to one, the character is fully in the next pose. It is great that the pose values can be out of the interval [0-1], they can be even negative. All of them are then blended together to create the pose. These values can be animated in the same way as any others. In Unity, this technique is called Blend Shapes, in Unreal Morph Targets.

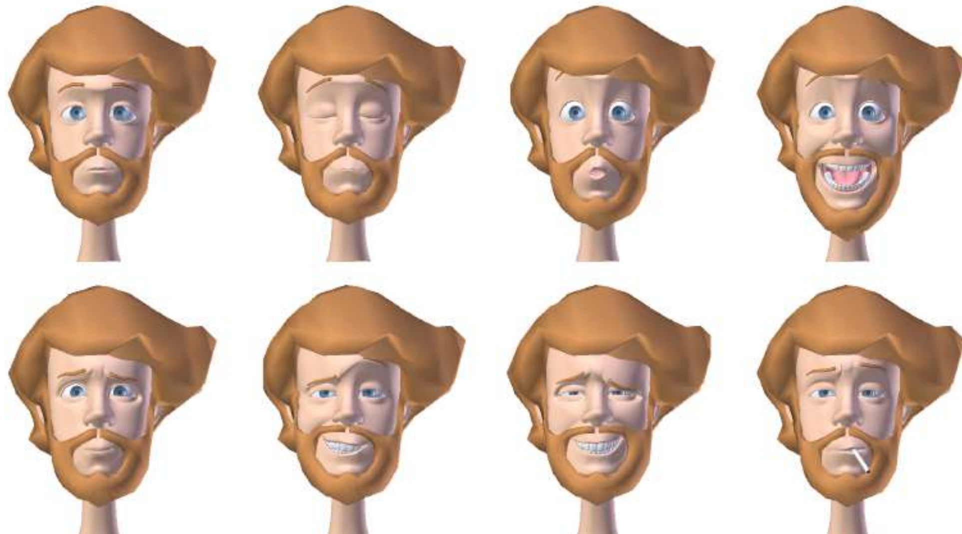


Figure 2.4: Victor facial samples - an example of using blend shapes as facial expressions. Image courtesy Cosmos Laundromat, opensource film

2.2.3 Physical engine

Physics is an important part of the gameplay. Most of the games take place in real world, so the objects within the scenes should act in a physically correct way. The game engine usually provides a solution for such physical simulations. Each object has a collider (simplified boundary of the object used for physics) and some physical material which determines the mass and properties of the surface. The engine calculates movement vectors and collisions between the objects. We can distinguish between static objects that do not move at all like the ground and rigid bodies which have a mass, a material with friction, which reacts to forces, falls with gravity, etc. Additionally, the engine might simulate soft bodies, which change their shape according to outer forces (cloth). Although the physics simulation does not have to be extremely accurate, it is an important part of the game development. Some engines use external libraries for physics, such as Havok or the NVidia GPU powered PhysX.

2.2.4 Artificial intelligence

Apart from the player's character video games usually contain enemies to fight against. These non-player characters need a certain amount of intelligence to behave reasonably. Especially single player off-line games need the artificial intelligence to be a bit clever so that the player does not get bored.

The main tasks of the AI are finding a path from point A to point B and walking with the character along that path. Game engines provide solutions for such tasks. Usually, the engine generates a navigation surface called NavMesh (see Fig. 2.5) which is based on the static objects in the scene. This mesh can be dynamically modified during the gameplay according to the characters movement or rigid body physics. The AI characters are able to walk on that mesh, and they find paths on it. The mesh can be divided into areas with different movement costs, for example, road, mud, swamp, water. According to the area, the AI might choose an appropriate animation.

The AI characters also have state machines, controllers and act according to certain rules. For example in Unreal Engine, there are behavioral trees implemented. These trees are graphs that the controller use to decide what task the character should do. However, the game engine is not supposed to program the AI intelligence instead of the developers. It just offers a set of tools and systems that the developers can use, to make their AI behave as they need.

Figure 2.5: NavMesh of the Pac-Man benchmark in Unreal Engine.

2.2.5 Scripting

The game engine also has to provide a way to describe the game components behavior. We can write scripts as components for objects so that the objects can react to player's impulses and interact with each other. In Unity, the scripting language is C# and javascript, Unreal offers native C++ or Blueprint visual scripting and Cry Engine scripts in C++ or Lua. It is important that the scripts can be added and reused to object in the scene in a way that even a non-coder game designer could use them. It depends on the engine's editor how it deals with this issue. In Unity, the game designer can add the script as a component of an object and set the properties manually. Moreover, in Unreal, there is the whole Blueprint concept of visual coding which allows the user to create complicated logic without writing a single line of code.

2.2.6 Audio

An important part of every game is a sound design and music which induces the atmosphere. Game engines provide tools for playing and stopping soundtracks based on game events. Advanced audio engines can simulate echo in the 3D space or play spatial sound according to player's position.

2.3 Contemporary game engines

There is a competition between the game engines. The big companies such as Ubisoft, Rockstar games or even Bohemia Interactive use their proprietary engines. The third party engines are used mainly by independent developers or smaller studios. The most used game engine in 2016 was Unity [Web] (fig. 2.6). And it is probable it will stay this way because Unity is so easy to use that everyone can start developing with it. The closest competition to Unity with only 17% of its users is the Unreal Engine. Therefore I am comparing those two engines in this thesis. Very powerful and used is also the Cry Engine. But it is very complicated and advanced tool, not recommended for beginners, so it is used mainly with studios.

Figure 2.6: Logotypes of the most used 3rd party engines.

2.3.1 Unity 3D Engine

Unity Technologies company develops Unity game engine which is currently in version 5.6.0. It is the world's most used engine. The first version was released at Apple conference in 2005 and targeted only OS X development. Since then Unity has developed and currently supports 27 platforms including VR.

The most important quality of Unity is the ease of use. Development in Unity is very fast, especially on mobile platforms. The projects and the build games are small, and the export process is rather simple. The component architecture of the engine is easy to understand. Scripting in C# is fast and efficient. Unity has a big community; there are forums full of answers which make the debugging easier. And lastly, there is an asset store which is relatively cheap and contains a lot of useful assets. Only regarding graphics Unity is a bit worse looking compared to Unreal or Cry Engine, and it does not have good support for foliage and terrains.

2.3.2 Unreal Engine 4

Unreal engine is being developed by the Epic Games company. It is currently the leading engine in realistic visualization, vegetation, and terrain creation. Unreal engine is suited for larger projects. It is not suitable for developing small mobile games, although it supports the iOS and Android support. Unreal has Blueprint system for visual scripting. Blueprints are graphs made of blocks connected together. The connection creates certain logic instead of the scripts. Unreal has opened source code in C++. Also the coding language is C++ giving the developers great control over the whole system which is great. However, the framework is complicated and difficult to learn. The rendering technology is a big benefit, the post-process effects are fast and support many features. Unreal has an editor for creation of custom materials. Unreal provides great tools for optimization and visual debugging. Lastly, Unreal does not have such a large community as Unity, and the documentation is weak at some points.

2.3.3 Cry Engine 3

Cry Engine is a powerful engine designed by Crytek company. It is targeted on PC and consoles. Cry engine has a state-of-the-art lighting; the rendering capabilities are very high. Also the animation systems are advanced. Cry engine also has powerful level design tools and supports dense vegetation. However, it is not very user-friendly and does not suit for beginners. Logic in Cry Engine is done by C++ and Lua scripts.

Chapter 3

The Pac-Man Benchmark

In this chapter, I will describe the Pac-Man game mechanics in the context of the tested components.

Game design is a difficult discipline [Sch08], one of the well-done designs is the Pac-Man game. Pac-Man is one of the most iconic video games of all time. It is not too complex, but it still has a potential to employ many components of the game engine. The original Pac-Man was an arcade game developed by the Namco company [1], released in 1980 in Japan [Lon07]. Pac-Man (Figure 3.1)is a yellow ball with big mouth eating little dots, also called biscuits. The player controls the Pac-Man through the maze. There are four ghosts in the maze, who are trying to catch the Pac-Man.

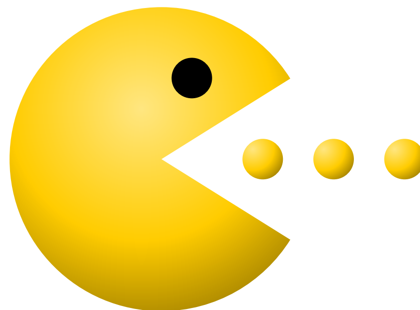


Figure 3.1: Pac-Man, the main character.

The game is simple to play; originally it was controlled by a joystick, in this benchmark version I use a script to move the Pac-Man precisely or keyboard arrows to make the game actually playable. I have transferred the maze into today's graphics, using physically based shaders. There is a physical component used for moving the characters around. The benchmark also uses navigation AI component to manage the ghost's movements.

[1] http://pacman.com/en/

3.1 Game Concept

The Pac-Man is an arcade game. The main yellow character is going through the maze, eats biscuits and avoids the ghosts. Ghosts begin in prison located in the center of the maze. Pac-Man has three lives, if he gets caught, he loses one.

There are ten score points for each biscuit. There are also four larger biscuits called the energizers. When Pac-Man eats such a biscuit, he gets energized for a while. The fortune is changed, and he can chase the ghosts. When a ghost is caught, he moves back to ghost prison and player's score is increased by 200 points for the first ghost, 400 for the second. When Pac-Man eats all the biscuits player has completed the level, the maze restarts and the chase starts again. In the next level, ghosts move slightly faster. This cycle goes on until the player loses all his lives.

3.2 The Maze

In the original Pac-Man, there is a static maze (Figure 3.2). The large orange circles are the energizers. In the Pac-Man benchmark, I generate the maze using a script. The individual biscuits are instantiated as dynamic objects at the beginning of every level. The number of separate objects increases the draw calls amount, which is performance heavy for the rendering engine and tests how efficiently it can batch the draw calls.

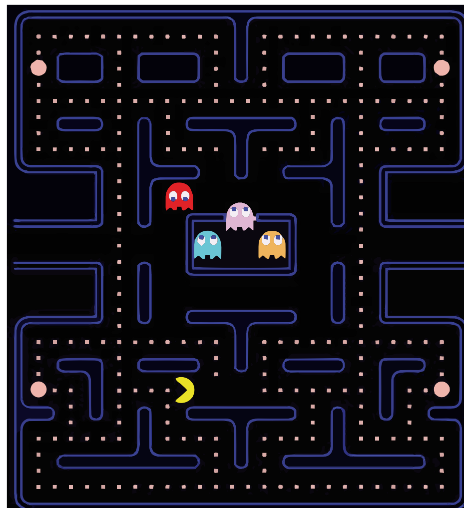


Figure 3.2: Screen from the original Pac-Man maze. Image courtesy of Shaune Williams.

In the middle of the maze, there is a prison for three ghosts. Pac-Man starts at the first junction exactly under the prison. There are no dead ends. On the left and right side, there are corridors which lead into darkness. Those corridors act as teleports. When a game character enters the teleport, it appears on the other side. The dimensions of the maze are 28x31 tiles; it is symmetrical around the Y axis. The prison is 8x5 tiles including walls. From the logical perspective of the game the corridors have width of one tile, however visually they appear two tiles wide.

The maze itself (Figure 3.3) offers a variety of opportunities to test the rendering engine. In Pac-Man benchmark the walls are covered with a displacement map, there are models of small roofs on the walls. The material on those roofs uses hardware tessellation to create roof tiles. Moreover, there is grass on the ground. The grass consists of many tussocks with alpha texture to test the engines ability to handle transparent materials. The direct shadows are computed in real-time. The static maze is ideal for precomputed indirect lighting, baked into the lightmaps [AMMH08](pages 339, 411).

Figure 3.3: Benchmark maze with various materials and precomputed indirect lighting. Screenshot from the Unity Benchmark version.

3.3 Pac-Man Movement

The original Pac-Man does not move in a physically correct way. He moves at a constant speed, begins to move immediately, stops immediately and does not have to slow down to take a turn. This behavior is part of the gameplay. It would be simple to implement without physics component. To take the physics into account, I have created a system of collision detectors and forces (Figure 3.4) to make the Pac-Man move right in the physically correct environment.

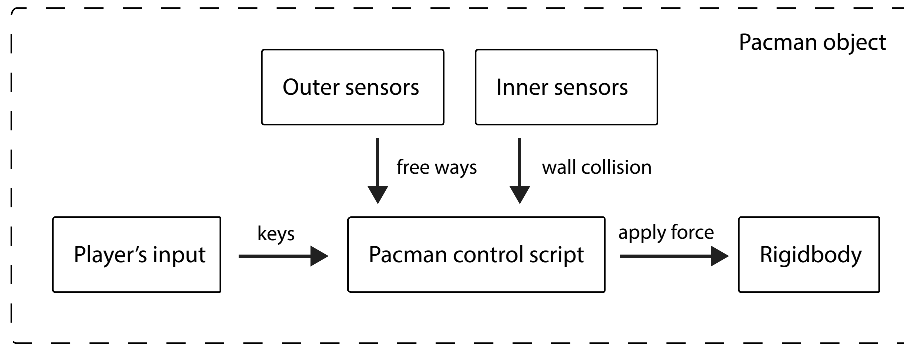


Figure 3.4: Components of the Pac-Man control system.

The basic characteristic of the Pac-Man's controls is, that he does not stop in the middle of the corridor. If the key is pressed, he continues that way, until he runs into a wall. According to the key's map, the control sript determines the direction of the force, that keeps pushing Pac-Man to move at a constant speed. When he bumps into a wall, the detectors notice that situation.

To avoid pulsing in speed, when the top speed is reached, the script only adds the force needed to achieve the target speed, based on the formula 3.1. Where $vmax$ is maximum speed, v is current speed and t is the frame time.

$$F = m * \frac{vmax - v}{\Delta t} \tag{3.1}$$

To deal with the unrealistic way of the Pac-Man's turning, I have used sensors to detect free ways to turn. The control script evaluates the input data, and if the way is free and turn key pressed, it touches the internal physics vector of the rigidbody's velocity and modifies it's direction (See Figure 3.5). This is not physically correct, but it leads to the desired behavior.

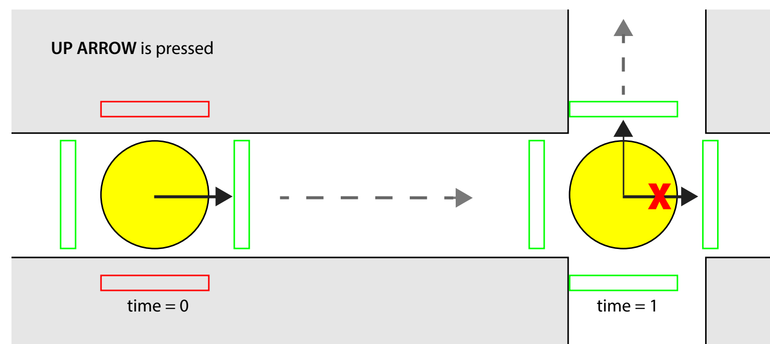


Figure 3.5: Decisions during turn on a cross.

3.4 AI characters

Characters in the game are always standing on one tile. However, their body is approximately 2x2 tiles large, so it fits in the corridor.

In the game, there are four ghosts (Figure 3.6). Each one of them has a different personality. The character of the ghost determines the way he chooses his target. They only make decisions when they enter a new tile. They never change direction to go back to the tile they came from. They choose the next tile per distance to their target (each ghost has a different target), the lowest distance wins. Ghosts do not collide with each other.

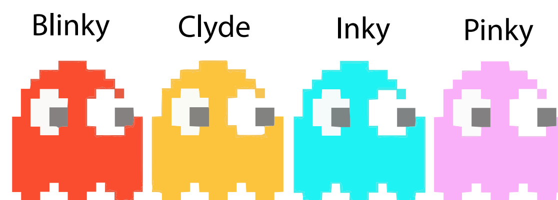


Figure 3.6: There are four ghosts in the game.

The red ghost Blinky goes after the Pac-Man. The Pac-Man character itself is his target.

Pac-man has a point called bait, which is always 5 tiles in front of him. This bait is the target of the pink ghost Pinky. If Pac-Man is heading up, the bait moves another 5 tiles left.

The blue ghost is called Inky. There is another bait, let's call it $bait_2$. It acts as the first bait, but it is just 3 tiles far. There is an imaginary line between ghost Blinky and Inky's target, which moves so that the $bait_2$ is always in the middle of the line.

Ghost Clyde, the orange one, has Pac-Man as his target. However, when he approaches Pac-Man to a distance of 10 tiles, he changes his mind and switches the target to the left corner. When the distance to Pac-Man is above 10 again, the target is switched back.

The ghosts do not calculate the optimal way to their targets but decide on each tile instead. Therefore, I could not use the navigation system of the AI game engine component. Instead, we have implemented custom scripts to control ghost's behavior. However, I do use the AI component to physically move the ghost's rigidbody from one tile to another.

Ghosts always move according to one of the movement modes: Chase, Scatter, Frightened. The game is most of the time in the Chase mode state.

In this mode, ghosts are pursuing their targets. However, the Chase mode is not active all the time. The game uses a timer, which changes the Chase and the Scatter mode [Bir].

In the Scatter mode ghosts forget about their targets, and each one chooses a target in one corner of the maze. Switching between the Scatter and the Chase modes creates a wave effect. So, it seems that ghosts attack Pac-Man and after sometime lose interest, then attack again. This makes the game more interesting to play.
The last movement mode Frightened is activated whenever the Pac-Man eats an energizer. Ghosts change color, slow down and randomly decide which way to go. This behavior creates the illusion of trying to run away from the Pac-Man.

3.5 Maze generator

Figure 3.7 shows the Original Pac-Man maze divided into logical tiles. It is the same maze in every level. In my implementation of Pac-Man, I have prepared a way to generate various maze models. Maze generator creates a model by analyzing the input text file and populating the right models on their locations. It has to be done in the editor, before compiling the executable game. The reason for this is baked light. To achieve the best visual results, I had to bake the indirect lighting into lightmaps. I have not found a way in Unity, to run this process inside the final executable game. Because the maze generation does not affect the real-time performance and therefore plays no role in the comparison, I have implemented just the Unity version. Still it is possible to move the newly generated maze into Unreal through the .obj format.

There are several rules for the maze, to be valid Pac-Man maze. The maze is 28x31 tiles large, symmetrical about the vertical axes. It has a prison in the middle. There should be no dead ends. Pac-Man always has another way to leave corridor. The walls are at least two tiles thick, which implies there are no sharp turns. In the middle part, there are two teleports on the sides. Any maze with these parameters can be considered Pac-Man maze. I have chosen the input to generate the original maze for the game, but it is not necessary. To create any maze in Unity editor we need to follow six steps:

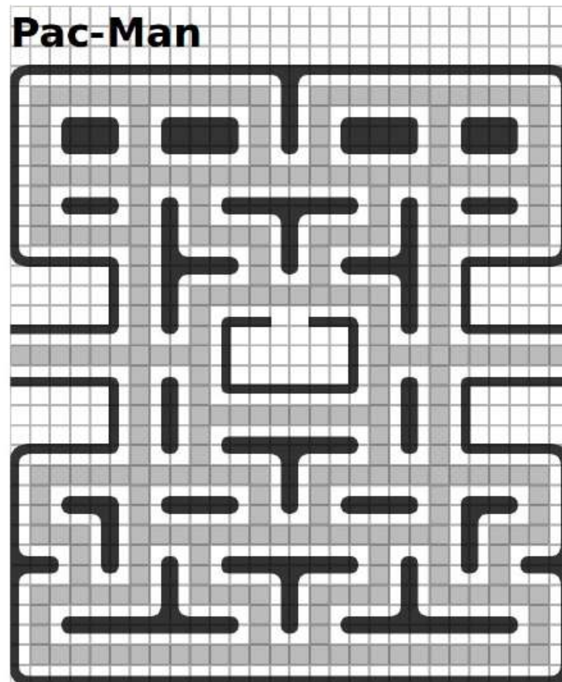


Figure 3.7: The grid that shows 28x31 tiles creating the original Pac-Man Maze.

1. Modify input file map.txt in the root folder
2. Remove the old maze from game hierarchy
3. Check Instantiate Mesh Walls in the MazeCreator component
4. Run the game and copy paste the newly generated walls into the scene
5. Set the walls Solid; lighting will then rebuild automatically
6. Regenerate the NavMesh (Navigation, Bake)

In the following paragraphs I will describe how exactly the generator works. In the beginning, the script loads the map from txt file and saves it into a two-dimensional integer array. It makes the new map one tile larger on each side and writes walls in there. The map syntax is simple: one means wall and zero means free corridor. The script continues by evaluating this array.

On each tile position, we assign a different number, according to the wall type, which fits there. The method `evaluateBrickType` counts how many neighbors of the file are walls. Then there is a big switch that decides about the rotation of the brick.

Let's go through one example, which is displayed in Fig.3.8. We have to evaluate the brick type of the tile. We count there are three neighbor walls. That means our tile mush be corner brick. So, we look at the four corner positions, to know how the corner is rotated. The top left corner is wall, which means the wall is a southeast corner.

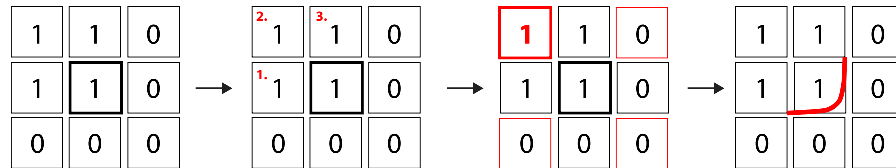


Figure 3.8: Tile evaluation diagram describes the process of choosing the right tile type to be instantiated.

The reason why I have made the map larger on each side is clear now. There is a condition, that every wall has to be at least two tiles thick, this would not be true for the border walls. So, we make the border walls two tiles thick to keep the algorithm working and then simply remove the additional walls.

When the map is evaluated, we just need to instantiate the right objects on places. In the hierarchy, under maze/wallTypePrefabs there are object prefabs to create the maze. These can be modified as well. The script instantiates walls according to the evaluated map and food instances in the corridors. This happens at the beginning of the game. Walls can remain the same, but the food is regenerated in every level.

3.6 Scaling the problem

The goal of this bachelor project is to create a benchmark for multiple gaming platforms: Gaming PC, laptop, Android phone and VR. These platforms differ in controls as well as in graphical performance. I have defined three configurations (see Table 3.1), to match the targeted platforms. Q++ are used for PC and notebook, Q+ for mobile and the light version Q- for GearVR.

To scale the problem and create various versions I had to modify some of the game components. For VR deploy, I use fast mobile shaders with baked light. However, on the PC version, I have chosen physically based shaders, together with real-time direct and indirect lighting, HDRI sky based global illumination, reflection probes and other advanced techniques provided by the game engines. To make the calculation, even more, performance heavy for the gaming PC, I have duplicated the maze up to seven times and created autonomous mazes where ghosts move independently.

	Q++	Q+	Q-
models	full	full	simplified
maze instances	1 - 7	1	1 lowpoly
shaders	PBR	PBR	mobile
realtime light	yes	yes	no
baked light	yes	yes	yes
reflect. probes	yes	yes	no
SSAO	yes	no	no
motion blur	yes	no	no
antialias	FXAA2	no	4x

Table 3.1: The platform features overview.

In the final compare test, I have configured the game to look as similar as possible on both Unity and Unreal engines. Most of the parameters can be measured and configured. Theoretically, the games should look identical.

3.7 VR adjustments

I have been porting the benchmark to VR as well. I have chosen the GearVR platform with Samsung S6 phone because unlike Google Cardboard it has a clear hardware specification while it still remains reasonably affordable VR solution. Moreover, if we compare the serious VR headsets (not taking cardboard in account) GearVR is the best selling VR solution in 2016 [Kor].

With the screen-space effects turned out Samsung can run the Unity Benchmark version with framerate over 20 frames per second, which is impressive for a phone, however still too low for fluent VR experience. That is why I had to rebuild the environment; use less vertexes, bake all the lighting and use mobile unlit shaders according to the Unity VR optimization guide [VRo]. In the end of the process, there are three 4K textures with baked lighting to cover the maze. The large textures are no problem for the phone; it has enough memory. There are only 33K triangles. I have baked the lighting setup in Blender Cycles render engine. There is no doubt the game looks much worse as we can compare looking at the Fig. 3.9, but it runs on mobile as VR in real-time.

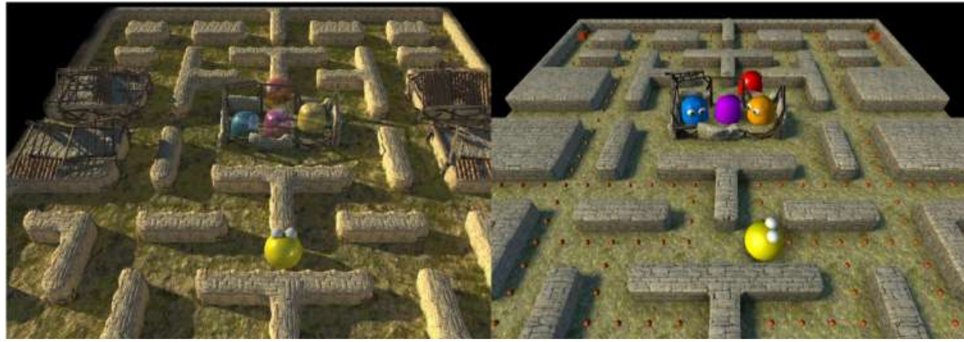


Figure 3.9: The Unity Editor environment.

The last thing was mapping the controls to GearVR touchpad. The touchapad recognizes swipes, but compared to keys swipe can not remain pressed during a time interval. Moreover there can be only one swipe active at a time while we can hold multiple keys pressed at once. To overcome this difference I have decided that swipe means pressing the key and releasing the other keys. The swipe key remains pressed until the next swipe. The gaming experience is not the best, but it is controllable and for the purposes of a benchmark it serves as a sufficient solution.

Chapter 4

Unity implementation

The first part of the benchmark is implementation in Unity 3D engine. Unity Engine is one of the industry standards in game development. It is a component based multi-platform solution, it is easy to learn, and it has large developers base [Wat11]. I have been using version 5.3.3. This engine relatively easy to learn. On the other side, it is a bit like a black box. It does not allow the user to go too deep into the settings and overall, we do not know how exactly are the functions implemented. I have been writing scripts in C#, because unlike JavaScript or Boo it is similar to Java. I have built 3D models in Blender 3D version 2.7 and exported them into Unity through .fbx format. Unity has support for direct import of .blend files, but it seems to have some unresolved problems since Blender 2.71 and Unity 5. I have used free PBR textures from http://textures.com.

There have not been any major problems during the Unity development. The whole Unity environment is user-friendly as Fig. 4.1 shows. Unity documentation is detailed and easy to search. I had no problem to find all the answers on their support forums. Obviously, there is large Unity developers base.

4.1 Game architecture and components

Unity is a component-based game engine. The basic entity in the scene is a game object. The game object itself does not have any functionality. It serves as a container for the components. There is no hierarchy of the components, they are stored in a list inside the game object. Each component takes care of single functionality. Usually, the primary components are Transform (where the game object is located in the scene), Mesh (the visual

Figure 4.1: The Unity Editor environment.

representation of the object), Mesh Renderer (how the object is drawn in the scene), Material, Animator, Rigidbody, Script and many others. One of the components can be a script. The scripts are written in C# or Javascript. Scripts take care of the functionality of the object. They can reference each other, change the game objects components or react to the player's actions. Obe game object can contain many scripts for different kinds of behavior.

Another important concept in Unity is Prefab asset. Let's say we configure a few game objects, add certain components, set them up, so they work together and create one unit. Then a Prefab can be created from them. The settings of the components and their relations are stored inside the Prefab enabling us to spawn an exact copy of this setup in the runtime. In this way, we can prepare complicated structures and reuse them in the game multiple times.

The Pac-Man Benchmark uses the Prefabs for ghosts, biscuits, energizers or walls. On the top of the game object hierarchy, there is a maze object. This object does not have any visual interpretation, but it contains important scripts for the maze logic. There is a script for the creation of a maze based on the text file, then Navigation controller which takes care about the ghost pathfinding logic and finally there is a Game Controller script which stores the game state variables. Because of the object hierarchy, characters inside the maze have an access to this maze object and communicate with it. The maze object is equivalent to the Unreal level blueprint with one exception. In the versions of the benchmark with more mazes, there are actually more maze objects, while in Unreal the level blueprint is a singleton.

4.2 Ghost navigation

Unity has navigation system implemented as part of the engine. We can control AI behavior easily through the API calls. There is an automatic NavMesh generation, which constructs mesh for AI navigation from static objects in the scene. Unity has NavAgent component, which controls the character's movement. The only problem with using this system directly was that it was too clever and ghosts in the original Pac-Man acts a bit different. The game would be unplayable if all the ghosts just plan the optimal trajectories to catch Pac-Man. They act in a simpler way that I have described in 3.4. However, to test the Unity navigation system, I could not just throw the NavMesh away. Apart from finding the way, navigation system solves one more important issue. It applies forces to the characters to move them from place to place. I wanted to use at least this feature. So, the final implementation has a custom navigational system based on the original Pac-Man. However, the final ghost movement is performed by the NavAgents.

The navigation system has one central component (Navigation Controller it is shown in Fig. 4.2) which makes decisions about the ghost's target locations, switches the movement modes, etc. This component is connected to the maze object itself, and it closely cooperates with the Game Controller component, which manages the game's state. Ghosts just have simple ghost Navigation Script, which is parametrized for each ghost type. The script watches ghost's position, and when ghost reaches new tile, it makes a call to the central navigation component, to ask for new coordinates.

This centralized architecture gives up better overview and management of the ghost movement states, compared to separate intelligent ghost scripts. In this way, the ghost is in a position of a terminal to the server. It just does, what it is said to and does not contain almost any logic.

4.3 Visuals

Considering visuals Unity has been behind other game engines since it's first release. The Unity 5 version as made a great step forward introducing the Unity Standard shader, physically based shading and image-based lighting techniques, however in comparison to Unreal or Cry Engine Unity still has some deficits. It is obvious that the best-looking graphics is not the main goal of the Unity engine. Unity has become the world's most used game engine solution because it is availability and ease of use. This is a great example

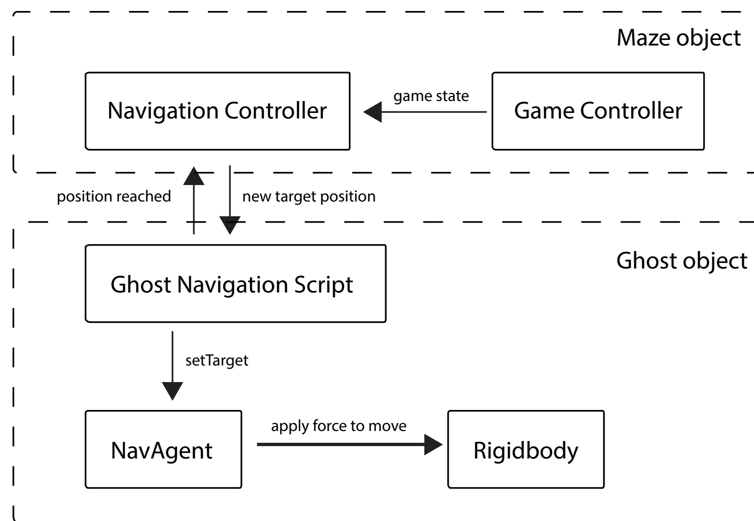


Figure 4.2: Ghost navigation system in Unity.

showing us that the visuals are not always the most important aspect of the game development.

4.3.1 Models and animations

I have made the models in open source software Blender. I have exported them as .fbx files and imported into Unity. To animate ghost's waving, I have used Blender Shape keys (Fig. 4.3) and animated them in as Blend-Shapes inside Unity Editor.

Shape keys are commonly used for facial animations. The model has a base shape key, which holds the default state. Every other shape key represents a modification of the vertex positions. The position is relative to the position in the base shape key. This allows us to combine multiple shape keys together and animate the percentage of use of particular shape key. The workflow turned out successful, so I used it for animating Pac-Man as well.

4.3.2 Materials

Materials are a crucial part of the graphical presentation. Realistic materials and textures are more important than the quality of the 3D models. Unity Standard Shader is a built-in implementation of Physically Based Shading (PBR). In real-time graphics, this is quite a recent concept which empathizes

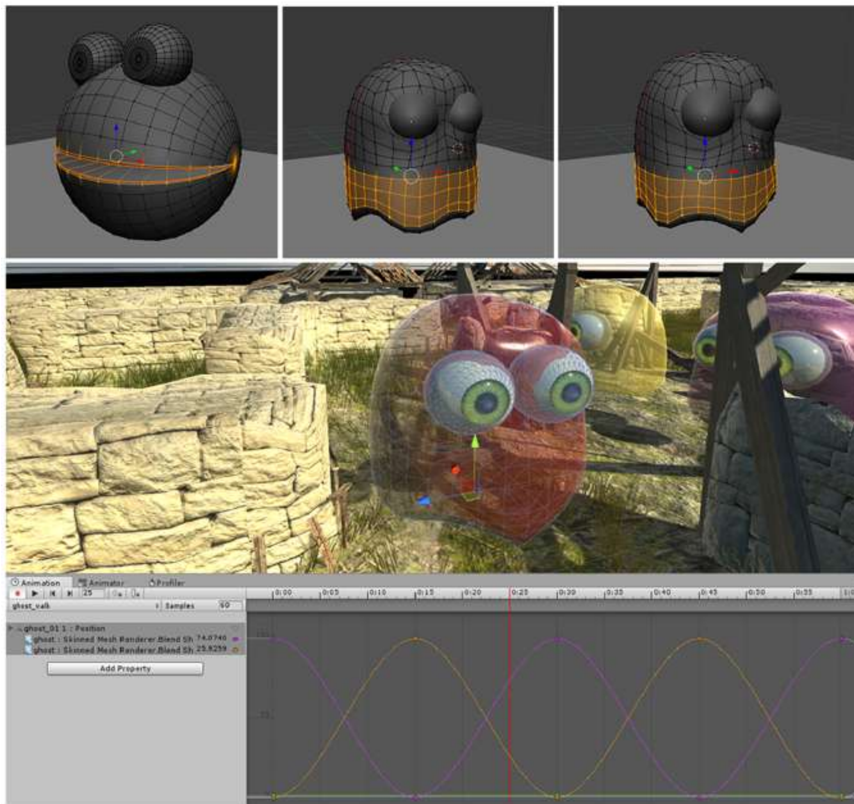


Figure 4.3: Animation using shape keys inside the Unity Editor.

realistic material behavior. In contrast to Phong Shading Model, the artist does not configure materials visual parameters, but it is physical properties such as Metallic or Roughness (Fig. 4.4).

Albedo defines the material color. Metallic says whether the material is metal or diffuse one, such as ground. Smoothness specifies how shiny the reflections are. Whether it is clear reflection like a mirror or a blurred one. There is a possibility to define Normal Map for bump mapping, Height map, Occlusion map, and Emission. These are the basic PBR parameters. There is an option to change Rendering mode, from default Opaque to Cutout (binary alpha channel), Fade, or Transparent.

Most of the materials in Pac-Man Benchmark are Standard Shader Opaque. Ghosts use the Transparent render mode. They have to be rendered on top of the grass. If we let Unity decide automatically based on the camera distance, the grass suddenly pops in front of the ghost while he is going above it.

To test one of the latest features in OpenGL 4, or DirectX 11 I could not forget the parallax mapping. This technology dynamically changes the number of vertices in the object. Based on the high map it generates real

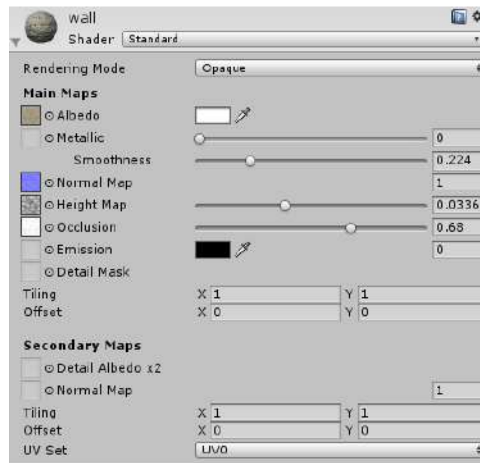


Figure 4.4: Animation using shape keys inside the Unity Editor.

Figure 4.5: Unity Tesselation Shader used on the roof, creating real bump according to the height map.

bumps, a new structure on top of the existing object. I have used Tesselation Shader on the roofs (Fig. 4.5) to achieve this effect. The only tricky part was that Unity takes the height map from alpha channel of the height map, which might be unexpected. For grass straws, I have used a mobile shader. It is fast and allows us to render a large number of transparent objects at once. Grass has three different material colors, to add variety.

4.3.3 Lighting

The lighting setup is very simple. There is one directional light as the sun, with soft shadows and slightly yellowish light color. Then the scene is lightened up by sky. The sky is spherical high dynamic range image; Unity uses it as an ambient light source. An interesting feature in Unity's lighting is the ability to precompute real-time global illumination. It saves lightmap with indirect light intensity and its direction. In real-time, the indirect component depends on the direction of the light. Unity does not need to rebake the lighting when we change the intensity, color, position or direction of the lights whitch is very useful while tweaking the lighting setup. This real-time global illumination solution is called Enlighten [Geo] and it is developed by company Geometrics.

4.3.4 User Interface

The User interface in Unity is a bit difficult to handle. Unity has a Canvas game object that has a Canvas component attached. The UI elements are supposed to be the children of this game object. The Canvas elements are shown as 2D. However, the canvas object is part of the 3D scene for some reason. It gets a little confusing. The UI elements can interact with other scripts; they can be animated. The problem is that there is no central place to manage the UI. Therefore, the UI development is not as intuitive as the rest of Unity Editor.

After hours of trying to get the elements positioned right, I have used the Immediate Mode GUI (IMGUI) feature that is totally separate from the rest of the GUI. IMGUI is code-driven interface intended for programmers. The elements are not objects in the scene, but Unity generates them procedurally from the script code. The setup was much more intuitive this way.

4.3.5 Screen-space effects

Concerning screen-space effect, I decided to use just the typical representatives, that are common for most of the AAA games. I have used motion blur, anti-aliasing script and screen space ambient occlusion They work very well on the computer, but they were unusable on other platforms. The settings are shown in Fig. 4.6. The mobile was just not capable of handling them; framerate has dropped from 20 to 4 frames per second.

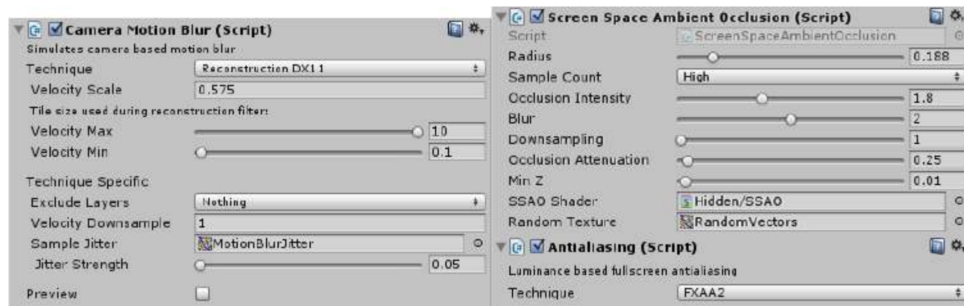


Figure 4.6: Screen- space effects setup. Motion blur, Antialiasing and ambient occlusion

4.4 Profiling

Unity has own profiling system called Profiler. It offers analysis of the processor time as well as the memory, physics, and graphics. The only problem is that it can run only in the Unity editor. Otherwise, the data are unavailable.

For benchmark purposes, I have measured frame time and saved it into a file. The frame time is one of the few variables available without Unity Editor. We can read this one from Time.deltaTime, which tells us the time it took to prepare the last frame. It is commonly used to make some variable real time dependent instead of framerate dependent. (Player with higher framerate should not have faster gameplay) There is an option to connect profiler to an instance of the game on the PC, but no way to profile the mobile instance (ideally over Wi-Fi, because of the VR).

There is a way to save and load Profilers data, for later analysis. However, the data are saved in byte form and are readable again only in the Profiler. The script saves 300 frame samples into files, which we can load in. Still, the game has to run inside the Unity Editor for this feature to work. We can also save profiler data into text files from our scripts. The class `UnityEditor.UnityStats` is not documented, but the API is opened to read from.

Chapter 5

Unreal implementation

To implement the second version I have used Unreal Engine 4.15. Unreal is a complex development system. The working environment is called Unreal Editor (See Fig. 5.1). It can deploy apps to almost any platform, Unreal supports PC, gaming consoles, mobile devices, VR platforms, smart TVs. Unlike Unity, it is targeted on large projects. Unreal offers solutions for the development of large-scale environments and massive multiplayer games. They expect the developers to aim the latest hardware. Therefore, the system requirements are high for the development as well as for the final runnable games. Unreal is considered the leading platform in photo-realistic rendering and architectural visualization.

Unreal Engine has opened C++ source code. It allows the developers to write the pure C++ and have greater control over the engine's actions. On the other hand, Unreal has an option for non-coders as well. It is called The Blueprints. Moreover, it is essentially the node-based editor that allows users to implement logic without the knowledge of coding. Every node has a certain function; it has inputs and outputs. The nodes are connected and create the logic. The Blueprints can contain functions, macros, custom events or event dispatchers. They have variables, data types, arrays, enums and hash maps. In Unreal most of the components use blueprints. The game designers build the game logic with Blueprints, the artists setup the materials as blueprints, and even the artificial intelligence uses blueprints too. At first I have tried to script the game exclusively in C++. However, the Unreal framework is so complicated, that it just does not make sense to write game logic inside C++. The code is long, complicated and hard to read. Paradoxically the Blueprints were much closer to C# scripting from Unity. An example of a blueprint ensuring the ghost target update is at Fig. 5.2

The game objects in the scene are called Actors. The objects that can be controlled by player or AI are called Pawns. Unreal terminology for such

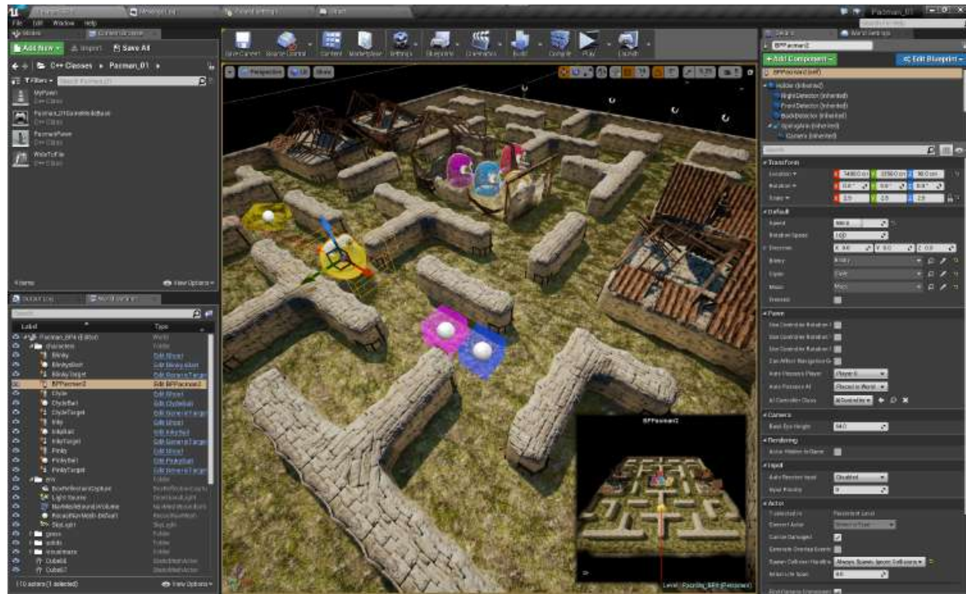


Figure 5.1: The Unreal Editor environment.

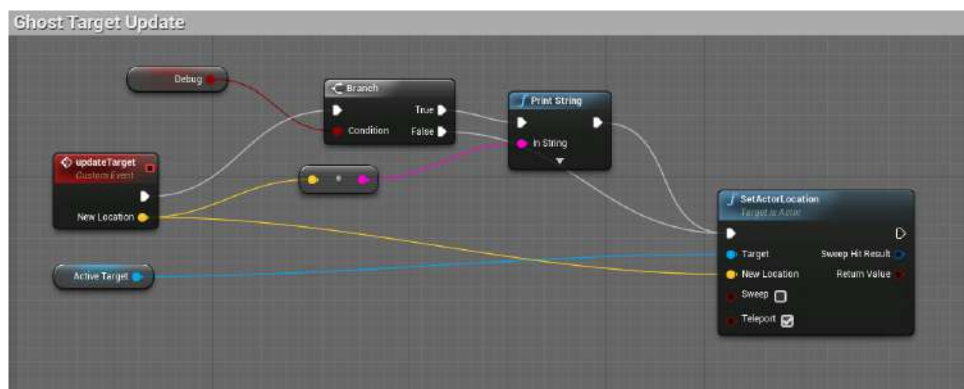


Figure 5.2: An example of simple blueprint for Ghost Target update.

control is that the Pawn *is possessed* by the Player.

Unreal is a component-based engine. The components inside a game object are stored in a hierarchy. The root of the hierarchy is usually a blueprint which contains the logic. Inside the hierarchy, there are the components - functional parts of the object itself. For example, the mesh, triggers, colliders, or even a camera. This is very different from Unity (Unity does not implement any component hierarchy just game object which contains list of components where the control script is usually one of them.)

5.1 Game architecture and components

In Unreal Engine, every scene has a level blueprint. This blueprint is running through the whole level, and unlike other blueprints, it has access to all the objects inside the scene. It is ideal to place the top level game management here. In our case, the level blueprint contains all the settings at the beginning of the game. It solves the Pac-Man-ghost collisions (based on common tile, not an actual component collision) and it manages the game state and player's score.

The level blueprint can be very useful for one more thing. In the scene, there might be many objects that need to interact with each other. However, we do not want them to reference each other directly, that would lead to spaghetti code. Moreover, some of the objects may not be in the scene the whole play time. The connections may appear or be broken after some time. It is a good to maintain low coupling.

Blueprints can have custom events. We can call these events from the level blueprint. Additionally, blueprints have event dispatchers which act like a trigger informing that some action has happened on the current blueprint. With this setup, one object can fire an event dispatcher. The level blueprint listens on this action and reacts to it lets say by calling event on some other objects. The original object does not know about the following actions, and the final objects do not know where the action has come from. They are not connected to each other directly. The connection is set up in the level blueprint. This indirect invocation is shown in Fig. 5.13. I have used this method multiple times especially in the context of Pac-Man-ghost collision. Each character fires event dispatcher whenever it reaches new tile. The level blueprint evaluates the collisions and the subsequent actions based on the game state.

The setting of the game objects in Pac-Man Benchmark is nothing complicated. As we can see in the Fig. 5.4. Pac-Man is possessed by the player controller and receives the input keys. Pac-Man blueprint has detectors of walls and free ways as I have described them in Section 3.3. The blueprint informs level blueprint about the new tiles, biscuits, and energizers that the Pac-Man has eaten and about the new ghost bait positions. The level blueprint process these information and reacts accordingly by changing the game state or invoking characters events. The ghosts are possessed by the AI controllers and communicate with the level blueprint in a similar way that Pac-Man does.

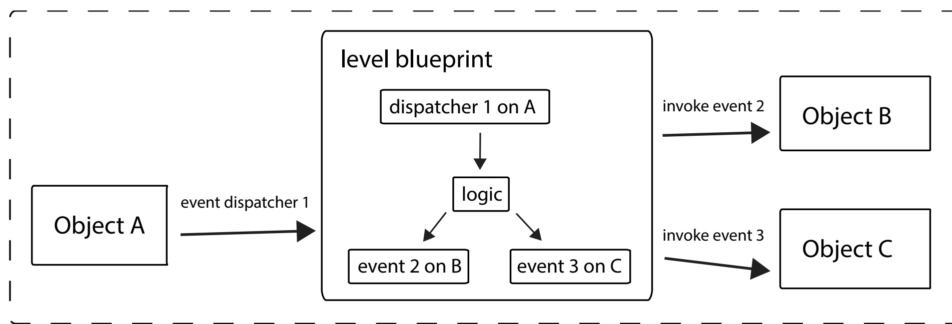


Figure 5.3: Indirect event invocation using the level blueprint.

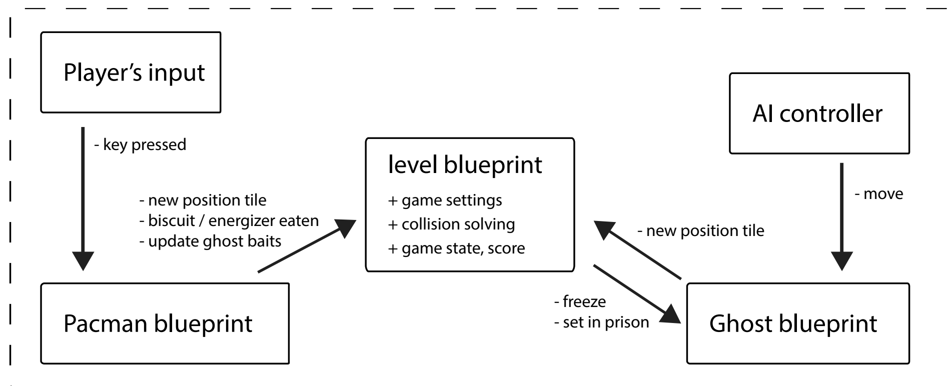


Figure 5.4: Diagram of the main components in the Unreal Pac-Man Benchmark.

5.2 Ghost navigation

The AI itself is much more complicated in Unreal compared to Unity. The basic structure is illustrated in Fig.5.5 The Character class is a child of Pawn suited for AI characters. It is possessed by Character controller. Behavioral trees are process diagrams that describe the decision process of the character. Black board is a set of variables determining the characters state. Behavioral tree references a copy of the Blackboard and makes decisions based on the values of the blackboard variables. Character controller then runs the behavioral tree, uses and sets the variables of the blackboard accordingly to the situation. The controller also uses NavMesh - navigation surface generated based on the static meshes in the scene. Characters can walk on the mesh and find paths on it.

In order to simulate the original Pac-Man ghosts, I had to change the AI system from the basic setup. Ghosts decide which way to choose when they enter the new tile. They measure the distance to their bait and choose a direction. In the center of the next tile in the chosen direction there is the position of a ghost Target object. Target is an Actor that the ghost follows all the time and it moves from tile to tile, depending on the direction chosen by

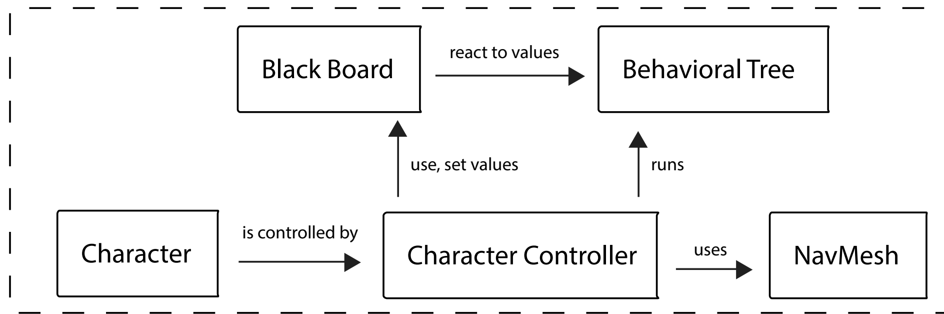


Figure 5.5: The basic Artificial Intelligence setup in Unreal Engine.

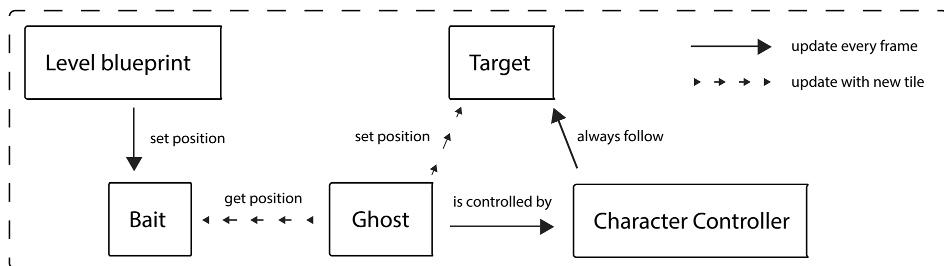


Figure 5.6: The ghost control setup in Unreal Pac-Man Benchmark.

the ghost. With this setup, the level blueprint only changes the ghost baits and each ghost then set his target accordingly. The relationships between the objects are shown in the Fig. 5.6.

5.3 Visuals

Unreal engine is famous for its visuals. It allows the user to build his materials with nodes (basically shader programming with a visual interface). The lights look great and can be real-time or baked. The best of all are the inbuilt post-process effects, which make the difference. Unreal engine is widely used for architectural visualizations in the form of real-time walk-throughs. The developer does not need any other plugins to make the game look as good as today's AAA titles. However, all these visuals are performance heavy matter and require appropriate hardware.

5.3.1 Models and animations

The unreal engine does not support such a wide spectrum of formats as Unity does, but it can import the most used ones. For 3D the supported formats are .obj and .fbx. Both of them are supported in Blender as well. So getting

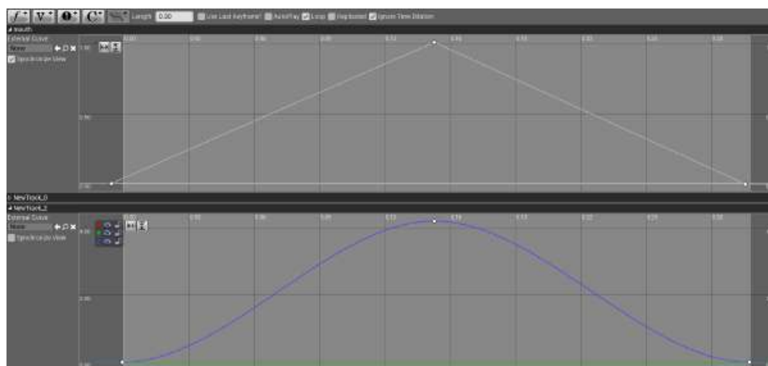


Figure 5.7: The animation curves inside The Unreal Editor.

the assets from Blender to UE4 was no problem at all I have used the .fbx format. The UV maps have migrated correctly as well. I only needed to set the coordinates system right. Unreal uses centimeters as the base unit and the coordinates are rotated 90 degrees clockwise around the Z axes. The translation Blender → Unreal therefore is X → Y; Y → -X; Z → Z.

The animation process was rather similar to Unity. I have exported the character animations (ghost waving and Pac-Man mouth opening) as Shape Keys and imported them as Morph Targets to Unreal. This process only worked with FBX 6.1 ASCII version and Apply Modifier deselected. The modifiers inside Blender must be applied separately before exporting the model. Animating and timing the translations between poses of the morph targets was simple animation with curves inside the editor (Fig. 5.7) very similar to the one in Unity.

5.3.2 Materials

Material system is one of the best features that Unreal offers. It works as a node-based system for creation of materials (Fig. 5.8). For each material, Unreal compiles the shader. We can use the material as it is or change with parameters to create different variations of the same material called material Instances. This method gives the artist powerful tool for creating stunning materials. The downside is that the compile times of more complicated shaders might be higher (up to minutes).

For walls, ground, and wood I have used a simple Lit material with three textures for Color, Roughness, and the Bump map. The result was very similar to Unity. The artist can add parallax mapping to any material with any settings which is great. (Unity has parallax shader with limited settings considering roughness and reflections) I have noticed the largest difference while creating the ghosts material. Unreal supports light refraction

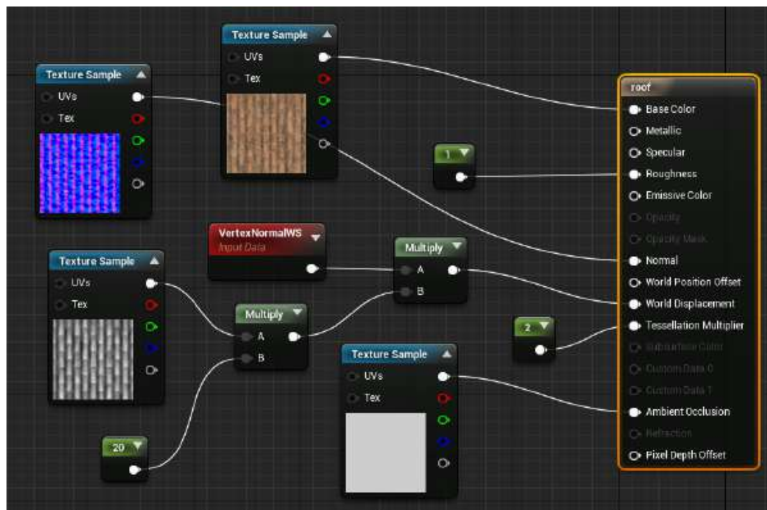


Figure 5.8: An example of a parallax material of the roof.

Figure 5.9: The ghost reflective material.

and screen space reflections. We can combine these features with Fresnel effect [AMMH08](pages 231-236) to achieve better-looking results. If we look at face from the vector of its normal, it seems more transparent while when we lower the view angle close to zero, the surface gets more reflective. That is how the ghosts material (Fig. 5.9) appears.

The only complicated material was the grass. Grass has to be transparent, for the purposes of this benchmark it can be unlit, but the shader should be as fast as possible. In Unity version, I have used the mobile transparent shader, and it worked well. In Unreal the situation was not that easy, I had to construct the material by hand. After experiments with dithered masked alpha, I have set regular unlit material with two-sided transparency. This material fades out with the distance, so the far away grass does not seem super green while the close tufts are still visible.

5.3.3 Lighting

The scene has simple light setup. It is lit by one directional light as the sun that casts real-time shadows and then one environmental light to lit the scene with the HDR sky. The setup and baking are slightly faster than in Unity because the editor allows us to set up the baking quality. Still, we must rebuild the lighting after every change. (Unity does not need to rebuild lighting after the light adjustments thanks to Enlighten [Geo]. This solution is available for Unreal as well, however it is monetized.) I should mention that Unreal also supports the IES light profiles [Dra] even though I have not used them in the benchmark.

Unreal engine implements an optimization of shadows called Cascade Shadows [AMMH08](page 358). It renders multiple shadow maps based on the camera distance so that the objects close to the camera have sharper shadows compared to the objects far away. There is also nice blend between the shadow maps, so the map change is almost not noticeable, This technique helps Unreal to boost the performance, and it is very efficient especially for the larger scenes (the setup with seven maze instances).

5.3.4 User Interface

For the creation of user interface Unreal has a very powerful, yet simple tool: Unreal Motion Graphics UI Designer (UMG) (Fig. 5.10). The core of this tool are Widgets - pre-made functions that we use to create pieces of the interface such as buttons, progress bars, text boxes, check boxes... The UMG consists of two tabs: The Designer and The Graph tab. In Designer tab, we can set up the composition and appearance of the UI elements. The Graph tab contains widget blueprint that controls the functionality of the UI. The UMG provides a much more efficient way of UI development than the Unity tools.

5.3.5 Screen-space effects

As I have mentioned before, the Unreal screen-space effects [Doc] are on the top industry standard. These effects are very easy to setup from the menu, and the results are visually stunning. For Pac-Man Benchmark I have used only Anti-Aliasing, Motion Blur and Ambient Occlusion to match the Unity setup. However, it is worth to mention the rest as well because together they have a great impact on the final result.

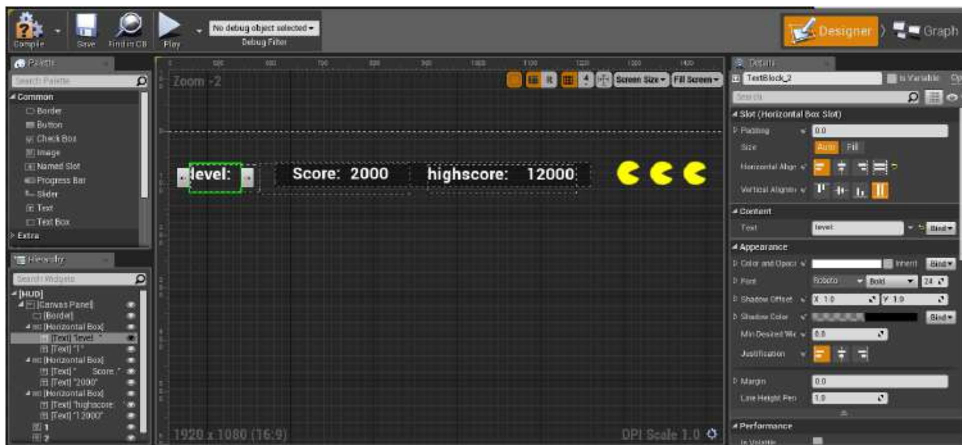


Figure 5.10: The Unreal Motion Graphics UI Designer interface.

Figure 5.11: Chromatic aberration example, image courtesy Epic Games, Inc.

The vignette effect darkens the edges of the screen; this happens in photography because in the center of the lens there is more light, then on the edges. The effect is usually visible at low f-numbers (f/1.4). Another real lens effect is a chromatic aberration (Fig. 5.11). When the light is going through the lens, each wavelength refracts in a slightly different angle which causes the colors to split on the edges of the image. Then Unreal can simulate lens flares, bloom and depth of field, all of them are greatly customizable. An interesting effect is eye adaptation which changes the scene exposure dynamically. When the player looks into the dark, the image gets brighter, and when the player reaches a very shiny scenery, the exposure decreases accordingly. This effect acts as a simulation of a human pupil movement. Lastly Unreal supports color grading. The artists can define the function of mapping the High Dynamic Range render to the low dynamic range screen. For further color refinement, there is an option of loading a color grading LUT texture [Fri, AMMH08](page 474) or grading the image manually with sliders. Three different LUTs and the graded results are shown in the Fig. 5.12.

Figure 5.12: Look Up Tables and the color graded results, image courtesy Epic Games, Inc.

Figure 5.13: GPU Visualizer showing the duration of processes.

5.4 Profiling

Unreal Engine provides developers with various Profiling tools. We can simply show the FPS on screen, or enable more detailed views like stat SceneRendering which allows us to read the draw calls, lights in the scene or number of static/dynamic meshes. The data can be recorded into a file and analyzed later. (Unity needs a plugin to do this) Unreal has two tools for performance data analysis: The GPU Visualizer tells us what amount of a frame time the parts (lights, translucency, shadows, post process) of the GPU process take. The second option is Session Fronted tool which can load the detailed data and display the processes that we pick in time. An important part of performance optimization is also Shader Complexity view. It is one of the view modes that we can choose from (lit, unlit, wireframe...). It shows how difficult is to evaluate certain pixel on a scale of green to red.

Chapter 6

Results and Comparison

Comparing the engines is nothing trivial. Each one of them is different; each one has strong features and weaknesses; each is suited for different project or developer. Here I present four approaches to engine comparison: The table comparison which might serve as a base point for detailed research of the features. Then I present the benchmark results on multiple platforms and different scene scales. After that, I will compare the visuals themselves with the offline Blender cycles render [FS15]. And finally, I will share a subjective opinion from a perspective of a developer.

6.1 Features

At first, I would like to give a brief comparison with a simple table where we can find the most common responsibilities that engines have and a short description of the solutions that certain engine offers. Both engines are powerful and both provide complete pack of tools for game development. Most of the times it is a matter of preference, which tool the developer finds more suitable. However, there are areas, where one of the engines provides apparently better or more complete solution:

1. Unity supports much more target platforms, but Unreal has the most important of them as well. The exact same example is with import formats. Unity supports more of them, but in the end of the day, it does not matter.
2. Considering materials Unreal has material editor which enables the developer to create new shaders. This approach is much more powerful compared to Unity Standard Shader.

3. Unity supports prebaked real-time global illumination, while Unreal does not.

4. Unreal has much better cascade particle system.

5. Unreal offers great inbuild post process effects. Unity can achieve similar results only with external plugins.

6. Unreal provides cinematic tools. Matinee cinematic toolset and sequencer tool allow developers to work with animations and camera in a way of movie production. This features are great for creating in-game cinematics.

7. Unreal has much better and complex solution for terrain and foliage. Similar tools can be purchased and added to Unity with plugins.

Feature	**Unity**	**Unreal**
Platforms		
desktop	Win, OSX, Linux, WebGL	Win, OSX, Linux, HTML5
mobile	Windows Phone, iOS, Android, BlackBerry 10, Tizen	iOS, Android
console	Xbox 360; Xbox One; Wii U; PlayStation 3; PlayStation 4; PlayStation VitaNintendo Switch	PlayStation 4, Xbox One, Nintendo Switch
VR	SteamVR/HTC Vive, Oculus Rift, Google VR/Daydream, Samsung Gear VR, Hololens, Playstation VR	SteamVR/HTC Vive, Oculus Rift, Google VR/Daydream, Samsung Gear VR, Playstation VR, OSVR
TV	AndroidTV, Samsung Smart TV, tvOS	tvOS
Editor	Unity Editor	Unreal Editor, VR Editor for HTC Vive
Special features	scene view and game view as two windows	Possess & Eject, simulate, content browser
Scripting Languages	C#, UnityScript	C++, Blueprint visual scripting
Engine Code	closed	C++ source code available via GitHub

Feature	Unity	Unreal
import formats		
image	psd, jpg, png, gif, bmp, tga, tiff, iff, pict, dds, exr, hdr	psd, jpg, png, exr, dds, exr, hdr, tga, pcx, float, bmp, ies
audio	mp3, ogg, aiff, wav, mod, it, sm3	mp3, wav, wma, wave, snd, caf, cdda, aiff, aif, au
video	mov, avi, asf, mpg, mpeg, mp4	mov, avi, mp4, wmv, aac
3D models	fbx, .dae (Collada), .3ds, .dxf, c4d, jas, lxo, blend, skp	fbx, obj, srt (speedtree)
Rendering	Deferred shading	Deferred shading or Forward shading for VR
Materials	Physically based, Unity Standard shader, tesselation shader, mobile shaders	Physically based, Blueprint material editor, tesselation, layred materials, material instances, lit translucency, subsurf shading model
Lighting	Directional, Point, Spotlight, Area Light	Directional, Point, Spot light, Sky Light, IES light profiles
Shadows	realtime hard/soft shadows	realtime hard/soft shadows, cascade shadows, distance filed shadows
Global illumination	Image-based GI, Precomputed realtime GI, Baked GI	Image-based GI, Baked GI
Reflections / Refractions	Reflection probes	Reflection probes, screen space reflections, Refraction
Particles	simple curves based particle system	Cascade particle system with GPU particles
PostProcess	Effects with additional assets (e.g. Post Processing Stack)	Effects as a part of the engine (AA, Bloom, Color Grading, DoF, Eye Adaptation, Lens Flare, Scene Fringe, Vigenette, Screen space reflections), support for post process materials
Animation	Animation curves editor, Skeletal animations, Blend shapes, animation weights, events at animations, State Machine and transitions	Animation curves editor, Persona animation toolset, state machines, physics-based animation, animation blueprints, morph targets

Feature	Unity	Unreal
Physics	rigidbodies, collisions, joints, cloth, wheel collider, physic materials, PhysX	rigidbodies, collisions, physic materials, APEX integration, PhaT (skeletal mesh physics editor), vehicle physics, PhysX
Cinematic Tools	-	Matinee cinematic toolset, sequencer tool
Terrain & Foliage	simple Terrain Engine, Texture painting, Tree Editor, plugins for world building	Very advanced Landscape and Foliage painting tools inside the editor.
UI	UI objects on the Canvas, IMGUI	Widgets and blueprints
AI	NavMesh, NavAgents, path finding	Behavioral trees, Character controllers, NavMesh, path finding
Optimization	Unity Profiler (CPU, GPU, memory, physics), LOD support, quality presets	GPU/CPU Profiling, Saving statistics to file, Hierarchical LOD, automatic LOD generation, Optimization viewmodes

PC	i7-4770S 3.10 GHz; 16GB RAM; NVidia GeForce GTX 970; Win 7 64bit NR: 1920 x 1200, TR: 1920 x 1200
Notebook	ThinkPad Edge 430; i7-3632QM 2.20 GHz; 16GB RAM; NVidia 635M; Win 7 64bit NR: 1366 x 768, TR: 1920 x 1200
Mobile	Samsung Galaxy S6; Exynos 7420 Octa 2.10 GHz; 3GB RAM; Mali-T760MP8; Android 6.0.1 NR: 1440 x 2560, TR: 1920 x 1080
VR	Samsung GearVR + Samsung Galaxy S6 NR: 1440 x 2560, TR: 1440 x 2560

Table 6.1: Benchmark platforms. NR - native resolution, TR - tested resolution

6.2 Benchmarks

I have designed four tests to measure frame time on each platform. I have automated the tests with the script which simulated user input to create the same conditions multiple times. Table 6.1 contains the test platform specifications.

In the Fig. 6.1 we can see the comparison of the average frame times the benchmark measured. The Unity times start at lower values than the Unreal ones, and they grow approximately 4ms per added maze. This constant growth is independent on the platform (PC/notebook). Unreal times start at 10ms for one maze instance on PC which is two times more than the Unity. However, Unreal times for more maze instances stay stable. They are even slightly lower. Obviously, Unreal is optimized for more powerful machines and more complicated scenes. Each maze has around 1.2 million vertices, so the version 7 has over 8 million vertices. For Unity, this amount means considerable slow down (six times slower than the test with one maze), while for Unreal this makes absolutely no problem.

There are two main reasons for this results. Unity has dynamic draw call batching during the real-time, while Unreal batches the static meshes before compilation. When it comes to larger scenes, Unity can not optimize so many draw calls at once. The second reason for this might be Unreal's cascade shadows [AMMH08] (pages 358-361). Unity supports cascade shadows from version 5.6 released at the end of March 2017, I have used the 5.3.3 version. When we add more mazes and look at them from a greater distance, the shadows far away have lower resolution. If Unity has to calculate all the shadows in high quality, it has to be slower than Unreal.

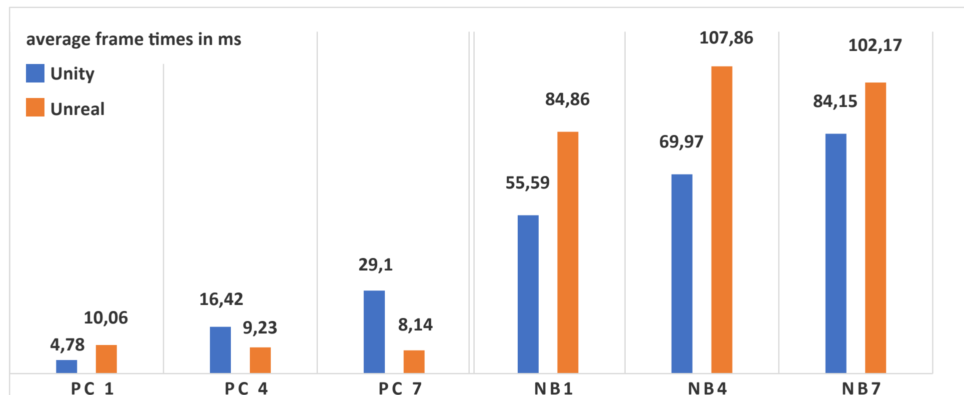


Figure 6.1: Average frame times on PC and notebook. Versions of the benchmark with one, four and seven maze instances. (1.2 - 8.4 million vertices)

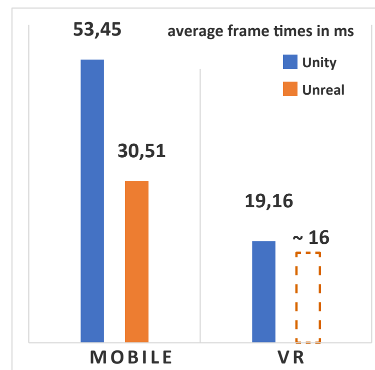


Figure 6.2: Benchmark results on Android and GearVR platforms. Unreal version of the VR was not measured exactly, however from the character of the gameplay I assume the frame time is around 16ms.

On mobile devices, Unreal has better results as we can see in the Fig. 6.2. The benchmark runs faster than Unity, and the game itself looks better (compare for yourself 6.3). The .apk file is larger (Fig. 6.5) and the phone heats up faster. However, the frame time is considerably lower. The simplified VR version has run on Unreal better as well. The gameplay was fluid without any lags, while the Unity version was playable with minor lags. The problem is that I was not able to measure the framerate value correctly, so I can not state it in the graph. Despite that, the anticipated value is around 16ms.

The size of the projection does matter in Unreal but does not play a role in Unity according to the Fig. 6.3. The Unity stays at the same frame time, while Unreal speeds up almost three times. Both engines use deferred shading, but it seems Unreal has better optimization than Unity.

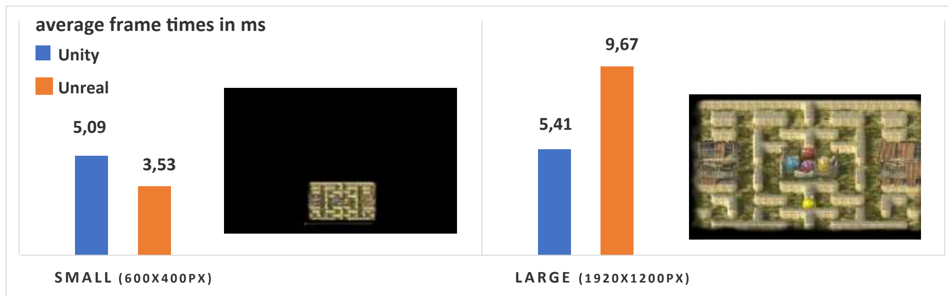


Figure 6.3: The effect of the projection size of one maze to the framerate. Unity frame time stays the same, while Unreal time decreases almost three times.

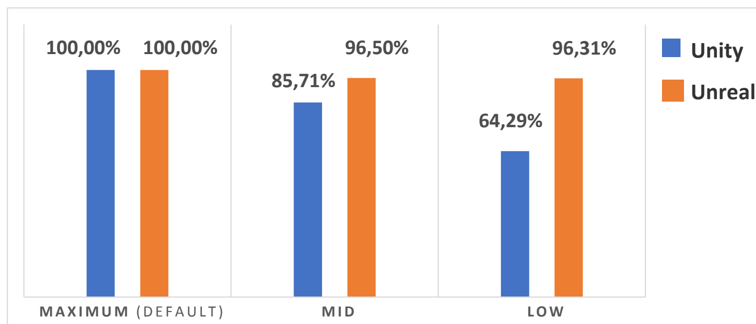


Figure 6.4: Performance speed up with lowering the render settings. Measured with one maze instance on PC.

Unity and Unreal both offers different quality settings. In the Fig. 6.4 I have shown the speed up percentage with lowering the quality settings. I have shown this in percent because this is what would happen during the development. We develop the best looking game graphically, and when the target PC parameters are insufficient, we lower the engine quality settings. Unity has dropped to 64% from maximum to lowest quality, but the low result does not pleasing at all. It is much better to optimize the assets or turn off screen space effects manually inside the Unity Editor than decrease the quality settings in the case when we need higher performance.

In Fig.6.5 there are sizes of the final builds. Considering development for PCs and consoles it is not a big deal, however, on Mobile platforms, space is still precious. The Unreal version of mobile Pac-Man benchmark was three times larger than Unity version. Also, the Unreal project sources are incredibly large, the Pac-Man benchmark had over five GB.

Lastly, I will comment two time-graphs. The first one is Fig.6.6, it is the graph of frame times measured with Unreal benchmark running on a notebook with one instance of the maze. The frame times are changing based on the percentage of the screen that contains the maze. If the Pac-Man is going through a corridor close to the edge of the maze, the maze is getting out of the screen. Therefore the frame times decreases. Also, Pac-Man is

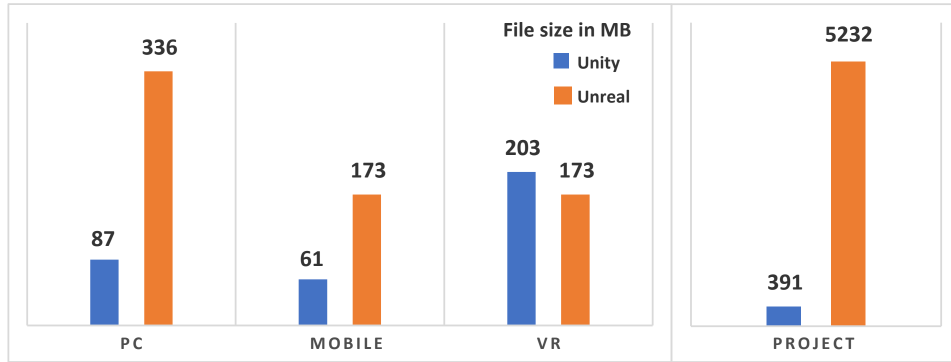


Figure 6.5: Size of the final builds and the game projects in MB.

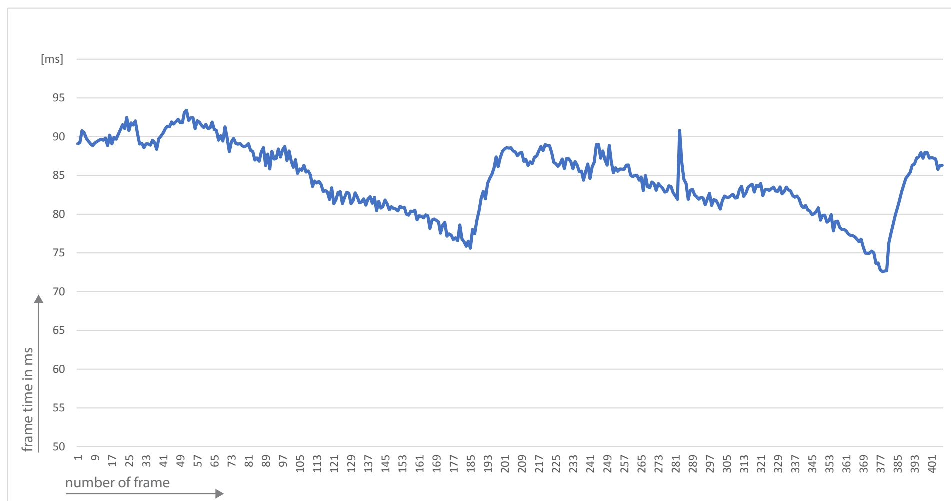


Figure 6.6: Frame time measured with Unreal benchmark running on a notebook with one maze instance. The frame time decreases based on the screen coverage and number of objects in the scene.

eating the biscuits, so the number of objects and draw calls decreases as well. At frame 185 Pac-Man is caught by a ghost, the level restarts and Pac-Man appears in the middle of the maze again.

The second time graph (Fig.6.7) shows frame times measured with Unity benchmark running on a PC with one maze instance. We can say that the frame time remains the stable for most of the time. The garbage collector starts running at frames 175 and 283, slows down the process and creates the peaks in the graph.

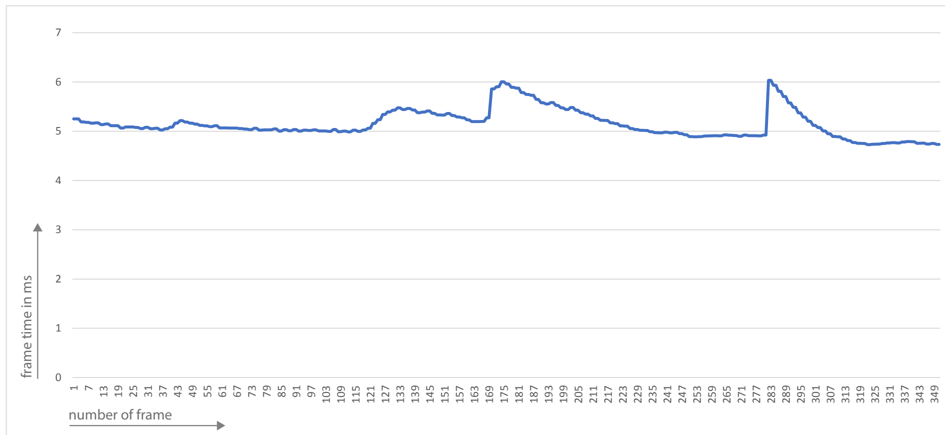


Figure 6.7: Frame time measured with Unity benchmark running on a PC with one maze instance. The frame time decrease peaks are caused by the garbage collector.

Figure 6.8: An example of using Filmic color management in Blender cycles. Image courtesy Blender Guru.

6.3 Visual quality

Visual quality is an important aspect of a game. Players rate the game mostly by gameplay and graphics. In this section, I will show how the benchmark looks like in Unity and Unreal and compare it to the offline render from Blender Cycles [FS15]. The cycles render engine + backward path tracing method (sending rays from camera instead from light sources). I have been rendering on Nvidia GTX970 with one thousand render samples. The render time was between one and three hours per frame. The scene was set up the same as the one in the real benchmark. There was no direct light, I have lit up the scene entirely using the hdri image of the sky.

To achieve even more realistic results I have used Filmic color configuration [Sob]. The Filmic plug-in brings significantly higher dynamic range 6.8, then regular cycles with sRGB color management. This is very important in the context of reflected light. If I have used just the sRGB settings, the white would not be bright enough to produce the sufficient amount of indirect light. Filmic solves this problem. I have then gamma corrected the final image to match the game engine results.

Figure 6.9: Maze overview rendered with Cycles engine inside Blender. Render time approximately 2 hours on nvidia GTX970.

The Fig.6.10 shows the overview of the maze. This is the default view at the beginning of the game. Fig.6.9 shows the reference offline render. Unity has sharper shadows compared to Unreal. However, shadows in Unreal have darker and colder feeling because of the stronger ambient occlusion effect. In the offline render, the shadows are sharp at the touching point and get softer with the distance. Also in the offline realistic simulation, the direct light has to be much stronger to create sufficient indirect reflection, close to the game engine results. The offline render is much more contrast between lights and shadows. There is also no ambient occlusion.

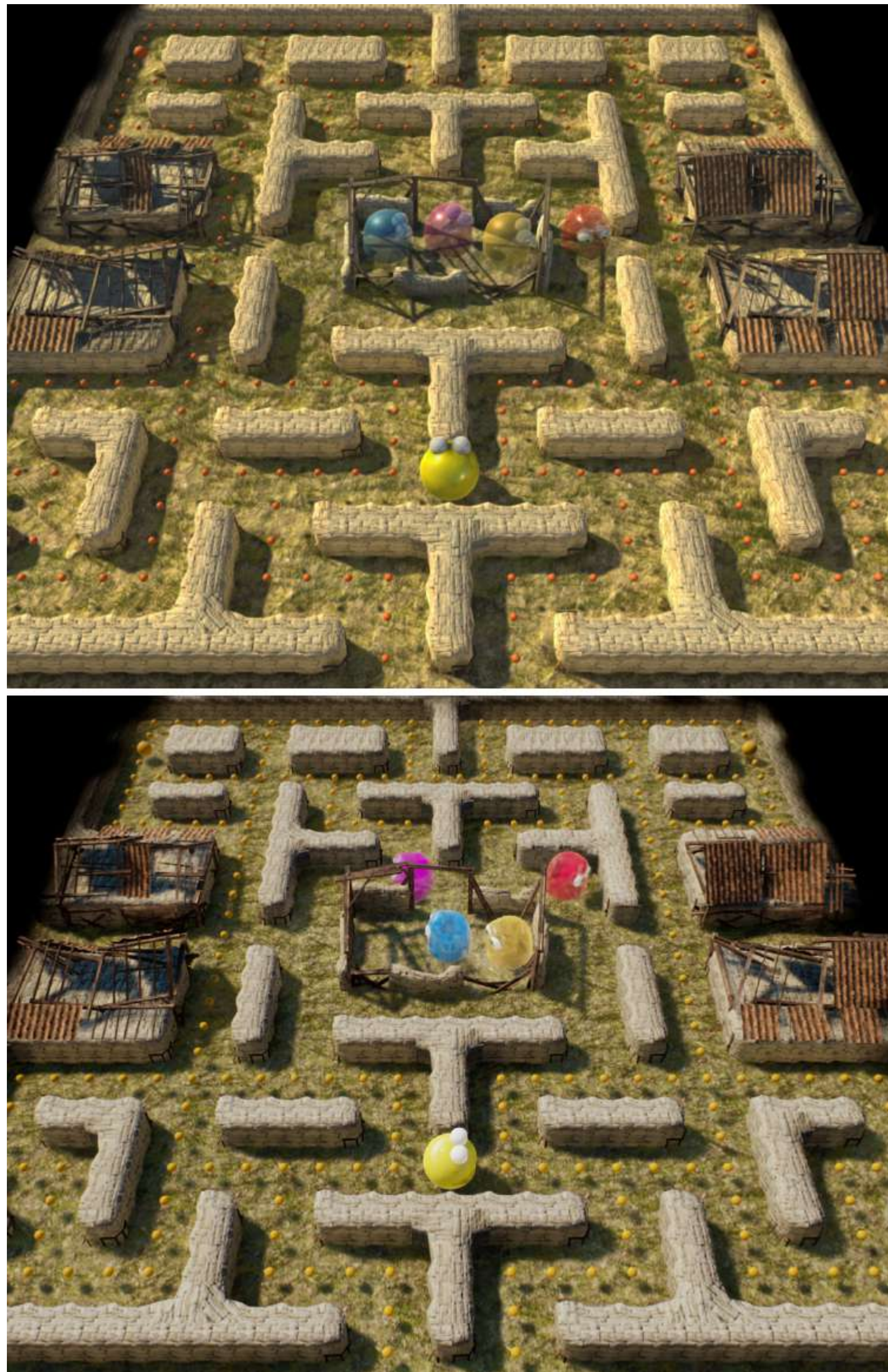


Figure 6.10: Maze overview comparison. The upper image is produced by Unity, the lower one comes from Unreal.

Figure 6.11: Pac-Man closeup, reference render from Blender Cycles. Yellow light reflected from the Pac-Man to the wall corner is apparent.

Figures 6.12 and 6.11 show detailed view of the Pac-Man. The most noticeable difference of the real-time pictures and off-line render is the indirect light reflected from Pac-Man. Game engines can precompute the light reflected from static meshes. Bake it into the lightmaps or apply it to the dynamic meshes. However, in Unity neither Unreal there is no simulation of indirect light reflected from the dynamic mesh (Pac-Man). In Fig.6.11 we can clearly see the yellow light on the wall corner left from Pac-Man. This light reflects from Pac-Man, and it is missing in the real-time pictures.

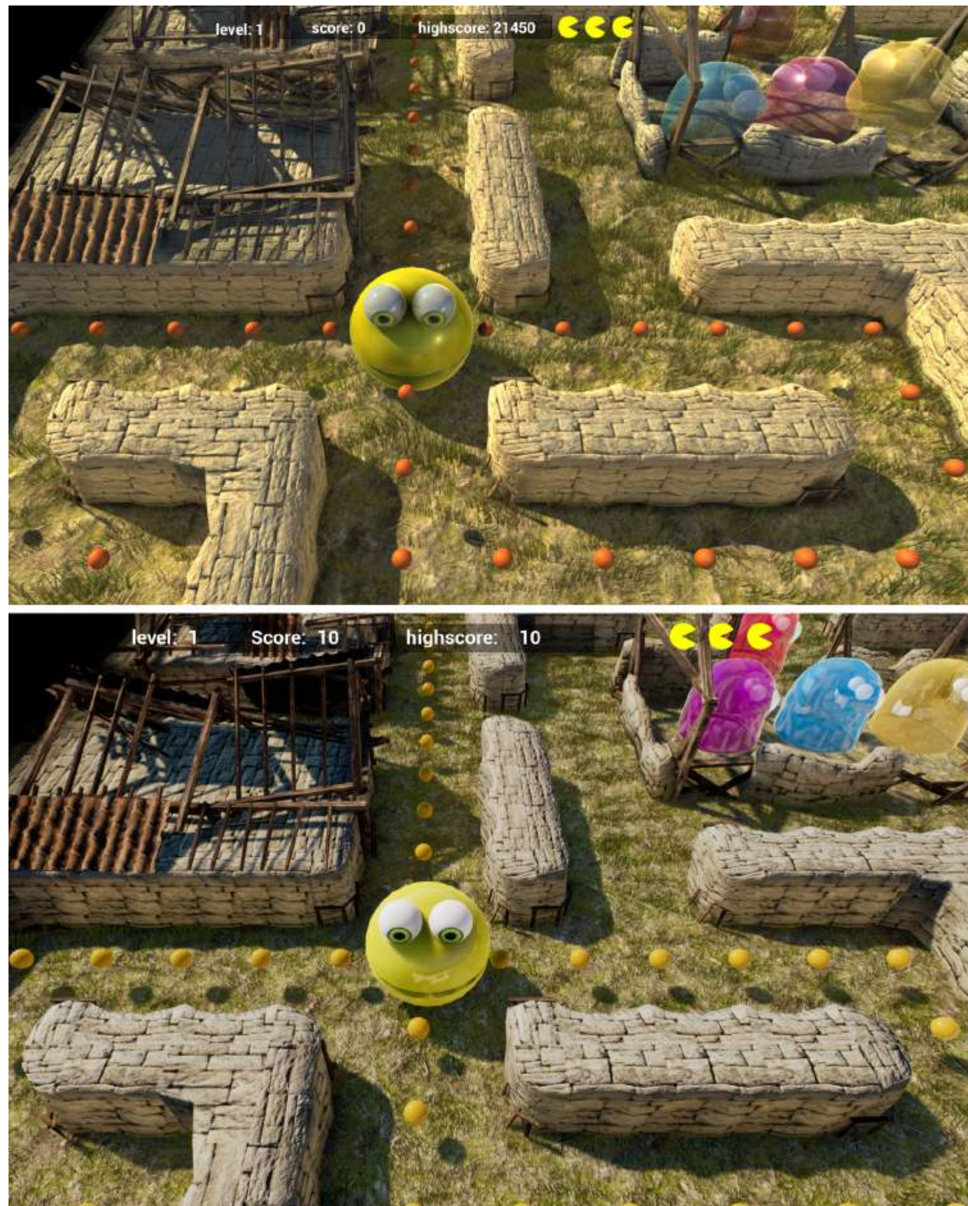


Figure 6.12: Pac-Man close up from Unity (top) and Unreal (bottom). None of the images shows light reflected from the Pac-Man to the wall corner.

Figure 6.13: Ghost close up reference rendered with Blender Cycles. It is obvious that the ghosts with refractive glass material cast shadows.

The ghosts are visible in Figs.6.14, 6.13. The material in Unity is just transparent. There is no light reflection or refraction. There are possibilities of adding this features to transparent materials with custom shaders, but the Standard Shader does not support that. Unreal supports light refraction and reflection on transparent meshes together, although the refraction is somehow simplified. These parameters also depend on the view angle thanks to the Fresnel effect [AMMH08](pages 231-236). Fig.6.13 shows the offline simulation, and there are ghost shadows visible which are not visible in the real-time pictures. Also, the light on the wall (lower right corner) is much brighter compared to the shadows.

The last comparison (Fig.6.15) shows the mobile versions of the benchmark. The Unity version runs almost twice slower than the Unreal version. However, the shadows are a little sharper, and the bumps on the walls are much more clear in Unity compared to Unreal. The Unreal has more vivid colors and the indirect light reflected from the ground to the biscuits is much more visible.

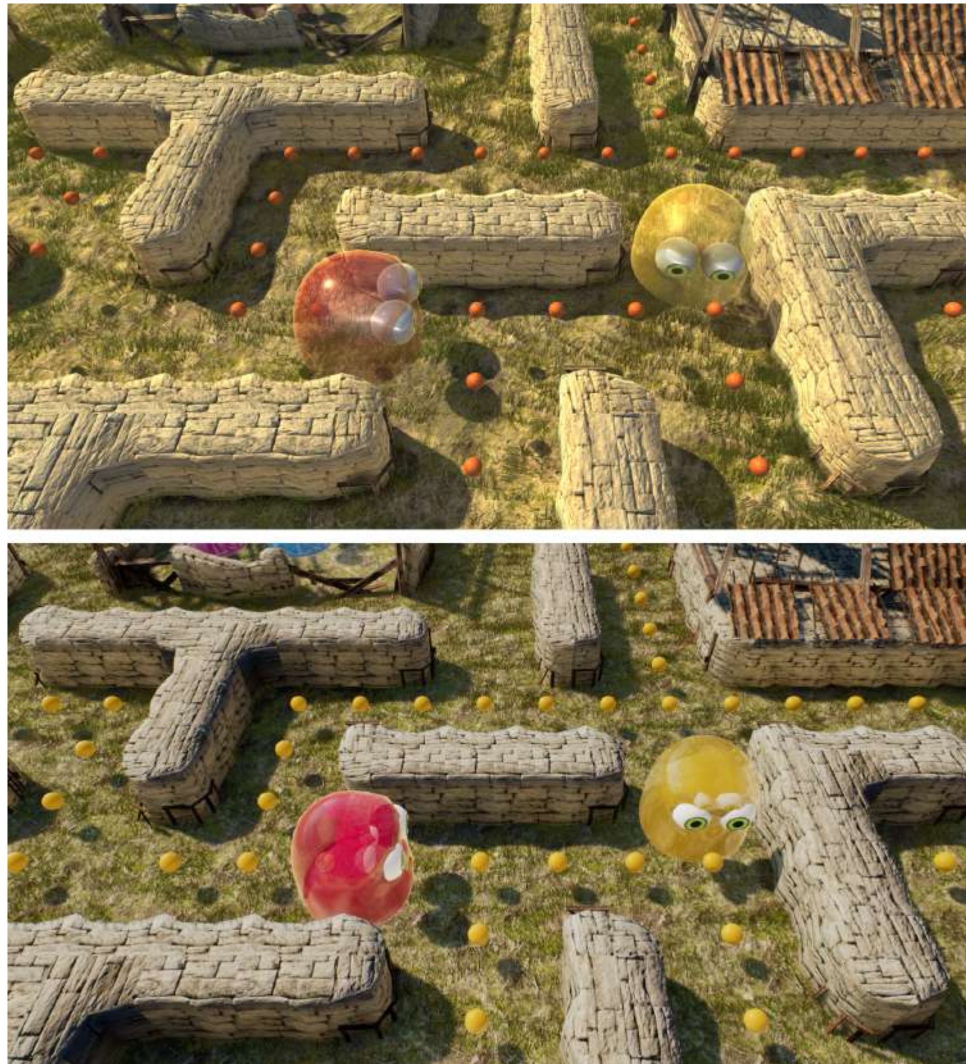


Figure 6.14: Ghost close up rendered with game engines. The upper image is Unity; the lower one is from Unreal. Unity supports no refraction; the ghosts are only transparent The indirect light reflected from the floor to the biscuits is also much more visible in the Unreal version.

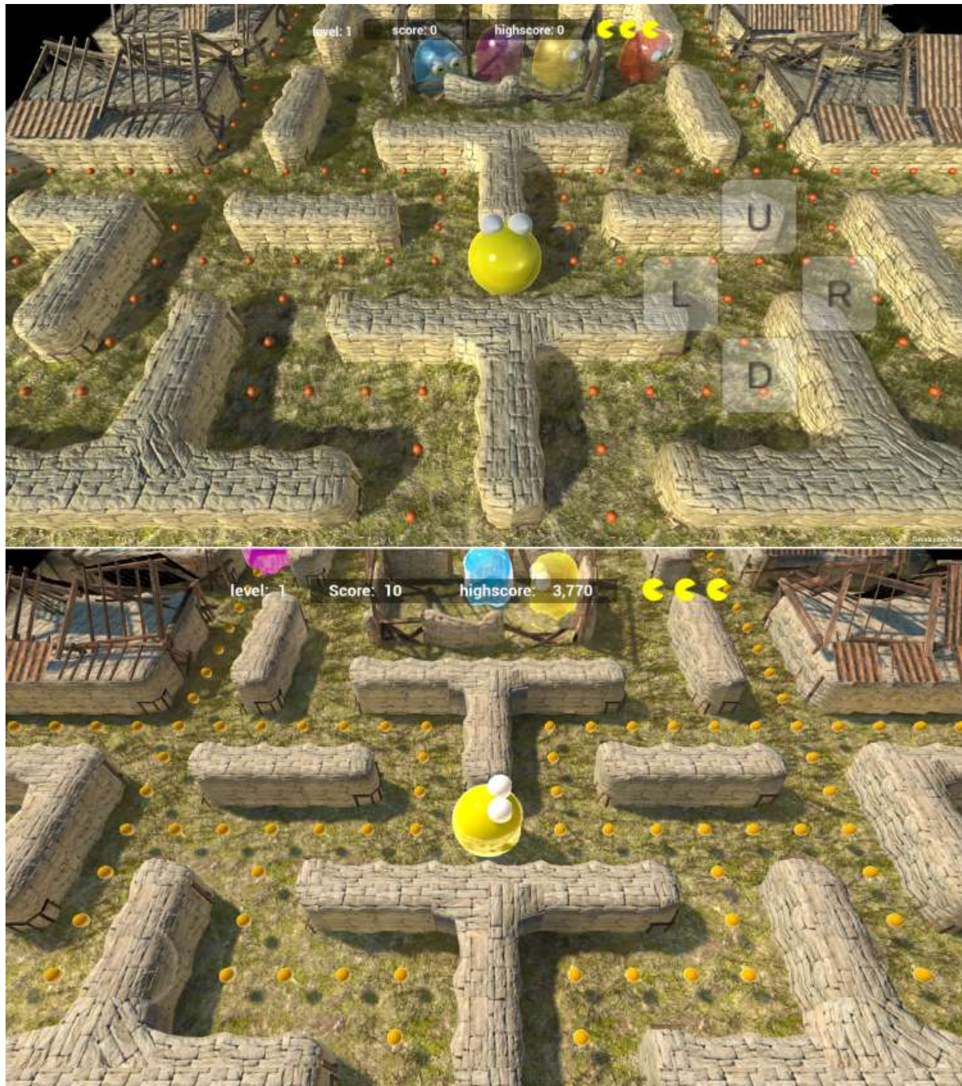


Figure 6.15: The mobile versions running on Samsung Galaxy S6. The top one is Unity (53ms), the bottom one is Unreal (30ms).

6.4 Subjective developer's opinion

Developing in Unity is easy. The engine's architecture is quite simple. There are game objects inside the scene, each object has components, and that is all. Some of the components can be scripts, which control the behavior of the object and modifies the variables of other components. The scripting language C# is very powerful. I was able to learn the Unity framework within a week to become reasonably productive. Also, the documentation is very well written. It contains explanations and examples of the code, which can be copy-pasted and modified for user's needs. The community is very large, I have found all of my questions answered at answers.unity3D.com.

Developing a game in Unity is comfortable. Unity solves many problems for me, so I do not have to care about them. The development process is fast and efficient. Because the Engine does not have any strict architecture considering the script relations, it is up to the developer to manage the overall structure by himself. If a script has a public variable, the variable automatically appears in the Editor, and the initial value can be set in the Editor, outside of the script. That is comfortable for tweaking the gameplay. However, it supports hand-made relations between the objects and high coupling. The code becomes a mess very easily because assigning an object as a reference to a script's variable works as drag and drop. Once there are many objects with different scripts, and all of them are connected inside the editor, it is easy to loose control the game's behavior.

Unity offers a Prefabs concept which helps to put the scene together. Prefab is a set of objects connected together with variables already set up. This pre-made object can be then instantiated and modified to create similar instances. For example, I have created all the ghosts from one prefab. They have a lot of the behavior in common. Just the pathfinding strategies and colors are different. Another example of a prefab use are the biscuits. They all look the same and have the same tags. Therefore they are instantiated from one prefab with a script.

Unity materials are solved mainly with the Unity Standard Shader which is easy to set up and does not force the user to spend too much time on it. The light baking was great and surprisingly fast. However the post processing is weak, there were some scripts for screen space effects, but since they were not part of the engine, the results turned out rather average. Export to various platforms works fine without any problems. The setup of export is simple and well documented.

I have enjoyed the Unity development a lot. It is a pleasure to work with Unity, the results appear pretty quickly, the development is efficient, and the workspace itself is simple to understand and use. I would recommend

Unity to any beginner game developer. It is a great platform.

Starting with Unreal was nothing simple. Clearly said, Unreal is not suited for beginners. The engine is very powerful. However, it is equally complex as well. The first weeks with Unreal were painful. The biggest problem was different architecture from Unity.

In Unity, there is a game object in the scene, and this object has a list of components. One of the components is a script which controls the others and adds the interactive functionality. In Unreal, the basic idea is that every interactive object is a Blueprint. The Blueprint (or C++ script) stays on top of a hierarchy of components. The components could be various mesh objects, lights, cameras, functions... With this philosophy, Blueprint represents an object but functionally is closer to Unity's Prefab.

Scripting the game in C++ is complicated. The framework is open source; it is very large and complex. The compilation times are very long compared to Unity. I have spent two weeks trying to script the game using C++, then I gave up and moved to blueprints entirely. Blueprints are an event-based way of visual scripting. My first impression was that visual scripting could not be anything serious, but blueprints cover almost everything a developer might need from scripting logic. It took me a few days to get used to connecting nodes instead of writing code. However, the development has accelerated rapidly. I have ended up with two C++ scripts in the entire benchmark, I have solved the rest with blueprints.

The more time I have spent with Unreal, the more I have liked the environment. The Unreal tools are much more powerful that those in Unity. Moreover Unreal covers some areas that Unity entirely skips like an advanced terrain creator or vegetation tools. The AI tools are a bit complicated, but they create a solid base for creating an advanced character logic.

Material creation in the Unreal engine is breathtaking. With the node-based logic similar to Blueprints Unreal allows us to write shaders without ever realizing it. The process of creating new materials is creative, and this amazing tool opens entirely new possibilities of materials.

In comparison to Unity, the Unreal source projects are very large; the development is slower. However, the results are much more solid. Unreal aims on large game development; expects skilled developers and gamers with powerful machines.

For any simple or mobile project, I would choose Unity, because of its ease of use. For anything serious targeted on PC or consoles, I would go with Unreal.

Chapter 7

Conclusions

In this bachelor's thesis, I have proposed a method for comparison of the game engines. The method is based on implementing a game of appropriate complexity using tested game engines and measuring the performance. I have implemented the Pac-Man game in Unity and Unreal engine and deployed it on PC, notebook, Android mobile and GearVR platforms. The Pac-Man benchmark can scale the load from one up to seven independent maze instances.

At first, I have briefly described game development process and the contemporary game engines in Chapter 2. Then I have analyzed the Pac-Man game and clarified the principles of the game and how they are used in the benchmark in Chapter 3. After that, I have implemented the game twice using Unity and Unreal engines with physically based shaders and models from Blender animated with Shape Keys. I have described the development process and techniques that I have used in Chapters 4 and 5. I have deployed the benchmark to for platforms PC, laptop, mobile, and VR. I have designed tests to measure the system's performance on each platform, performed the testing and presented the results in Chapter 6.

The Unity benchmark runs with frame time around 4ms on PC, 55ms on the laptop. In Unity adding another maze instances does not multiply the frame times, but just adds the constant of 4ms/maze, both on the PC and laptop. I found that interesting concerning the significant difference between the graphical cards. On mobile, it runs with frame time slightly above 50ms. The VR version's frame time moves around 20ms, which is not optimal, but acceptable on the GearVR.

The Unreal Benchmark runs with frame time slightly above 10ms with one maze on PC. However, the time does not grow with adding another maze instances. That means with one instance Unreal is twice slower than Unity,

but with seven instances Unreal becomes three times faster. The notebook was too weak for the Unreal benchmark; the resulting frame times stayed between 85 and 110ms per frame. On mobile, the Unreal benchmark runs with 30ms per frame, which is much better result compared to Unity. I was unable to measure the VR times correctly. However the game was running fluently, so I expect frame times around 16ms. The frame time is independent on the projection size in Unity benchmark. However, it is dependent in the Unreal benchmark.

In Chapter 6 I have also shown a visual comparison of the benchmarks with an off-line render from Blender Cycles and explained the differences. Lastly, I presented my humble opinions and recommendations from the perspective of a developer.

As a future work, I would like to test the implementation in Cry Engine, mainly to compare the visual results with Unreal. I might be interesting to deploy the benchmark on more powerful VR platforms such as Oculus Rift C1 or the HTC Vive to compare the performance of a full benchmark in VR. Concerning Unreal and Unity, the next steps are the location of the bottlenecks and proposal of optimization techniques. The final output of such work in the future might be an optimization guide for the game developers for each engine and target platform.

References

[Ac] E. F. Anderson and col., *Choosing the infrastructure for entertainment and serious computer games - a whiteroom benchmark for game engine selection*, 2013 5th Intl. Conf. on Games and Virtual Worlds for Serious Apps, pp. 1–8.

[AMMH08] Tomas Akenine-Moller, Tomas Moller, and Eric Haines, *Real-time rendering*, 3rd ed., CRC Press, 2008.

[Bir] Chad Birch, *Understanding pac-man ghost behavior*, http://gameinternals.com/post/2072558330/, Accessed: 2017-05-09.

[Doc] Unreal Engine 4 Documentation, *Unreal engine 4 documentation*, https://docs.unrealengine.com/latest/INT/Engine/Rendering/PostProcessEffects/, Accessed: 2017-05-07.

[Dra] Atul Dravid, *Understanding ies lights*, http://www.cgarena.com/freestuff/tutorials/max/ieslights/, Accessed: 2017-05-07.

[Ebe04] D. H. Eberly, *3d game engine architecture: Engineering real-time applications with wild magic*, Morgan Kaufmann, 2004.

[Fri] Jay Friesen, *What is look up table (lut), anyway?*, http://nofilmschool.com/2011/05/what-is-a-look-up-table-lut-anyway, Accessed: 2017-05-07.

[FS15] Gottfried Hofmann Frederik Steinmetz, *The cycles encyclopedia*, Blender Foundation, 2015.

[Geo] Geometrics, *Enlighten*, http://www.geomerics.com/enlighten/, Accessed: 2017-05-20.

[Kor] Maria Korolov, *Report: 98phones*, http://www.hypergridbusiness.com/2016/11/report-98-of-vr-headsets-sold-this-year-are-for-mobile-phones/, Accessed: 2017-05-09.

[Lon07] T. Long, *Oct. 10, 1979: Pac-man brings gaming into pleistocene era*, http://archive.wired.com/science/discoveries/news/2007/10/dayintech_1010, 2007.

[Par15] Tony Parisi, *Learning virtual reality : developing immersive experiences and applications for desktop, web, and mobile*, O'Reilly Media, Inc, Sebastopol, CA, 2015.

[PDdFP10] P. Petridis, I. Dunwell, S. de Freitas, and D. Panzoli, *An engine selection methodology for high fidelity serious games*, 2010 2nd Intl. Conf. on Games and Virtual Worlds for Serious Appss, March 2010, pp. 27–34.

[Pit] Jamey Pittman, *The pac-man dossier*, http://www.gamasutra.com/view/feature/3938/the_pacman_dossier.php?print=1, Accessed: 2017-05-19.

[Sch08] J. Schell, *The art of game design: A book of lenses*, CRC Press, 2008.

[Sob] Troy James Sobotka, *Filmic view and look transformations for blender*, https://sobotka.github.io/filmic-blender/, Accessed: 2017-05-10.

[VRo] *Official unity documentation: Optimisation for vr*, http://archive.wired.com/science/discoveries/news/2007/10/dayintech_1010.

[Wat11] A. Watkins, *Creating games with unity and maya*, Focal Press, 2011.

[Web] The Next Web, *This engine is dominating the gaming industry right now*, https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now/, Accessed: 2017-05-10.

Figures

2.1 Example of a first person view. Player is looking through the characters eyes, he can see his arms and weapon. Image courtesy Crytek. 4

2.2 Game engines components overview. 5

2.3 Viewing frustum of the camera. 6

2.4 Victor facial samples - an example of using blend shapes as facial expressions. Image courtesy Cosmos Laundromat, opensource film 8

2.5 NavMesh of the Pac-Man benchmark in Unreal Engine. 10

2.6 Logotypes of the most used 3rd party engines. 11

3.1 Pac-Man, the main character. 13

3.2 Screen from the original Pac-Man maze. Image courtesy of Shaune Williams. 14

3.3 Benchmark maze with various materials and precomputed indirect lighting. Screenshot from the Unity Benchmark version. 15

3.4 Components of the Pac-Man control system. 16

3.5 Decisions during turn on a cross. 16

3.6 There are four ghosts in the game. 17

3.7 The grid that shows 28x31 tiles creating the original Pac-Man Maze. 19

3.8 Tile evaluation diagram describes the process of choosing the right tile type to be instantiated. 20

3.9 The Unity Editor environment. 22

4.1 The Unity Editor environment. 24

4.2 Ghost navigation system in Unity. 26

4.3 Animation using shape keys inside the Unity Editor. 27

4.4 Animation using shape keys inside the Unity Editor. 28

4.5 Unity Tesselation Shader used on the roof, creating real bump according to the height map. 28

4.6 Screen- space effects setup. Motion blur, Antialiasing and ambient occlusion 30

5.1 The Unreal Editor environment. 32

5.2 An example of simple blueprint for Ghost Target update. 32

5.3 Indirect event invocation using the level blueprint. 34

5.4 Diagram of the main components in the Unreal Pac-Man Benchmark. 34

5.5 The basic Artificial Intelligence setup in Unreal Engine. 35

5.6 The ghost control setup in Unreal Pac-Man Benchmark. 35

5.7 The animation curves inside The Unreal Editor. 36

5.8 An example of a parallax material of the roof. 37

5.9 The ghost reflective material. 37

5.10 The Unreal Motion Graphics UI Designer interface. 39

5.11 Chromatic aberration example, image courtesy Epic Games, Inc. . . 39

5.12 Look Up Tables and the color graded results, image courtesy Epic Games, Inc. 40

5.13 GPU Visualizer showing the duration of processes. 40

6.1 Average frame times on PC and notebook. Versions of the benchmark with one, four and seven maze instances. (1.2 - 8.4 million vertices) . 46

6.2 Benchmark results on Android and GearVR platforms. Unreal version of the VR was not measured exactly, however from the character of the gameplay I assume the frame time is around 16ms. 46

6.3 The effect of the projection size of one maze to the framerate. Unity frame time stays the same, while Unreal time decreases almost three times. 47

6.4 Performance speed up with lowering the render settings. Measured with one maze instance on PC. 47

6.5 Size of the final builds and the game projects in MB. 48

6.6 Frame time measured with Unreal benchmark running on a notebook with one maze instance. The frame time decreases based on the screen coverage and number of objects in the scene. 48

6.7 Frame time measured with Unity benchmark running on a PC with one maze instance. The frame time decrease peaks are caused by the garbage collector. 49

6.8 An example of using Filmic color management in Blender cycles. Image courtesy Blender Guru. 50

6.9 Maze overview rendered with Cycles engine inside Blender. Render time approximately 2 hours on nvidia GTX970. 51

6.10 Maze overview comparison. The upper image is produced by Unity, the lower one comes from Unreal. 52

6.11 Pac-Man closeup, reference render from Blender Cycles. Yellow light reflected from the Pac-Man to the wall corner is apparent. 53

6.12 Pac-Man close up from Unity (top) and Unreal (bottom). None of the images shows light reflected from the Pac-Man to the wall corner. 54

6.13 Ghost close up reference rendered with Blender Cycles. It is obvious that the ghosts with refractive glass material cast shadows. 55

6.14 Ghost close up rendered with game engines. The upper image is Unity; the lower one is from Unreal. Unity supports no refraction; the ghosts are only transparent The indirect light reflected from the floor to the biscuits is also much more visible in the Unreal version. 56

6.15 The mobile versions running on Samsung Galaxy S6. The top one is Unity (53ms), the bottom one is Unreal (30ms). 57

Appendix A

CD Contents

```
DVD
├── bin ........................ contains the executable games
├── images ..................... screens from the game
├── latex ...................... LATEX source files of this text
├── src ........................ source files of the benchmarks
├── thesis.pdf ................. PDF version of this text
└── README.txt
```