The Pacman Benchmark

Antonín Šmíd*
Supervised by: Jiří Bittner

Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University in Prague, CZ

Abstract

Contemporary game engines are invaluable tools for game development. There are numerous engines available, each of which excels in certain features. We have developed a simple game engine benchmark using a scalable 3D reimplementation of the classical Pacman game.

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

Keywords: Game Engine, Benchmark, Unity, 3D Pacman reimplementation

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. That means that 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 we have decided to compare two of the major engines on the today's market: Unity3D (version 5) and Unreal Engine 4.

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 we aim to accomplish.

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

In this paper, we cover the Unity implementation and measurement results of the benchmark. We will continue by implementing the Unreal benchmark in the future. At first, we briefly describe the game engine's components in Section 3. Then we go through the principles of the Pacman game and explain how we use them to test the game engine's performance. In Section 5 we cover the details of the benchmark implementation in Unity game engine. Finally, in Section 6 we present the measured data.

2 Related work

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* [2] may offer just API and the developer has to code all the logic including the game loop.

Anderson et al. [2] 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 proposed engine selection methodology proposed by Petridis et al. [5] empathizes the rendering features. However, those papers do not mention any performance measurements and we would like to include those as well.

*emails: smidanto@fel.cvut.cz, bittner@fel.cvut.cz

3 Game engine components

Game engines are divided into several components, each one providing specific functionality (see Fig. 1) [3]. We will briefly go through the most important ones.

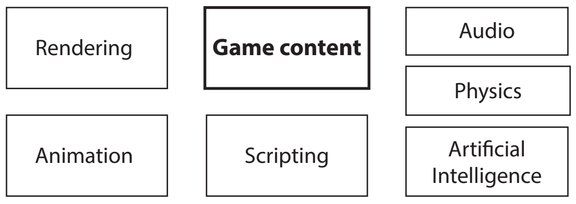

Figure 1: Game engine's components overview

3.1 Rendering

The rendering engine is one of the most important parts of the game engine. It accesses the graphics card API and allows to draw 3D objects in the scene. The rendering engine calculates objects positions in the camera space, materials, and lighting. It usually comes together with a number of shaders to simulate different materials. Rendering often takes the majority of the update time. Therefore, we focus the benchmark primarily on the rendering features.

3.2 Animation

As some of the game objects do not only move in space but change their shape or move their parts, every game engine needs to handle animations. These could be skeleton animations (walking), shape animations (facial expressions) or just interpolating some attributes of the game objects (transforms, color, speed).

3.3 Physics

The physical engine simulates the physical behavior of the objects in the game. 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.

3.4 Artificial intelligence

Some engines offer API to configure artificial intelligence (AI) of the characters. Typical tasks are finding a path between two points, moving the characters along this path. Another use case is the representation of the behavior of AI characters, for example, using behavioral trees which decide what the characters should do, how to react to players actions and trigger animations.

3.5 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, interact or communicate with each other.

3.6 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 players position.

4 The Pacman Benchmark

In this chapter, we will describe the Pacman game mechanics in the context of the tested components.

Game design is a difficult discipline [6]. One of the well-done designs is the Pacman game which 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 Pacman was an arcade game developed by the Namco company [1], released in 1980 in Japan [4]. Pacman (Fig. 2) [2] is a yellow ball with big mouth eating little dots, also called biscuits. The player controls the Pacman through the maze. There are four ghosts in the maze, who are trying to catch the Pacman.

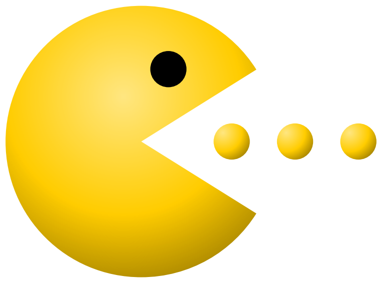

Figure 2: Pacman, the main character.

The game is simple to play; originally it was controlled by a joystick. In this benchmark version we use a script to move the Pacman precisely or keyboard arrows to make the game actually playable. We 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/

[2]https://pixabay.com/p-151558/, CC0 Public Domain

4.1 Game Concept

The Pacman is an arcade game. The main yellow character is going through the maze, eats biscuits and avoids the ghosts. Ghosts begin in the prison located in the center of the maze. Pacman 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 Pacman 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 Pacman eats all the biscuits then 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.

4.2 The Maze

In the original Pacman, there is a static maze (Fig. 3). The large orange circles are the energizers. In our benchmark, we 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.

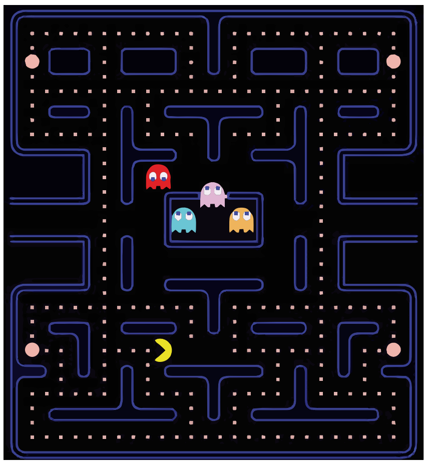

Figure 3: Game screen of the original Pacman maze. (image courtesy of Shaune Williams)

In the middle of the maze, there is a prison for three ghosts. Pacman 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 28×31 tiles; it is symmetrical around the Y axis. The prison is 8×5 tiles including walls.

The maze itself (Fig. 4) offers a variety of opportunities to test the rendering engine. In our 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. Direct shadows are computed in real-time. The static maze uses precomputed indirect lighting, baked into the lightmaps [7].

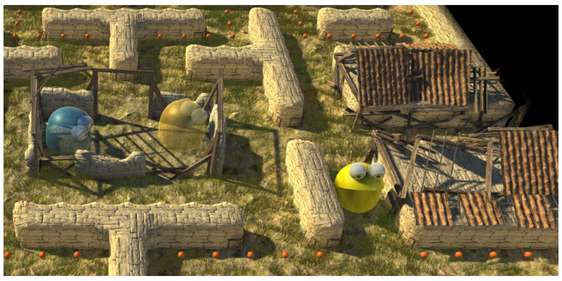

Figure 4: Benchmark maze with various materials and precomputed indirect lighting.

4.3 AI characters

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

In the game, there are four ghosts (Fig. 5). 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 5: There are four ghosts in the game.

The red ghost Blinky goes after the Pacman. The Pacman character itself is his target.

Pacman 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 Pacman is heading up, the bait moves another 5 tiles left.

The blue ghost is called Inky. There is another bait, let's call it bait2. 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 bait2 is always in the middle of the line.

Ghost Clyde, the orange one, has Pacman as his target. However, when he approaches Pacman to a distance of 10 tiles, he changes his mind and switches the target to the left corner. When the distance to Pacman 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, we could not use the navigation system of the AI game engine component. Instead, we have implemented custom scripts to control ghost's behavior. However, we 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 [3].

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 Pacman 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 Pacman 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 Pacman.

4.4 Scaling the problem

We aim 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. We have defined three game configurations (see Table 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 we had to modify some of the game components. For VR deploy, we use fast mobile shaders with baked light. However, on the PC version, we 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, we have duplicated the maze up to seven times and created autonomous mazes where ghosts move independently.

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

[3]http://gameinternals.com/post/2072558330/

configuration	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 1: The platform features overview.

5 Unity implementation analysis

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 [8]. We have used the Unity 5.3.3 version to implement our benchmark. In this chapter, we will describe the most significant sections of the development.

5.1 Pacman movement

The original Pacman 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, we have created a system (Fig. 6) of collision detectors and forces to make the Pacman move right in the physically correct environment.

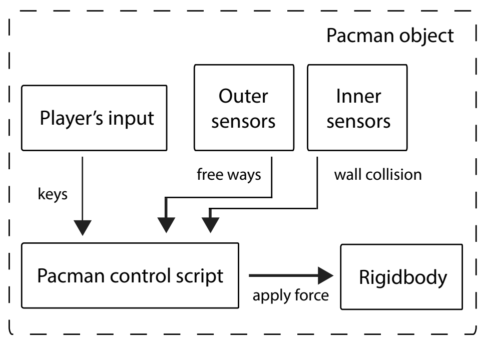

Figure 6: Components of the Pacman control system.

The basic characteristic of the Pacman'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 collides a wall. According to the key's map, we determine the direction of the force, that keeps pushing Pacman to move at a constant speed.

To avoid pulsing in speed, when the top speed is reached, we only add the force needed to achieve the target speed using a simple regulator based on the formula 1. Where *vmax* is maximum speed, *v* is current speed and *t* is the frame time.

$$F = m \times \frac{vmax - v}{\Delta t} \quad (1)$$

To deal with the unrealistic way of the Pacman's turning, we 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 Fig. 7). This is not physically correct, but it leads to the desired behavior.

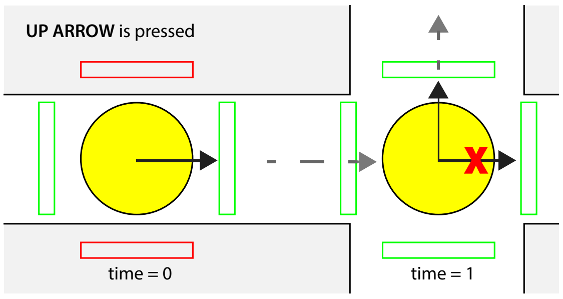

Figure 7: Decisions during turn on a cross.

5.2 Maze generator

The Pacman maze has to follow certain rules that we have described in Chapter 3.5. Our benchmark implementation allows creating custom mazes in conformity with those rules. The source for the maze is a text file with a matrix that describes the tiles. Inside the Unity Editor, a script can analyze the matrix and generate appropriate maze walls. This process has to be done before the compilation, because of the lightmaps, which Unity has to bake before it runs the game. It also needs to create the navigation mesh for the AI component.

Energizers and biscuits are instantiated during real-time at the beginning of every level.

5.3 Ghost AI

Unity has a navigation system implemented as a part of the engine. We can control AI behavior easily through the API calls. There is an automatic NavMesh generation, which constructs the mesh for AI navigation from static objects in the scene. Unity has NavAgent component, which controls the characters movement.

We could not use the NavMesh for complex pathfinding because the logic of ghost's decisions does not require that. To test the AI component of Unity, we let it move the ghosts from tile to tile as NavAgents with physical rigidbody.

5.4 Visuals

We have made models in Blender 2.7 and exported them into Unity through .fbx format. This format has transferred shape animations as well. We have created different shape keys in Blender and animated them inside of Unity as Blend Shapes (Fig. 8).

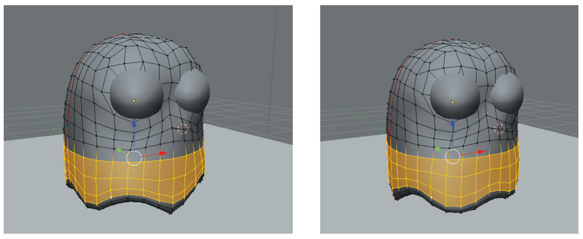

Figure 8: Ghost's shape keys inside of Blender.

For materials, we have used the Unity Standard Shader, a built-in implementation of Physically Based Rendering (PBR). In real-time graphics, this is quite a recent concept which empathizes realistic material behavior. In contrast to Phong Shading Model, the artist does not configure materials visual parameters, but it is physical properties such as glossiness or roughness.

To test one of the latest features in OpenGL 4, or DirectX 11 we used the parallax mapping [7]. This technology dynamically changes the number of vertices in the object. Based on the high map it generates real bumps, a new structure on top of the existing object. We have used Tesselation Shader on the roofs to achieve this effect.

For grass straws, we have used a mobile shader. It is fast and allows us to render a large number of transparent objects at once.

The ghosts use transparent Standard Shader. 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. We have enforced the ghosts to render on the top of the grass by changing the Custom Render Queue value [4].

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 the sky. The sky is spherical high dynamic range image [5]; Unity uses it as an ambient light source. The indirect light is then baked into lightmaps which contain the intensity and direction of the indirect light [7].

Concerning screen-space effects [7], we have decided to use typical representatives common for most AAA games. We have used anti-aliasing, motion blur, and ambient occlusion. Those effects are available only on the PC version of the Benchmark.

[4]http://answers.unity3d.com/questions/609021/

[5]The spherical sky image is part of Pro-Lighting: Skies package. https://www.blenderguru.com/product/pro-lighting-skies/

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

Table 2: Benchmark platforms. NR - native resolution, TR - tested resolution

5.5 Profiling

To monitor the engine's performance, we measure the frame time and save it into logs. For more detailed analysis of the use of systems resources, the game can be connected to Unity's profiler. The profiler allows us to monitor all the engine's components and their impact on the performance. There are graphs of CPU, memory, or graphics card usage over the time. This information can also be saved and analyzed later [6].

5.6 VR adjustments

To run the benchmark on mobile VR platform GearVR, we needed to optimize the environment [1]. We have turned off all the effects, used mobile shaders, baked the whole lighting setup. The large textures are no problem for the phone; it has enough memory. There are only 33k triangles. We have baked the lighting setup in Blender Cycles render engine. There is no doubt this version of the game looks much worse, but it runs on mobile as VR in real-time.

6 Results

We have designed four tests to measure frame time. The tests were executed with the script which simulated user input to create the same conditions multiple times. The test platforms are specified in Table 2.

6.1 Performance on different platforms

In the Fig. 9, we can see the PC and the notebook performance. We have measured one, four, and seven maze instances. If we add one maze (approximately 1.2 million verts), the frame time becomes about 4 ms longer. This is platform independent. PC with one maze reaches times around 4 ms; with 4 mazes, it is 17 ms; with 7 mazes around 30 ms. On the laptop, the frame times are much longer, but the shift between 1, 4, 7 maze instances remains the same, even though the graphics card is about 12 times slower [7].

[6]https://www.packtpub.com/mapt/book/game-development/9781785884580/1/ch01lvl1sec11/saving-and-loading-profiler-data

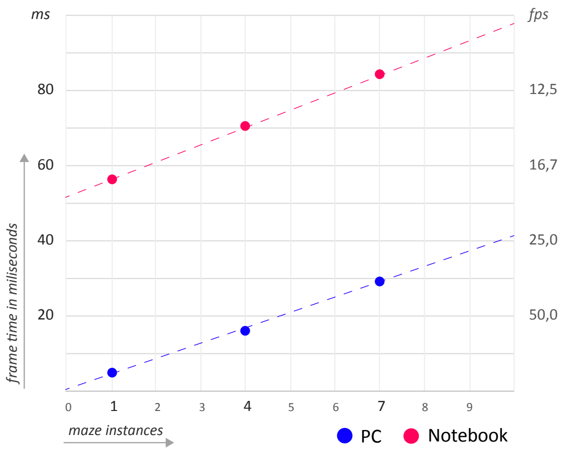

Figure 9: Average frame time on a PC and a Notebook in dependence on the number of the maze instances.

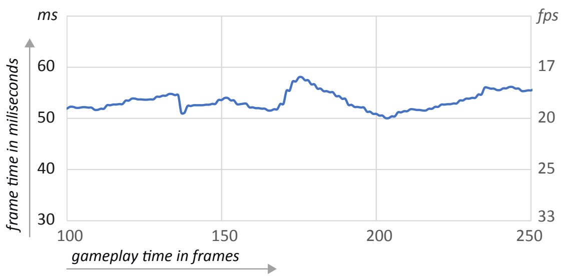

Figure 10: Frame time measured on Samsung S6.

The mobile version (Fig. 10) is the same with one exception of screen-space effects which are turned off. We have decreased the resolution from native $2560 \times 1440 px$ to FullHD to match the PC version; frame time ascended slightly over 50 ms. The VR times are in the Fig. 11. We have adapted the VR version to run in real-time (Chapter 4.6), although the times are not stable, they approach the target 60 fps.

The overview of average measured times (Fig. 12) will become a source for the most basic Unity-Unreal comparison. The PC and the Notebook values are derived from measurement with one maze instance.

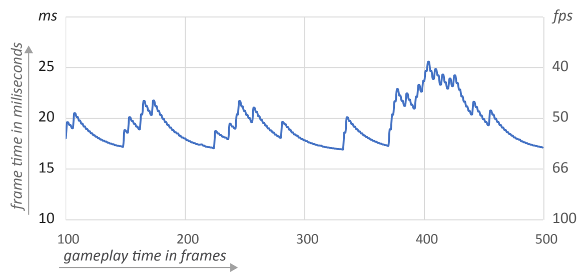

Figure 11: Frame time measured on GearVR.

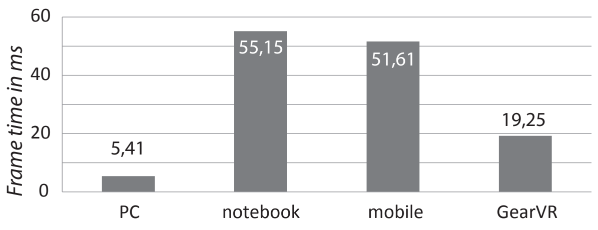

Figure 12: Overview of the Unity frame times.

6.2 Performance scaling on PC

We have tested the frame times just on PC by adding maze instances one by one (Fig. 13). The camera was stable at one point; just the instances have appeared on the screen. An interesting fact is, that it does not matter, how large the maze appears on the screen, frame time remains the same. This is shown in the first two measurements; single maze full screen and little single maze have similar results.

It is also clear here as in Fig. 9 that with each added maze instance, the frame time grows by a constant of approximately 4 ms.

6.3 Unity graphics quality settings

Unity offers different quality settings [8]. We have tested them on PC with 7 mazes (Fig. 14). Those presets can be selected before the game is launched. The settings affect mainly light quality; shadow draw distance, anti-aliasing or LOD distance settings. Their impact on the frame time is not substantial. However, the visuals are much worse on Fastest compared to Fantastic. Therefore it is 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. Detailed information on the exact values of the default quality settings can be found in the Unity manual.

6.4 Engine's components load

We present a brief analysis of the engine's component load. We have measured all the platform versions running inside of Unity Editor with the Profiler. On PC (Fig. 16) the rendering component (the blue parts) takes over 90% of the frame time. Drawing performs most render passes and applies shaders, UpdateDepthNormalsTexture and UpdateDepthTexture are calls creating the G-buffer textures[9]. The G-buffer contains depths and normals for screen pixels and is used with the screen-space effects. The UpdateDepthNormalsTexture is invoked by the Screen space ambient occlusion. These effects are turned off on the mobile and VR platforms. The final component load is shown in Fig. 15. The mobile and VR versions run much faster then the ones for PC. VR version has the target 90 fps on PC with Oculus Rift 2. Therefore the Other component contains around 9 ms of waiting for sync.

[7]$http://www.videocardbenchmark.net/high_end_gpus.html$
[8]https://docs.unity3d.com/Manual/class-QualitySettings.html

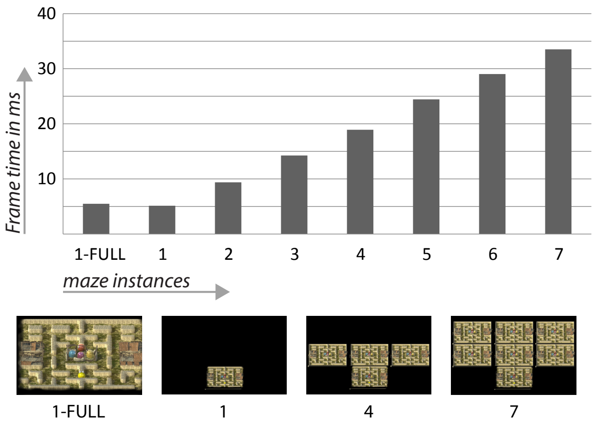

Figure 13: Frame times are growing with the number of maze instances on PC.

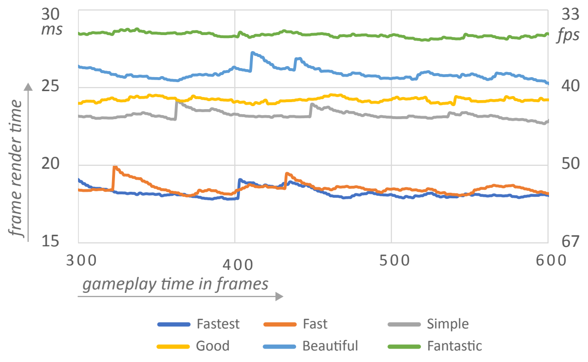

Figure 14: Unity runtime quality settings.

7 Conclusions

We have carried out the first part of game engine comparison. We have designed and implemented a benchmark application that measures the Unity game engine's performance on PC, notebook, mobile and GearVR. This benchmark was built as a game of adequate complexity - 3D

[9]https://docs.unity3d.com/Manual/SL-CameraDepthTexture.html

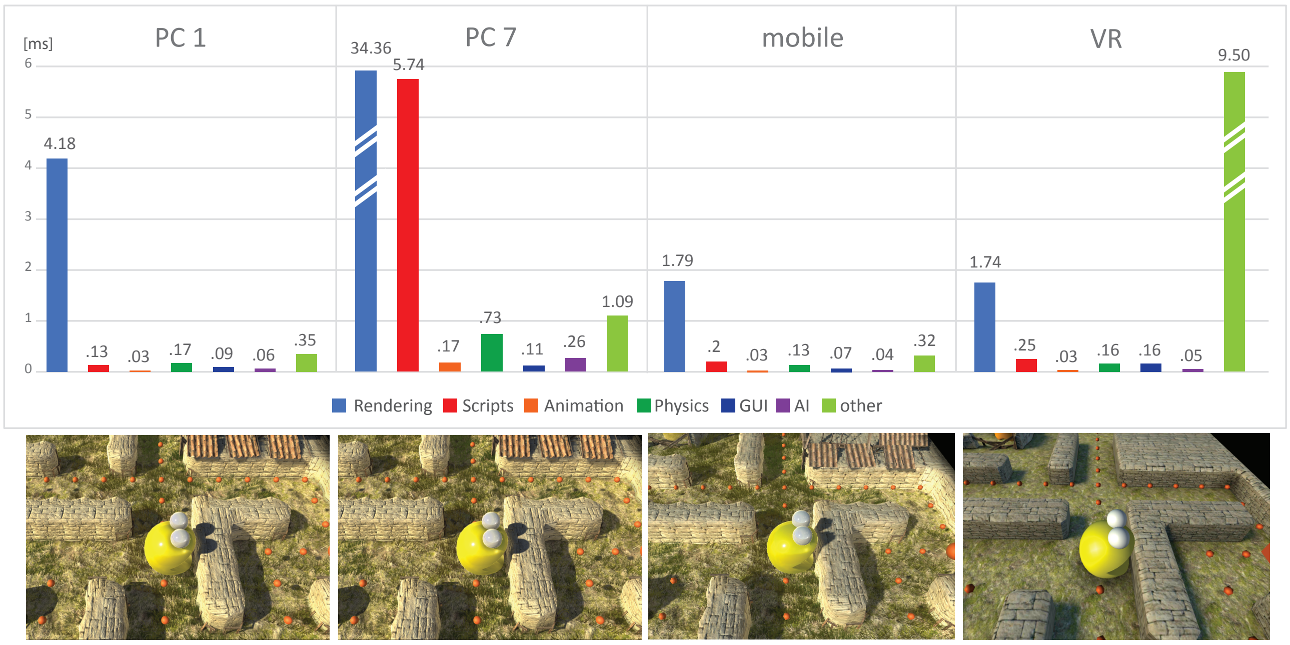

Figure 15: Components load for different platforms.

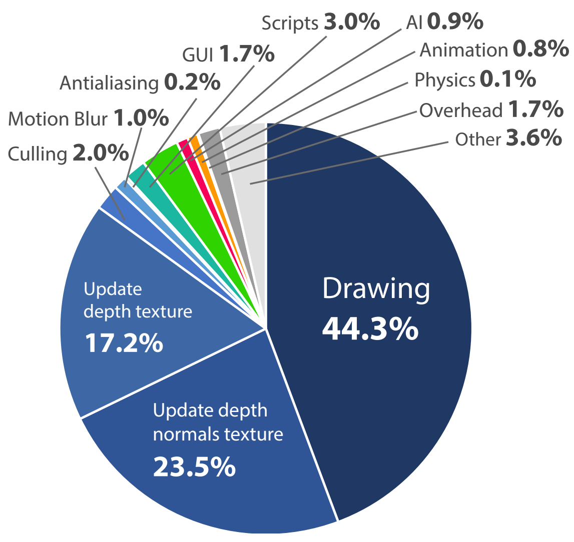

Figure 16: Components load ratio on PC, 1 maze instance.

reimplementation of the Pacman.

The application runs with frame time around 4 ms on PC, 55 ms on the notebook. Adding another maze instances does not multiply the frame times, but just adds the constant of 4 ms/maze, both on the PC and notebook. It is an unexpected result concerning the significant difference between the graphical cards. On mobile, it runs with frame time slightly above 50 ms. The VR versions frame time moves around 20 ms, which is not optimal but acceptable on the GearVR. The screen space effects do make a significant difference in the length of the frame time, while the overall coverage of the screen does not.

We will continue by reimplementing the same benchmark application in Unreal engine and comparing the data to the results from Unity.

References

[1] *Official Unity documentation: Optimisation for VR.*

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

[3] D. H. Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic.* Morgan Kaufmann, 2004.

[4] T. Long. Oct. 10, 1979: Pac-man brings gaming into pleistocene era. *Wired*, 2007.

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

[6] J. Schell. *The Art of Game Design: A book of lenses.* CRC Press, 2008.

[7] N. Hoffman T. Akenine-Moller, E. Haines. *Real-Time Rendering*. A K Peters, 2008.

[8] A. Watkins. *Creating Games with Unity and Maya.* Focal Press, 2011.