FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Physics Simulation Game Engine Benchmark
Student:	Marek Papinčák
Supervisor:	doc. Ing. Jiří Bittner, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

Review the existing tools and methods used for physics simulation in contemporary game engines. In the review, cover also the existing benchmarks created for evaluation of physics simulation performance. Identify parts of physics simulation with the highest computational overhead. Design at least three test scenarios that will allow to evaluate the dependence of the simulation on carefully selected parameters (e.g. number of colliding objects, number of simulated projectiles). Implement the designed simulation scenarios within Unity game engine and conduct a series of measurements that will analyze the behavior of the physics simulation. Finally, create a simple game that will make use of the tested scenarios.

References

[1] Jason Gregory. Game Engine Architecture, 2nd edition. CRC Press, 2014.
[2] Ian Millington. Game Physics Engine Development, 2nd edition. CRC Press, 2010.
[3] Unity User Manual. Unity Technologies, 2017. Available at https://docs.unity3d.com/Manual/index.html
[4] Antonín Šmíd. Comparison of the Unity and Unreal Engines. Bachelor Thesis, CTU FEE, 2017.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 8, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Physics Simulation Game Engine Benchmark

Marek Papinčák

Department of Software Engineering

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

May 15, 2018

Acknowledgements

I am thankful to Jiri Bittner, an associate professor at the Department of Computer Graphics and Interaction, for sharing his expertise and helping me with this thesis.

Declaration

In Prague on May 15, 2018 ....

Citation of this thesis

Papinčák, Marek. *Physics Simulation Game Engine Benchmark*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Väčšina dnešných hier využíva hernú fyziku aby maximalizovala zážitok z hrania. Na trhu je niekoľko herných enginov s integrovaným fyzikálnym enginom, ktoré uľahčuju vývojárom implementáciu hernej fyziky do ich hier. V tejto práci sa zameriavam na Unity herný engine s jeho integrovaným PhysX fyzikálnym enginom. Implementoval som sedem testov, ktoré testujú fyzikálne komponenty v Unity a možu pomôcť herným vývojárom s výberom toho pravého herného enginu pre ich hru. Posledný benchmark je jednoduchá strategická hra, ktorá využíva komponenty testované v predošlých testoch.

Klíčová slova Game Engine, Benchmark, Game Physics, Unity, PhysX

Abstract

Most of the contemporary games use game physics to offer the full experience. There are several game engines on the market with integrated physics engine to help developers implement physics in their game. In this thesis, I concentrate on Unity game engine with its iteration of PhysX physics engine. I implemented seven benchmark to test its components and help developers decided whether the Unity is the right engine for their game. The last benchmark is a simple real-time strategy game that makes use of physics components tested in the other six benchmarks.

Keywords Game Engine, Benchmark, Game Physics, Unity, PhysX

Contents

List of Figures

List of Tables

Introduction

Motivation

Video game is a real-time interactive computer simulation set in the real or imaginary world. Player takes control of a game character and is presented with a set of challenges. Imagine saving New York as the Spider-Man, clearing dungeon with a party of heroes, winning the Champions league with your favourite team or controlling the whole army of Greeks at the battle of Thermopylae. This and much more have been tackled by different game genres over the years. While early games were considered to be toys for children, games of today are a multi-billion-dollar industry rivalling Hollywood in size and popularity.

Modern games require a large number of developers. A big part of those are artists, who create audio and visuals for the game. Engineers design and implement the software that makes the game work. Game designers create the game world, write a story and design gameplay. Testers identify bugs and boring parts of the game. Producers and publishers take care of the marketing and business side of the development.

Every good game requires a way to render objects on screen, animations, physics, user interface, audio and a lot of scripting to join them into a quality product (more in chapter 1). Developing these parts takes a lot of time, so programmers started to make universal tools. Some of these tools cover only one part (physics engine covers physics), while others offer the complete package - game engine. A game engine allows you to focus on world-building, gameplay, and graphical or audio assets. You can learn more about game engines in Game Engine Architecture by Jason Gregory [1].

Two of the most popular engines on the market are Unity and Unreal engine. A. Šmíd compared these two engines in his bachelor's thesis [2]. He concluded that Unity is comfortable and easy to use, while Unreal is robust and takes time to get used to. Unity is also more widespread and has a large community, so it's more possible to find a solution for your problem online. As a beginner in game development, I chose Unity so I could focus on the true matter at hand - game physics.

Most games are set in the real world and players expect that objects will behave in a physically correct way. Game physics ensures that world objects won't fall through each other and defines what happens if they collide (more in chapter 1). This has a large impact on performance. Developers have a tight technical budget that physics is only a small part of. They have to choose wisely what kind of physical effects they want to use and when.

In this thesis, I work with Unity, one of the most popular game engines in the world. Unity supports a variety of physical effects through its physical components. I created several benchmarks that test these components and measure their impact on performance.

Structure

In chapter 1, I go deeper into game physics, how Unity operates and what kind of physics components are supported. I talk about game design and how its influenced by physics in modern games. I go deeper into the technical side and talk about physics engine. In the second part of the chapter, I examine capabilities of PhysX, the physics engine integrated into Unity.

I have created seven benchmarks of Unity physics testing: performance of the collision system, accuracy of the collision system, fire accuracy, cloth component, custom deformable object, physics animations and physics in a game scenario. The final benchmark is a simple RTS game that incorporates all other benchmarks into one. All of the mentioned benchmarks are detailed in chapter 2. Results of the benchmarks can be found in chapter 3. I present the results in form of graphs, tables, pictures, and commentary.

Chapter 1

Game physics

1.1 Game design and the role of physics

Physics is an inherent part of any modern game. It defines everything from how objects move when they fall under gravity to what they do when they tumble around and come into contact with other objects. A game doesn't have to have complicated game physics to be enjoyable, but bad physics system will completely remove the fun from most games. Player expects real physics in a game like Battlefield with realistic character models and animations. It would be weird and disappointing if grenades wouldn't bounce off the walls or dead players would be stuck to the floor after being blown up.

Game physics can be also used as a gameplay feature and many games have it as their main selling point. A great example of a physical-based destruction model is Besiege where player can construct custom siege engines and destroy castles. Another legendary examples are the gravity gun in Half-life 2, the first use of real bullet ballistics in Sniper Elite or fully destructible buildings in Red Faction: Guerilla.

1.2 Games and game physics - How does it work?

Video game is an example of soft real-time interactive agent-based computer simulation. That is a lot of words, let's break it down. Simulation is a mathematical model of the real or imagined game world. Agent-based simulations are those in which a number of distinct agents (objects) interact. Interactive real-time simulation means that simulation responds to player input in real time and changes the game world accordingly. [1]

At the core of every real-time system is the concept of a deadline. Video games require screen to be updated 30 or 60 times per second in order

Figure 1.1: Enviroment destruction in Red Faction: Guerilla developed by Volition.

to provide the illusion of motion. This means that there's only so much information that computer can process in one frame. Developers have to balance how much of this information will be game logic, physics, AI etc. A soft real-time system is one in which missed deadlines are not catastrophic. Missed deadlines in games mean lower performance. If game misses too many deadlines it will start to stutter and become unplayable. [1]

Game physics is more accurately described as rigid body dynamics simulation. A rigid body is an idealized, infinitely hard, non-deformable solid object. Dynamics refers to the process of determining how these rigid bodies move and interact over time under the influence of forces. Dynamics simulation makes heavy use of the collision detection system in order to properly simulate various behaviours like bouncing, sliding, rolling and coming to rest. While collision system can be used standalone, all games that move objects about have some sort of collision detection.

1.3 Physics engine

Physics engine provides an accompanying way of simulating real-world responses for objects in games [3]. It is basically a big calculator that does the mathematics needed to simulate physics. A good example is movement: engine gets mass, gravity, velocity and friction properties from an object and computes its new location. Physics engine itself doesn't know which objects should move or have physical properties. It's up to the game or game engine to decide.

Physics engine that represents all required physical properties from real life is called general-purpose physics engine. This type of engine is used in most triple-A games that strive to create a believable world. Some physics engines focus only on one property like water or mud. One-purpose game engines are better suited for simulations or simple games that revolve around one game mechanic(ex. water racing).

There are two compelling advantages to using a physics engine. Physics effects take a lot of time and effort to develop. It is much easier to import a general-purpose physics engine. The second advantage is quality. Popular physics engines of today have been developed for many years by highly competent developers.

The main reason not to import a general-purpose physics engine is speed. It is quite processor intensive. When you are working with a very simple game environment, this generality can mean wasted processing power.

1.3.1 PhysX

PhysX is a multi-threaded physics simulation SDK used in the large majority of today's games. It supports rigid body dynamics, soft body dynamics, rag-dolls and character controllers, vehicle dynamics, particles, volumetric fluid simulation and cloth simulation including tearing and pressurized cloth. PhysX started out as a library called Novodex, produced and distributed by Ageia. It was later bought by NVIDIA. CPU version of the SDK is provided for free [4]. Developers can also pay a fee to obtain full source code and the ability to customize the library as needed. PhysX is used by Unity and is the engine I will be working with in this thesis.

1.3.2 Havok

Havok is very popular commercial physics SDKs. It's developed by Irish company Havok that was acquired by Intel in 2007 which sold it to Microsoft in 2015. It has been used by leading game developers in over 400 titles [5]. A great example of Havok powered game is Spintires as seen in 1.2. Developers from Oovee Game studios pushed Havok physics to its limits and created an off-roading simulation with amazing mud and water physics [6]. Havok is comprised of a core collision physics engine, plus a number of optional add-on products including a vehicle physics system, a system for modelling destructible environments, and a fully featured animation SDK with direct integration into Havok's ragdoll physics system.

1.3.3 Bullet

Bullet [7] is a free and open-source physics engine. It has been used in several commercial games and movies. It uses a collision detection system that is

Figure 1.2: Excellent mud physics powered by Havok in Spintire by Oovee Games Studios.

integrated with its dynamics simulation. However, it is possible to use the collision system as standalone or use it integrated into other physics engines. Bullet offers support for rigidbody and soft body simulation. Soft body support includes cloth, rope, and deformable objects.

1.4 Unity game engine

Unity is one of the most popular game engines nowadays. Thanks to its intuitive and easy to learn toolset, Unity is used by many developers from hobbyists to professionals. It is used to develop 2D and 3D games for web, mobile, desktop, and consoles. In terms of scripts, Unity uses C# and Javascript. More about Unity development can be learned in [8].

Every test created for this thesis is made with Unity editor. It lets you create simple games in few hours, as most things are already done for you. World in Unity editor is made of game objects. Each object has a transform component that stores its position in the world. You can attach various components to this game object: lighting effects, physics, mesh, texture, custom scripts and more. After pressing play editor starts the main loop and launches the game.

1.4.1 Unity main loop and scripts

Game, like animation, is a continuous stream of frames. Each frame, a game loop script is executed that directs what's happening on the screen. Usually there are 30 or 60 frames per second and more frames means better performance. The game loop scrip is contained in MonoBehaviour base class which all scripts must derive from. Game loop calls many important functions including start, update, fixed update, late update, rendering functions and decommissioning functions.

At the start of a game, a start function is called which contains all required prerequisites. You can also use start function in a custom script. For example to allocate an array or to find another game object needed for execution of a script.

The update function is called every frame and contains game logic. This means that if you make a multiplayer shooter and tie shooting to update function, the player with lower framerate will shoot fewer bullets. This problem can be avoided by using *Time.deltatime* property and changing "bullets per frame" to "bullets per second" in your game.ewer

The fixed update is a special function used for physics. It is called every x milliseconds based on timestep value. The fixed update can be called several times during an update function or not at all if FPS is too high. In case of FPS being too low, it is possible to limit the number of fixed update calls in the Maximum Allowed Timestep setting in Time Manager.

After the update function, a late update function is called for the last bit of game logic and then begin the rendering functions. At the end of a frame, an end of frame and pause functions are called.

Before game ends, decommissioning functions are called and the game is closed. All functions in execution order can be found in documentation [9].

All scripts in Unity derive from *MonoBehaviour* base class as can be seen in its manual page [9].

1.5 Physics in Unity

The physics in Unity is supplied by integrated NVIDIAs PhysX engine. It's split into two parts: 2D Physics and 3D Physics. Although they can coexist together in the same scene, they are two separate entities that cannot communicate among themselves.

1.5.1 Rigidbody

Objects in Unity are not affected by physics by default. We don't want to move the road, trees, walls. We want to move a car on the road. To accomplish that we need to assign a rigidbody component to it. Such an object will immediately respond to gravity and is movable by forces. Gravity is a global force in unity and all rigidbodies respond to it according to their mass and drag properties. Force is an easy way to move rigidbodies in Unity. We shouldn't move rigidbodies by changing their transform properties. If we want to, we have to use kinematic rigidbodies that can exist without having their motion controlled by the physics engine. When moved, rigidbodies appear as active in the physics simulation. Unmoved rigidbodies change into an inactive state to save CPU usage.

1.5.2 Colliders

Collider component defines the shape of an object for the purposes of physical collisions. Colliders can be split into primitive colliders and mesh colliders. Primitive colliders are the box, sphere, and capsule colliders. Mesh colliders match the shape of the object's mesh exactly as can be seen in 1.3. Mesh colliders can't collide with other mesh colliders by default. This can be fixed by marking one of them as convex. They are much more processor-intensive and impact performance in a huge way. Mesh colliders should be used sporadically, mostly on static objects. It is much more effective to use compound colliders which are combinations of primitive colliders.

Colliders can be added to an object without a rigidbody component to create floors, walls, and similar motionless objects. Such colliders are known as static. Repositioning static colliders will hugely impact game performance. Colliders applied to an object with rigidbody are dynamic and should be moved by forces. Collider can be also added to a kinematic rigidbody. A moving kinematic rigidbody will apply friction to other objects and will wake up other rigidbodies when they make contact. A good example is doors, they normally act like immovable obstacles, but can be open/closed.

Physical materials allow us to add real physical properties to certain colliders. We can add slippery material to the floor and make it into ice, bouncy material with a lot of friction to a rubber ball, or we can create new materials with custom properties.

1.5.3 Deformable objects

Deformable objects are objects that can change their shape by applying forces on them, clicking on them or by other means. Unity doesn't support deformable objects directly, but it does support skinned mesh. The skinned mesh is a type of mesh that can be deformed by predefined animation sequences [?]. It uses bones that are invisible objects inside the mesh that are attached to each other to form a skeleton. This skeleton is powered by animation that dictates how it is supposed to move. If the skeleton moves the mesh bends itself accordingly. It is also possible to attach rigidbodies to this skeleton and power it by the physics engine. This is often used to create a ragdoll

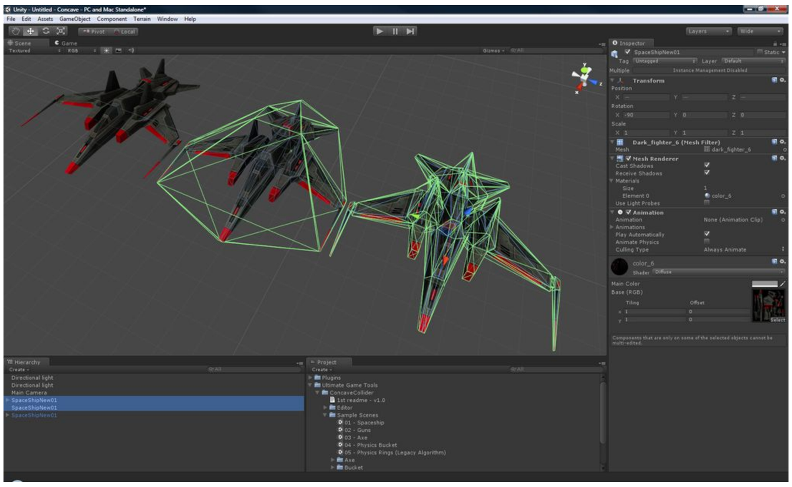


Figure 1.3: Object without collider(left), one with custom primitive collider(middle) and one with mesh collider(right). Picture is from unity asset store page for Concave Colliter tool for unity.

effect, where characters arms and legs bend after throwing it or striking it with an explosion. The community has also created solutions for deformable objects in Unity. Some of these solutions are the realtime mesh deformation package [10] that works with collision and other Unity physics systems, or a tutorial that teaches you how to create a stress ball [11].

1.5.4 Joints

Joint is a special type of constraint that connects two objects with a different force effect. There are 5 types of joints: fixed, spring, hinge, character and configurable joint.

Fixed joint is used to lock two rigidbodies at a fixed distance and orientation.

Spring joint uses an invisible spring to keep two rigidbodies at the same distance. When they get separated the spring pulls them back together with a predefined force.

Hinge joint keeps two rigidbodies at a fixed distance with customized orientation. Useful for doors, chains, pendulums, etc. Hinge joint can be customized to apply a spring and a motor to the objects. The motor spins the joint around its axis with a predefined force.

Character joint is used in ragdoll to connect its body parts. It has more options to set up constraints than previous joints. Options like twist

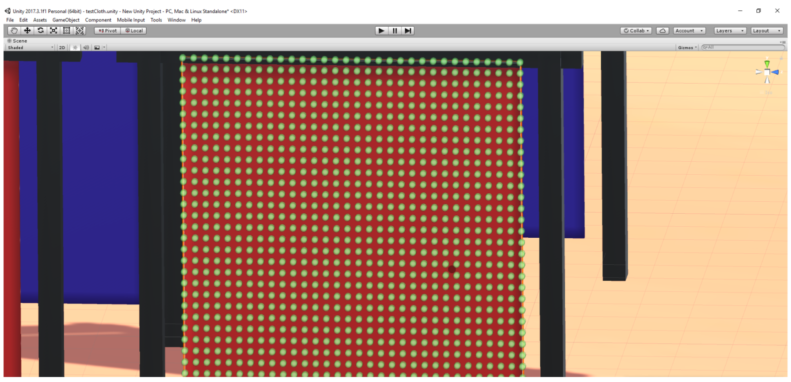


Figure 1.4: Visual editor for cloth self collision particles. Green dots represent self colliding particles.

limit, swing limit and even break force that specifies force at which the joint breaks.

Configurable joint is a highly customizable type of joint. It incorporates constraints and options from other existing joints. It is mainly used for complex simulation.

1.5.5 Cloth

The cloth component is designed to simulate fabrics using the physics engine. It is specially made for character clothing and only works with skinned meshes. The component doesn't normally react to the world and the world doesn't react nor recognize the cloth component. It's possible to manually add objects with the sphere or capsule collider from the world to the cloth component. The cloth component will react to them as they come into contact with it, but they won't receive any force from the cloth component.

The cloth component applies a constraint to every vertex from the cloth mesh. These constraints dictate how far can mesh vertex travel from its initial position during simulation. It is possible to set these values in a visual editor that highlights each constraint on the mesh.

Unity allows cloth self collision and intercollision. Cloth component adds collision particles to each skinned mesh. These particles collide with each other and create the effect of cloth self collision. Particles can be set up in the visual editor, see 1.4. Intercollision is a collision between two distinct objects with cloth component. It uses similar particles as for self collision, but has to be also activated in Physics Manager.

1.6 Physics optimization tools

1.6.1 Profiler

Unity editor offers the profiler window to inspect performance of games. The profiler records performance and shows it in the form of graphs and various statistics. It has several modes detailing CPU or GPU load, rendering, audio, physics and more. Each mode offers information about what kind of object or function is computed at that frame. For example, CPU mode shows a graph detailing CPU load per frame split into different areas like physics, rendering, etc. It is also possible to view computing time spent per frame on various functions(e.x. *FixedUpdate*).

Physics mode displays a number of active non-kinematic rigidbody, active kinematic rigidbody, and total rigidbody components. A number of static colliders, trigger overlaps, active constraints and pairs of contacts. Numbers might not correspond to the exact count of objects in the scene as some components are processed at a different rate.

The profiler can save the results into a byte data file that can be then loaded and viewed again only in the profiler. Not only it can be used in the editor, the profiler can also measure data from built games on the same PC or different machines including iOS and Android systems via WiFi connection.

1.6.2 Physics debugger

Physics debugger is a visual tool that highlights physics components in the scene. The physics debugger paints rigidbody components and colliders in different colors. It is then easier to spot which colliders should and should not be touching. The debugger can be seen in 1.5.

1.7 Current benchmarks

In this section, I look into current benchmarks and works that compare physics engines. Since Unity uses a modified version of PhysX engine, most discussed articles are over ten years old and work with a standalone version of PhysX, I will not compare nor discuss their results. However, it is worth to study aspects of the engines that were tested, and what kind of testing methods were used.

1.7.1 Tested physics properties

Physics engine properties tested in reviewed publications can be split into two categories[12].

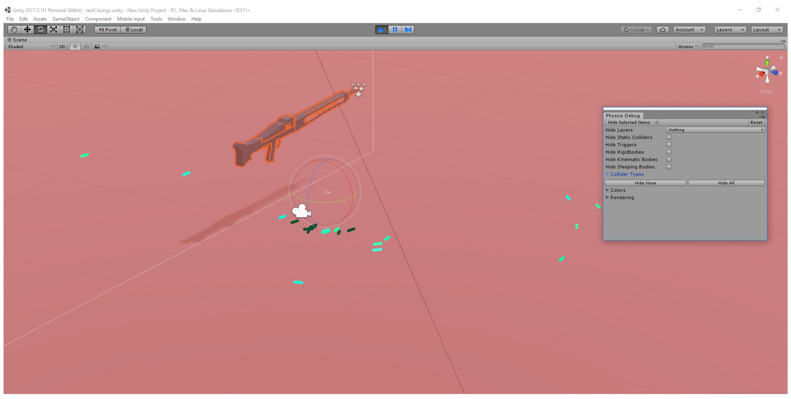


Figure 1.5: Picture of the physic debbuger. Static collider is highlighted by red color, active rigidbodies by light green and inactive have dark green color.

The first category covers properties important for the real world physics:

- Integrator calculates a position of an object after being struck by force. It has to consider many factors such as velocity of the object, possible collisions, constraints, etc. A better integrator can represent real world forces more accurately.
- Restitution is a force that object retains after a collision.
- Friction is a resisting force that affects the sliding of an object.
- Gyroscopic force stabilizes objects position and angular velocity.

In games, the consistency of calculations is important, but exact replication of real world physics isn't the highest priority.

The second category includes properties important for game development:

- Constraints lock object in certain axis or direction. Main types of constraints are prismatic which restrict object from rotating and only allow movement along a specified axis, revolute that allow rotation around one axis and spherical constraints that simulate rotation around a point. You don't want doors to move and block corridors, or wheels spinning away and crashing the car.
- Representation of objects in physics simulation. Objects are represented by colliders in the form of spheres, capsules, rectangles, etc.

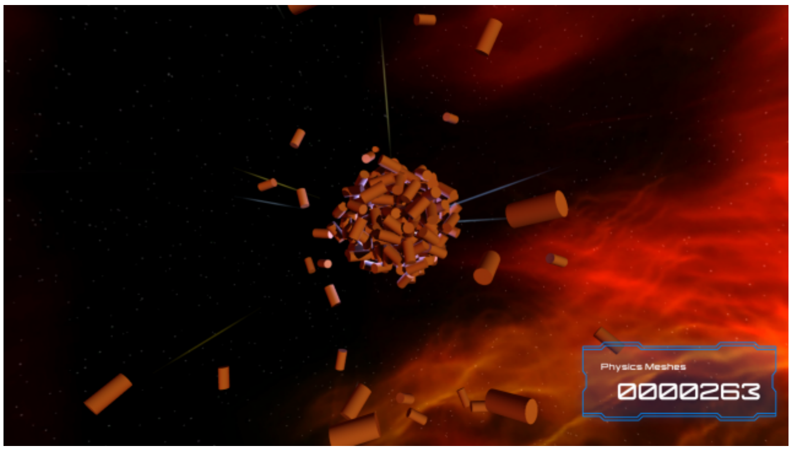


Figure 1.6: Unity WebGL benchmark. Courtesy of Unity Technologies.

- Collision system determines accuracy and detectability of intersections between colliders. Good collision system is essential in games, as one missed collision could render the entire game unplayable.

1.7.1.0.1 Unity official benchmarks Unity has several official benchmarks that propagate the engine on their website. Most of them are heavily focusing on graphics and animations, except for their WebGL benchmark[13]. In this simple benchmark, there are three scenarios that test 3D physics. Each scenario tests how many colliding objects can your computer handle at the same time. They use objects with sphere, cube or mesh colliders that are constantly spawning and accumulating in the center of screen1.6. The benchmark stops when FPS drops below a certain threshold.

1.7.1.0.2 Boeing and Bräunl compared a number of physics engines in 2007[12]. They used physics engine abstraction system PAL that provided them various interfaces to conveniently implement five scenarios. These scenarios test integrator, constraint, material, collision and stacking properties of engines.

The first test evaluates integrator. It involves a sphere dropped on a ground with a gravitation set to -9.8m/s, and timestep set to 10 ms. The position of the sphere is then compared to various ideal cases from the real world.

The second scenario tests material properties. It is split into restitution test and static friction test. In the restitution test, a sphere is dropped on a static box and spheres bounce is recorded and compared to ideal situations. Static friction test includes box sliding on a skewed plane. The recorded speed of sliding is again compared to ideal cases.

The third scenario tests constraint stability. The scenario uses a chain of spherical links connecting several spheres where most left and right spheres were attached to boxes. The test ran for 20 seconds. In the end, the position of spheres was compared to their initial position. Constraint stability is important to both games and simulation.

The fourth test analyzes collision system. Several spheres are dropped into an inverted pyramid. Test counts the number of spheres collider intersections with the pyramid. An error is recorded, whenever a length of spheres radius is lesser than its distance from the pyramid's wall.

The fifth scenario tested the efficiency of physics engine to handle stacked objects. Several boxes were dropped in a stack on top of one another. In reality, such stack should collapse.

1.7.1.0.3 A. Seugling and M. Rollin have done a similar comparison in 2006 [14]. They made tests for friction, gyroscopic force, restitution, constraints, and collisions. Some of the tests they did are the same as the previous article, so I will only talk about the differences.

The first scenario tests gyroscopic force. A box with starting angular velocity is placed in a world without gravity. After few seconds angular energy and momentum are compared to initial values.

The second test measures accuracy of physics calculations. It involves a pendulum with constraints on stick and ball. The ball is swung, velocity and retention are observed and compared to the real world calculations.

The third test observes colliders. It's split into three parts: primitives with static non-convex triangle mesh, convex triangle meshes with each other and non-convex triangle meshes with each other. Scenarios are observed and graded according to how well they simulate reality.

1.7.1.0.4 Erez et al. created quite different tests than in previous articles with one specific requirement in mind. They were looking for physics engine best suited for robotics [15]. Scenarios test collisions, restitution, friction, joints, performance and multibody dynamics.

The first one is a grasping test. A robotic hand is grasping a capsule. The hand is powered by joints and fixed spring-dampers.

The second test is a humanoid ragdoll model thrown on the floor. The model then wiggles from forces applied to its joint. This tests restitution and friction from contact with the floor.

The third scenario tests constraints. It involves a planar chain made from five bodies and five hinge joints. The test is initialized with forces applied to joints which then slowly stabilize. Path of the tip of the chain is observed and recorded.

The fourth scenario tests performance of the collision system. Several capsules are dropped on the floor and performance is recorded.

1.7.1.0.5 Summary In summary, I would say that reviewed papers provided interesting ideas how to test physics simulation. I took inspiration from the collision system test in the first paper. Instead of collisions with the pyramid, I focus on collisions between spheres in the collision accuracy test.

Chapter 2

Benchmarks

In this chapter, I talk about benchmarks that have been made for this thesis in Unity editor. I made several benchmarks that focus on physics properties important for game development. The benchmarks themselves represent common situations in games and are combined into a video game which is the last test in the chapter. Results if the benchmarks can be found in the next chapter.

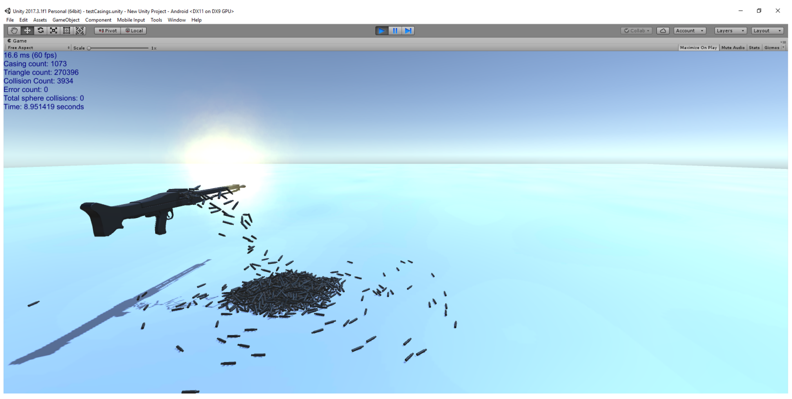
2.1 Performance test of the collision system

Performance test of the collision system is classic physics engine test. The idea is that with increasing number of colliding objects on the screen, the time needed to compute each frame rises. To test this hypothesis a simple test was created. A firing machine gun is releasing casings on the floor which create a big pile of colliding objects2.1.

At the start, the machine gun is prompted to fire and release casings with unrealistic speed to shorten the test. Casings are created at the side of the gun and thrown in a similar direction with *AddForce* function. Casings fall on the floor and create many collisions with it and each other. The test stops when FPS slows down below a certain threshold.

The test has several modes to find out how different settings affect performance:

- The first mode is the standard. All modes mentioned further keep the same setting as this one with other modifications. Casings have capsule collider, timestep is set to 10 ms, casing mesh has 252 triangles and the test is run in the editor.


Unity 2017.3.1f1 Personal (64bit) - testCasings.unity - New Unity Project - Android <DX11 on DX9 GPU>
File Edit Assets GameObject Component Mobile Input Tools Window Help
Pivot
Local
Game
Free Aspect
Scale
1x

```
16.6 ms (60 fps)
Casing count: 1073
Triangle count: 270396
Collision Count: 3934
Error count: 0
Total sphere collisions: 0
Time: 8.951419 seconds
```

Figure 2.1: Snapshot of the performance test. A machine gun is releasing casing on the ground which creates a heap full of collisions.

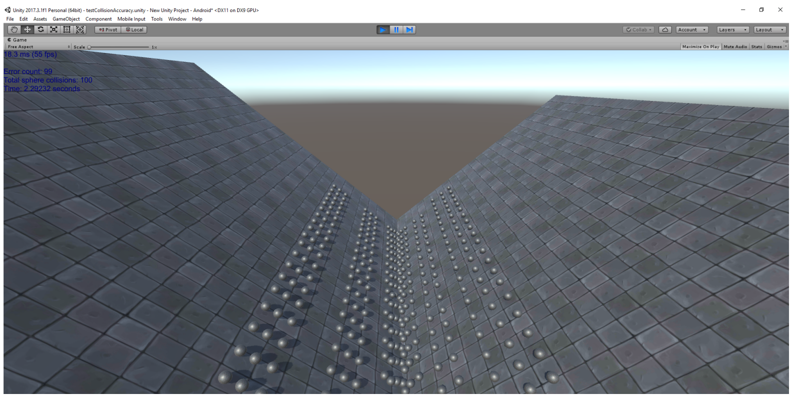
- In the next mode casing collider is changed from capsule to mesh collider. This should affect the performance in a bad way.
- In the third mode, half of the casings have capsule collider and other half have mesh collider. Colliders change by rotation of one.
- The fourth mode has adaptive force setting enabled in Physics Manager.
- The fifth mode modifies the timestep of the physics simulation. Timesteps tried are 20 ms, 40 ms, and 80 ms.
- In the last mode, the standard test is built into an application and ran on several different platforms.

2.2 Accuracy test of the collision system

Purpose of this test is to determine how accurately can Unity stop colliders from intersecting during a collision. 200 identical spheres are placed upon wide sides of a triangular prism. Spheres roll down the sides and clash in the middle2.2. All sphere collisions are recorded. On each collision, the distance between two centers of colliding spheres is compared to sphere radius times two. If the distance is shorter, an error is recorded.

2.3 Fire accuracy test

There are two popular types of firing systems in shooter games - hitscan and physics based ballistics simulation. Hitscan weapon fires a ray in a straight line hitting the first target the ray goes through. The other system utilizes physics engine to simulate bullet drop so player has to


Unity 2017.3.1f1 Personal (64bit) - testCollisionAccuracy.unity - New Unity Project - Android* <DX11 on DX9 GPU>
File Edit Assets GameObject Component Mobile Input Tools Window Help
Pivot
Local
Game
Free Aspect
Scale
1x

```
18.3 ms (55 fps)

Error count: 99
Total sphere collisions: 100
Time: 2.29232 seconds
```

Figure 2.2: Snapshot of collision accuracy test. Spheres are rolled down the sides of a triangular prism. They collide in the middle and collision error is measured.

often fire slightly above his target. This requires the physics calculations to be always consistent.

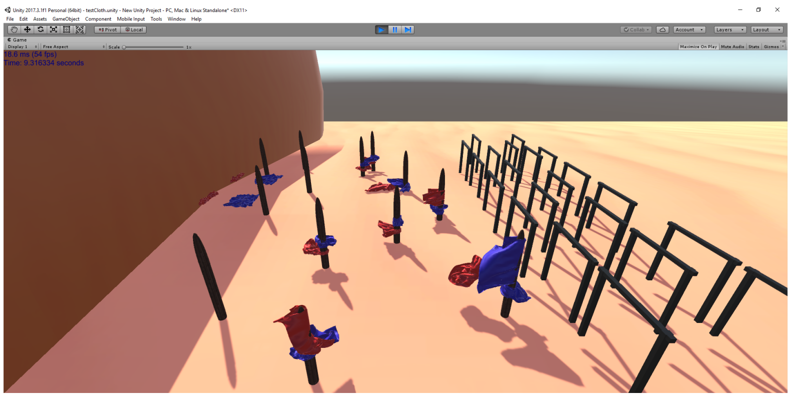
This test is split into two parts. First part tests the consistency of physical simulation. A gun is fired into a wall thousand times, the contact point is recorded and later compared to other points. The second part investigates how fixed timestep affects hit accuracy. Gun is firing bullets using *AddForce* function with default force mode. Bullets are fired in the same direction and with the same force.

2.4 Cloth component benchmark

Cloth component is used with skinned mesh to represent character clothing, flags and the occasional piece of fabric in games. This benchmark showcases its capabilities in Unity, how it interacts with the environment and how it affects performance. The benchmark consists of rugs hanging from racks. An increasing wind force is applied to the rugs which later tears them from racks. Torn rugs interact with poles and wall in front of them. In the end, the wind is stopped and rugs fall on the floor 2.3.

Rugs use cube mesh with 2304 vertices and 4092 triangles. Poles, wall and floor use capsule colliders. There are 21 rugs and 13 poles. Wind is simulated by increasing the external acceleration value in each rug's cloth component. When this value reaches a certain point, all constraints in cloth component are overridden and the rug is torn from the rack.

The benchmark is performed in three iterations:


Unity 2017.3.1f1 Personal (64bit) - testCloth.unity - New Unity Project - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Mobile Input Tools Window Help
Pivot
Local
Game
Display 1
Free Aspect
Scale
1x

```
18.6 ms (54 fps)
Time: 9.316334 seconds
```

Figure 2.3: Snapshot of the cloth component benchmark. Rugs in the picture collide with poles and later with the ground and the wall.

- The first iteration is the standard and works as described in the previous paragraph.
- The second iteration adds self collision to the cloth component. All collision particles are activated.
- The third iteration uses self collision and intercollision with all collision particles activated.

Time per frame is recorded and CPU usage in the profiler is observed.

2.5 Deformable object benchmark

This benchmark focuses on objects deformable by collision. Deformation is not inherently a physics component calculated by physics simulation, but it's certainly a physical effect that one would like to have in his game. It involves a gun firing at a wall. Wall gets deformed 2.4 whenever struck by bullet according to its velocity. Since only physics component that supports deformable objects is cloth, I created a custom script that changes objects mesh at runtime.

The script is inspired by the second script in mesh scripting class in Unity documentation [9]. When the wall collides with the bullet, the script takes the collision point and its collision force. If the collision has sufficient strength, the wall gets deformed. A new set of vertices is created and deformed according to the impact force. The set is assigned to wall mesh and mesh bounds are recalculated. Finally, wall's mesh collider is destroyed and a new one with new mesh is created in its place.

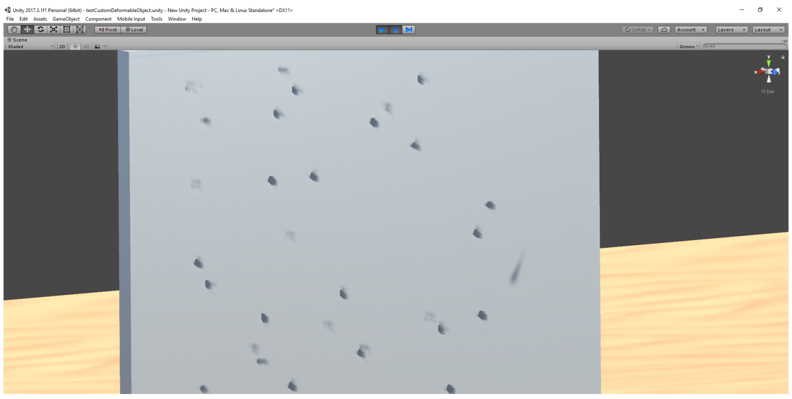


Figure 2.4: Snapshot of Deformable object benchmark. Holes in the plate are created upon collisions with bullets.

Benchmark works similarly to fire accuracy test. Gun fires a bullet with *AddForce* function. Bullet has a mesh with 448 vertices. Several meshes with 1144, 2304, 5824 and 12474 vertices are tried for the wall. Both bullet and wall use mesh collider. Framerate and visuals are later compared.

2.6 Physics based animation benchmark

The purpose of this benchmark is to recreate animation with physics components and research the capabilities of Unity in this topic. This can lead to interesting results, as such animated objects react to their environment. For this benchmark, I present a race of walking robots.

Robots are powered by joints. One robot has a Torso with configurable joint and 6 legs with each having 2 hinge joints. Upper leg has a motor that gets activated every 0.5 seconds. Motor tries to spin the leg but its stopped by an angular constraint and moved back with a spring. The lower leg is attached to the upper leg and moves with it. It has an angular constraint and spring that keep it in place. The configurable joint in torso has a y constraint that keeps it from moving sideways. This together moves the robot towards its target. The robot doesn't fall thanks to the strong springs in its legs.

Robots start on one side of the map and clumsily move to the other. They are slowed at stairs which they are capable to scale. Both of those create an uneven setting where different robot wins every time.

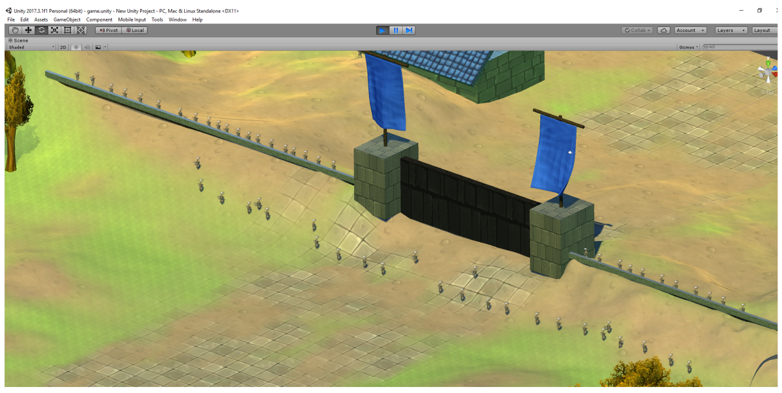


Figure 2.5: Picture of the defenders, positioned on the ramparts, with all types of units.

2.7 Game benchmark

The last benchmark combines previously tested components into one simple game. Isolated tests are good for showcasing the performance of individual physics components but don't paint the whole picture. The benchmark is designed to show how these components interact and perform in a game scenario. It's split into 2 parts. A real-time strategy game where player has to defend his base against incoming hordes of enemies. This demonstrates how components perform in a longer period during several waves of enemies. The other part is a scripted scenario where a single predefined wave of enemies attacks a base defended by entrenched forces. This type of scenario is better suited for CPU load and frame rate analysis.

2.7.1 Real-time strategy game

To win the game, player has to defend his base against several waves of enemies. The base is protected by short ramparts and a gate as seen in 2.5. Player gets a few units at the start that he can position at the ramparts. He can train more units from the side panel. Unit's health and damage can be upgraded. Units and upgrades cost gold which can be attained from killing enemies or surviving the enemy wave. The end wave gold income can be increased by investment. The game ends when player survives all waves or his gate is destroyed by the enemies.

There are four types of enemies. A basic enemy unit is an orc with a rifle that has low damage, low rate of fire but great accuracy. Orc is the most common unit and will be present in every wave. Mutant is

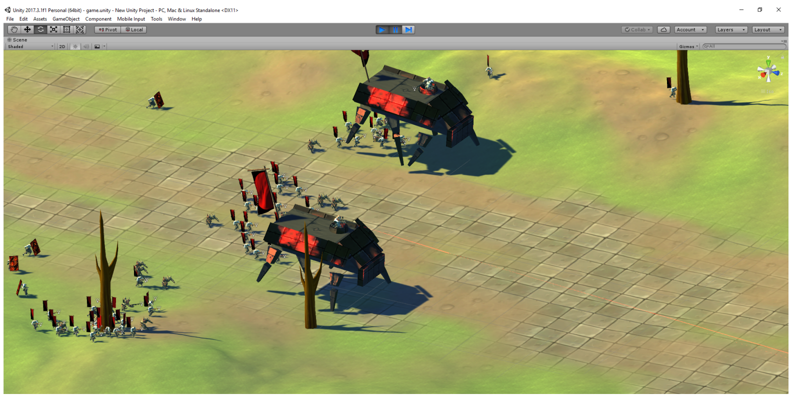


Figure 2.6: Picture of the incoming enemy wave with all types of enemies.

an improvement upon the orc. It wields a machine gun with a greatly increased rate of fire, has more health but lesser accuracy. The third enemy option is a sapper who is equipped with a shield and bombs. He goes straight for the gate and does the most damage to it. Last but not least, an armored walker is the most dangerous enemy unit. It carries a machine gun turret on top that has better chance to shoot defenders hidden behind ramparts. All types of enemies can be seen in 2.6

Player can recruit soldiers with four types of equipment. All types of units have the same amount of health, while damage varies, depending upon the equipped weapon. Soldiers can carry rifles and machine guns that perform similarly to their enemy counterparts. The third option is an anti-tank rifle that deals more damage to armored targets. Lastly, medics carry medical bags that can be thrown on friendly units to heal them.

There are more paths to victory. One player might focus on basic riflemen with early upgrades. Other can train few machine gunners with a lot of medics to heal them. The greedy player can recruit just enough soldiers to barely beat the wave, invest rest of his money into income and become unbeatable in later rounds. One important skill shared for all strategies is to pull units under heavy fire or those with low health.

2.7.2 Scripted scenario

The technical of the benchmark is a scripted scenario where 1 wave of enemies attacks the defenders on ramparts. The scenario is used to compare the CPU usage of *FixedUpdate*, *PhysicsSkinnedClothFinishUpdate"* and the overall time per frame.

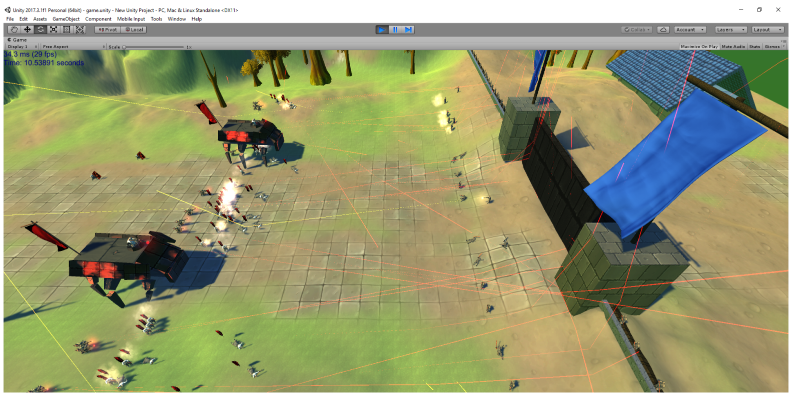


Figure 2.7: Picture of the clash of forces. Bullet trajectories are highlighted by sharp colors.

Benchmark is being run in Unity editor with the timestep of the physics simulation set to 10 ms. A preselected wave of enemies is spawned with 50 orcs, 15 mutants, 5 sappers and 2 walkers. Enemies are randomly spawned in a designated area. Enemies battle versus 47 riflemen and 14 machine gunners. 26 soldiers are on the ground in front of the gate, rest is positioned on the ramparts. Benchmark is stopped at an arbitrary point near the end of the battle when most combatants are no longer operational, and time per frame returns to initial value. All desired values are measured at this point and no more framerate spikes are expected in the future.

2.7.3 Physics components

The benchmark makes use of all physics components used in previous benchmarks. Components are featured in similar and new situations.

Units have 2 states with distinct representations. In the alive state, units have rigidbody component and capsule collider. They are moved around the map with *MovePosition* from rigidbody class. In the dead state, units lose these components and gain a ragdoll which consists of 8 capsule colliders, 2 box colliders, and 1 sphere collider all connected with character joints. Ragdoll freely falls to the ground simulating a corpse. The armored walker is an only exception, as it is the animated robot from the sixth benchmark.

Weapons fire bullets using *AddForce* function with default force mode. Bullets have a capsule collider, and their rigidbody component has a continuous dynamic collision detection to ensure that each bullet hits its mark. Friendly bullets have yellow and enemy bullets have red trail renderer to better showcase their trajectories. Trajectories can be seen in 2.7.

Cloth component is used in flags placed on walkers, orcs backs and gate towers. Orcs flags use a mesh with 384 vertices and 572 triangles, while walkers and towers use mesh with 304 vertices and 4092 triangles. A flag can collide with its 2 capsule collider poles and with itself.

The deformable mesh script is used for the gate deformation as it is struck by bombs thrown by sappers. It has a mesh with 2304 vertices and 4092 triangles.

2.8 Assets

An asset can be anything from a mesh to a script. It is a basic building block of any game. Assets can not only be created in Unity editor but also come from different sources. It can be a model from blender or audio file recorded on your phone. In the created benchmarks, I mainly work with my own assets. Scripts were created in Mircosoft's Visual Studio. Some of the models like the machine gun and the walker were modelled in Blender.

Most of the outside assets were imported from the Asset Store. This includes the hand painted materials [16], surface stone textures [17], trees [18] and RTS camera component [19]. Fantastic free live captured animations with models were imported from Mixamo [20].

Chapter 3

Results

In this chapter, I present results of the benchmarks. More about benchmarks can be found in the previous chapter. Most of the results are measured in the editor, so the framerate isn't as high as it would be in a built version. The reason for this is that I use a custom script to extract data from the profiler. The script makes use of the *UnityEditor* library and Unity couldn't build the scene with such script in it.

3.1 Performance test of the collision system

This test involves a machine gun releasing casings which then create a heap on the ground. The test is split into several versions.

Graph 3.1 shows standard version of the test with capsule colliders. The graph compares overall framerate of the test run in the editor, built version and the CPU usage of the version run in the editor. Time per frame is above 100 ms during initialization and then stabilizes below 20 ms per frame. With more casings and collisions, time per frame rises until it reaches maximum allowed value. The benchmark peaks at around 2.4K active rigidbody components and 10K collision pairs. Looking at the profiler, *FixedUpdate* function takes little more than 80% of the CPU load at this point. After the peak, time per frame lowers a little, since there are no more casings falling on the heap. Casings in the heap are still active and wiggle a bit. In the end, all active rigidbody components in the heap are put to sleep and time per frame returns to initial value.

Graph3.2 compares different collider options, standard version with adaptive force enabled and higher timestep. The last mentioned seems to have the biggest impact on performance. This, however, comes

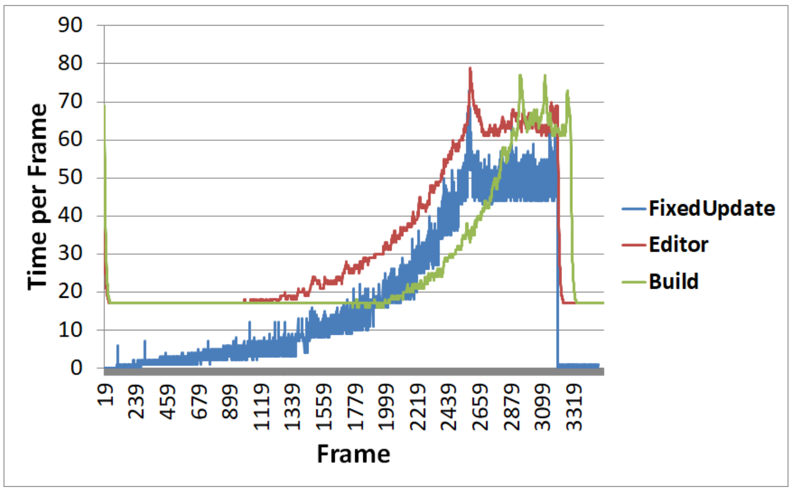


Figure 3.1: Graph detailing time per frame change with increasing number of casings. Graph compares standard version of the casings test run in editor, built version and CPU usage of the version run in editor.

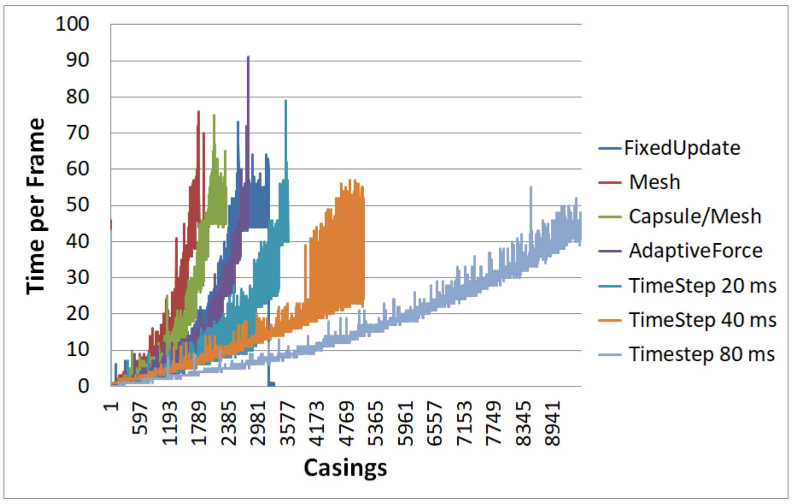
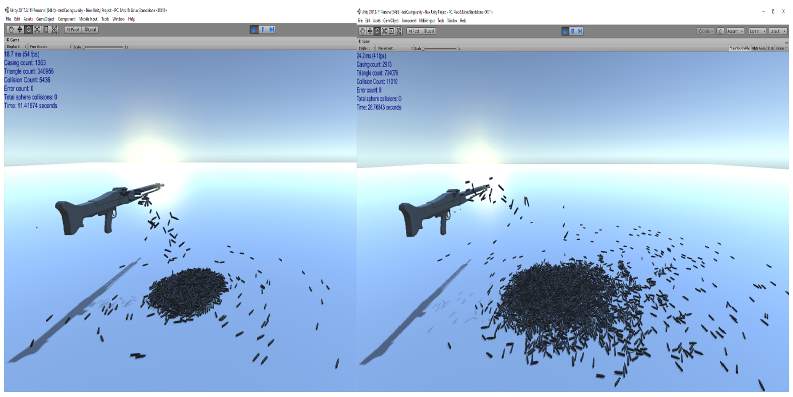


Figure 3.2: Graph comparing different versions of performance test.



```
18.7 ms (54 fps)
Casing count: 1353
Triangle count: 340956
Collision Count: 5436
Error count: 0
Total sphere collisions: 0
Time: 11.41874 seconds
```

```
24.2 ms (41 fps)
Casing count: 2913
Triangle count: 734076
Collision Count: 11010
Error count: 0
Total sphere collisions: 0
Time: 26.76843 seconds
```

Figure 3.3: Casing heaps created by collision performance test. Left heap is created with 10 ms and right with 20 ms timestep. Right heap is bigger than left as casings are thrown further with 20 ms timestep.

Platform	Time	Casings	Collisions
PC	34	3144	13350
Notebook	19	2059	8593
Tablet	10	658	2370
Mobile	10	543	1753

Table 3.1: Built standard iteration of Collision performance test on different platforms. Time is shown in seconds. Test uses capsule colliders and timestep is set to 10 ms.

at a price, with increased timestep, physics simulation is calculated less frequently and physics components start to stutter a bit. Test with 80 ms timestep looked like it was running at 15 FPS. Another interesting effect3.3 is that increasing timestep also increased the force which threw casings from the gun. I had to lower the velocity of throwing by half to get similar heap in the test with 20 ms timestep.

Table 3.1 shows results of built standard iteration of casings test ran on different platforms. Platform specifications can be found in 3.2.

With all things considered, capsule colliders with around 40 ms timestep and adaptive force enabled seems to be the best option to handle a heap of colliding objects. In a game scenario, casings could be deleted after a short while to ensure a low impact on framerate.

PC	i5-7600k 3.80 GHz; 16 GB RAM; NVidia GeForce GTX 1070; Win 10 64bit; NR: 1920 x 1080, TR: 1920 x 1080
Notebook	i7-4710HQ 2.5 GHz; 8 GB RAM; NVidia GeForce GTX 860m; Win 10 64bit; NR: 1920 x 1080, TR: 1920 x 1080
Tablet	Quad-core 2.2 GHz Cortex-A15; 2 GB RAM; ULP GeForce Kepler; Android 6.0.1; NR: 1920 x 1200, TR: 1920 x 1080
Mobile	Quad-core Max 1.8 GHz; 3 GB RAM; Adreno 530; Android 7.0; NR: 1920 x 1080, TR: 1920 x 1080

Table 3.2: Benchmark platforms. NR - native resolution, TR - tested resolution.

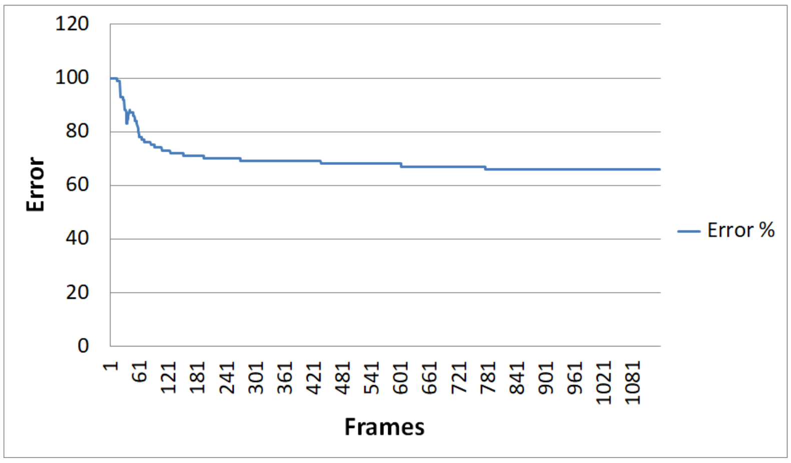


Figure 3.4: Graph showing proportional error of sphere collisions. Error is calculated as total count of collisions divided by error collisions. It occurs when centers of 2 spheres are closer than their radius sum.

3.2 Accuracy test of the collision system

In this test, 400 spheres are clashing to test the accuracy of their collisions. Graph3.4 displays a percentage of collisions where colliders of spheres were intersecting. The Percentage is counted as a total count of collisions divided by a total count of errors times 100. The graph starts at 100% due to high velocity behind initial clashes. Error count dwindles as spheres start to shuffle around creating low velocity collisions.

Times increased	1x	2x	3x	4x	5x
Timestep Default	4.4	14.5	16.4	17.1	17.4
Force	4.4	14.6	16.5	17.1	17.4
Timestep VelocityChange	4.4	4.3	4.2	4.1	4.1

Table 3.3: Bullet was fired at a wall with different physics simulation timesteps. Numbers represent y value of contact point 3D vector. In the first row timestep increased respectively, in the second a force behind bullet was increased, and a different type of force was used with increasing timestep in the third row.

3.3 Fire accuracy test

For the first part of this test, a gun is fired for thousand times at a wall to determine the consistency of calculations. Every shot is compared to the first one. Not a single one has a different contact point with the wall than the first.

In the second part, a gun is fired with fixed timestep set to different values. Table3.3 shows measured values. Contact point position vector didn't change in the x nor the z direction, thus only the y direction values are displayed. The first row represents gun fired five times with same force of 25 starting with 10 ms timestep and ending with 50 ms. In the second row, a gun was fired five times with the same timestep of 10 ms but with five different forces from 25 to 125. Values in the first and second row are almost identical. This strengthens the hypothesis from performance test that multiplying timestep by x increases the force of *AddForce* function x times. The possible way to battle this effect is to change force mode of *AddForce* to *VelocityChange*. This adds instant velocity change to the bullet, ignoring its mass. The third row in table3.3 shows how changing timestep impacts such solution.

3.4 Cloth component benchmark

This benchmark showcases the cloth component. Rugs are torn from racks and collide with poles, wall, and floor. The benchmark is split into three iterations: standard, with self collision and one with both self collision and intercollision.

Graph 3.5 shows time per frame for each iteration. The first iteration peaks at the moment, when torn rugs hit poles. Time per frame gets shorter as rush slowly hit the wall. The second iteration has similar curve until rugs hit the wall, there time per frame peaks as rugs are pressed against the wall and self collide the most. The third iteration

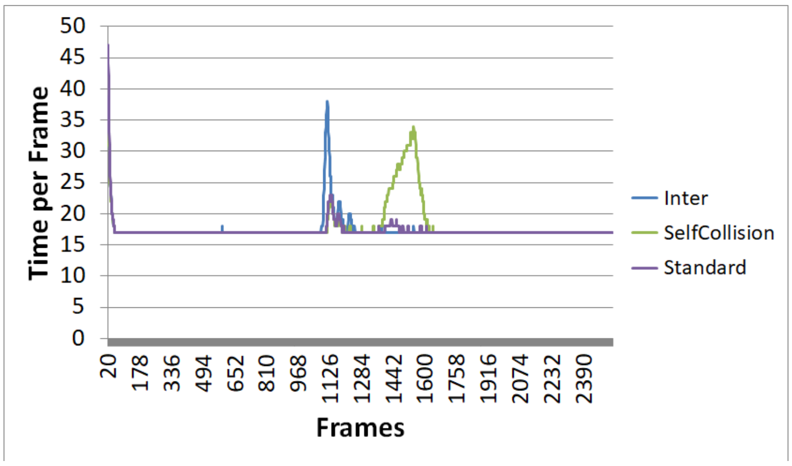


Figure 3.5: Graph of the cloth component benchmark. Graph shows overall framerate difference between active cloth self collision, active cloth self and inter collision and standard with neither of the 2.

peaks when rugs hit the poles and at the same time collide with each other. Some of the rugs get stuck on the poles, so there's not as long time per frame as in the second iteration during wall hitting phase.

Looking at the profiler, cloth simulation is calculated in *PostLateUpdate.PhysicsSkinnedClothFinishFunction* which starts at 30% of CPU load with 5 ms per frame before rugs are torn. It can reach around 90% of CPU load with 40 ms per frame during pole hitting phase in the third iteration.

Overall cloth component serves its purpose good enough. It is good for character clothing and the occasional cloth that characters interact with. Rug meshes in this test were exaggerated to better show framerate difference. Similar visual results could be achieved with much less detailed mesh and much less frame budget. Problems start when you want your cloth objects to interact with the environment. Fact that cloth component can interact only with capsule and sphere colliders makes this task rather difficult. I solved this problem by creating the whole environment from capsule colliders which is not optimal. Other possible solutions can be found on Asset store(e.x. Obi Cloth[17])

3.5 Deformable object benchmark

Benchmark of the custom deformable object. A gun is shooting bullets at a wall that create holes upon impact. The benchmark was run 4 times with 1144, 2304, 5824 and 12474 vertices for wall mesh.

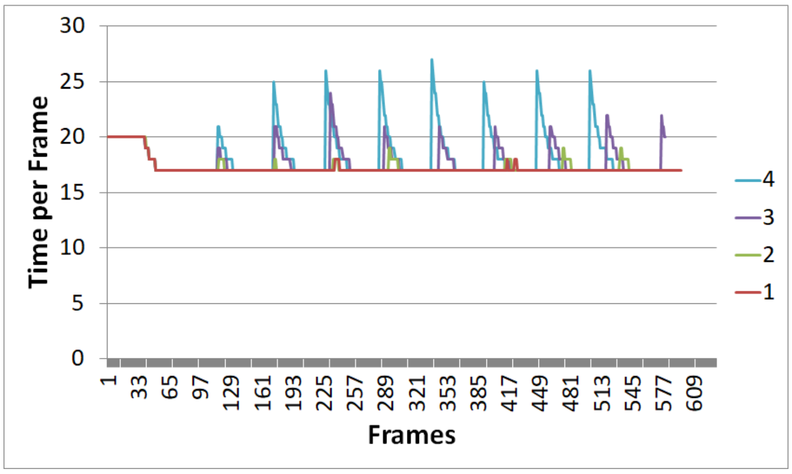


Figure 3.6: Graph of deformable object benchmark. Graph shows the impact on performance of mesh deformation. Vertex count of used mesh: 1 - 1144; 2 - 2304; 3 - 5824; 4 - 12474

Graph displays time per frame for each iteration3.6. Time is increased with each iteration. A visible tear in the simulation can be seen in the last iteration with 12474 vertices. This means that script is not optimal for bullet hole simulation as only meshes with lower vertex count can be used. Problems come from the fact that each collision whole mesh has to be changed. This could be fixed by changing only affected vertices but mesh class doesn't offer such solution. Another problem is that mesh collider has to be destroyed and added every bullet hit. A possible solution could be achieved by using tessellation shader.

All in all, this script can be used for low detail mesh deformables. I could see an iteration of it being used for car deformation, snow deformation, and similar uses.

3.6 Physics based animation benchmark

In this benchmark walking robots are animated with joints. Robots race from the start of the map to the end. Physics animation creates an uneven setting which leads to different robot winning every time.

The potential of joint components in Unity surprised me. It is possible to create complex animations and it is so easy to set up. The real challenge comes from making the character move properly with an ability to scale obstacles. Robots used in this benchmarks have problems with simple steps. It is no longer a question of one's programming ability,

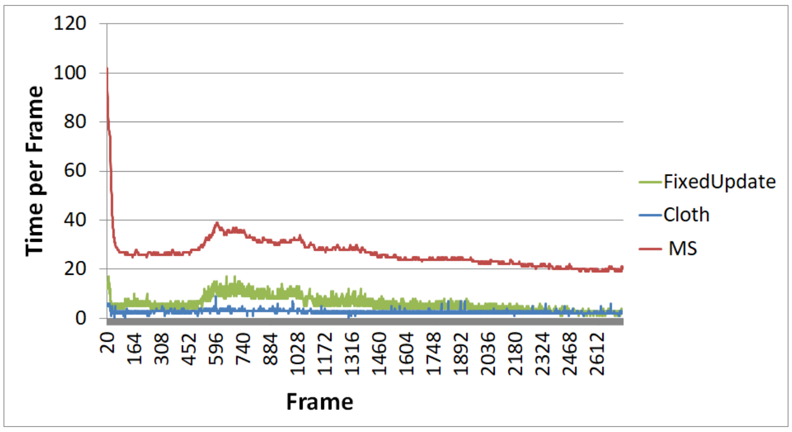


Figure 3.7: Graph of scripted game benchmark. The green line represents the CPU usage of *FixedUpdate*, the blue line CPU usage of *PhysicsSkinnedClothFinishUpdate* and the red line overall time per frame. Graph starts at 20th frame as overall time per frame is over 200 in first couple of frames.

but rather a question of his knowledge in the field of mechanics.

3.7 Game benchmark

The game benchmark is split into 2 parts: the scripted scenario and the actual game.

In the scripted scenario, a single wave of enemies attacks defenders at ramparts. CPU usage and time per frame are recorded.

Graph 3.7 shows CPU usage of *FixedUpdate*, *PhysicsSkinnedClothFinishUpdate* and overall time per frame. Units are randomly generated above ground to avoid spawning in the uneven terrain collider, *FixedUpdate* starts high because as units fall on to the ground. Next spike of *FixedUpdate* occurs when soldiers fire at each other, start turning into ragdolls and fall on the ground. Time per frame then lowers as fewer soldiers are active. *PhysicsSkinnedClothFinishUpdate* stays on a similar level because no flags are deleted during the scene. Overall time per frame follows the trajectory of previously mentioned functions.

The overall performance of the game benchmark is good. Framerate mostly stays below 30 ms per frame. Several optimization changes had to be implemented to keep it that way. Orc flags have to be deleted after a while as they are a constant burden on the CPU. The same goes for ragdolls. Although they are turned inactive after a while, any contact reawakens them which impacts performance. To conclude, the game works fairly well and shows the combined performance of all benchmarks.

Conclusions

In this bachelor's thesis, I have identified critical parts of physics simulation, and I have implemented seven physics simulation benchmarks in Unity game engine to test them. Some of the benchmarks test the accuracy of physics simulation and its impact on framerate, while others showcase the visual fidelity of Unity physics components, or whether it's possible to recreate a certain physics effect from the real world. Finally, I created a simple real-time strategy game a the seventh benchmark that combines previous benchmarks into one.

At first, I have touched upon contemporary physics engines in chapter 1. In the same chapter, I have also described how physics works in Unity and reviewed a couple of papers that compared physics engines. In chapter 2, I have described implemented benchmarks, what they test, why was a particular physics component used and how it was tested. I presented the results of these benchmarks in chapter 3. While all benchmarks brought the measured results, some of them also displayed unexpected interesting findings.

The first benchmark showed that mesh quality and timestep of physics simulation greatly affects CPU usage. It also highlighted the fact that the *AddForce* function in default force mode increases its power with different timesteps. This was better detailed in the fire accuracy test. The accuracy of collisions system test showcased that while intersection does happen during low-speed collisions, they are minimal and not detrimental to the simulation. Cloth component benchmark showed that the component serves its purpose well and with moderately detailed meshes isn't taxing on CPU. Custom deformable mesh benchmark results were a little disappointing. They proved that the proposed method can only be used with low detailed meshes, otherwise, it has a large impact on framerate. Results of the custom physics animation benchmark were rather surprising. I was astonished, how easy it is to create animations with the joint component. The game implemented as the last benchmark showed that it's possible to create a physics demanding game in Unity game engine.

For a future work, it is possible to expand the benchmarks, going deeper into each component. Another possibility is to implement similar benchmarks for a different game engine. The implementations in several engines could be then compared and the game engine with the best physics simulation could be found. This thesis also touched upon some topics that could be examined further. One thesis could be written about recreating humanoid characters with physics animations, while other could focus on different ways how to represent deformable objects.

Bibliography

[1] Gregory, J. *Game Engine Architecture*. Taylor and Francis Group, LLC, 2009, 9-10 pp.

[2] Šmíd, A. Comparison of Unity and Unreal Engine, Bachelor's Thesis, CTU, Prague, Czech Republic,. 2017.

[3] Millington, I. *Game Physics Engine Development*. Elsevier Inc., 2007.

[4] NVIDIA. PhysX home page. `https://www.geforce.com/hardware/technology/physx`, accessed: 2018-01-2.

[5] Havok. Havok home page. Available from: `https://www.havok.com/physics/`

[6] Zagrebelnyy, P. Rendering and simulation in offroad driving game. `https://www.gamasutra.com/blogs/PavelZagrebelnyy/20130613/194247/Rendering_and_simulation_in_offroad_driving_game.php`, 6 2013.

[7] Erwin Coumans, Y. B. Bullet quick start guide. `https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit#heading=h.2ye70wns7io3`, accessed: 2018-03-14.

[8] Goldstone, W. *Unity 3.x Game Development Essentails*. Pakt Publishing, 2011.

[9] Technologies, U. Unity Documentation. `https://docs.unity3d.com/Manual`, accessed: 2018-05-14.

[10] Pott, T. Realtime mesh deformation package. `http://labertorium.de/unity/789/unity-realtime-mesh-deformation-package/`, accessed: 2018-02-3.

[11] Flick, J. Mesh Deformation - Making a Stress Ball. `http://catlikecoding.com/unity/tutorials/mesh-deformation/`, accessed: 2018-02-3.

[12] Boeing, A.; Bräunl, T. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, ACM, 2007, pp. 281–288.

[13] Technologies, U. Unity WebGLBenchmark. `https://files.unity3d.com/jonas/WebGLBenchmark/`, accessed: 2018-05-14.

[14] Seugling, A.; M., R. *Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool.* Master's thesis, Umea University, Umea, Sweden, 03 2006.

[15] Erez, T.; Tassa, Y.; et al. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, IEEE, 2015, pp. 4397–4404.

[16] Lusth, A. Hand Painted Textures. `https://assetstore.unity.com/packages/2d/textures-materials/hand-painted-textures-31347`, accessed: 2018-05-1.

[17] M-Assets. Hand Painted Stone Floor Pack. `https://assetstore.unity.com/packages/2d/textures-materials/stone/hand-painted-stone-floor-pack-992471`, accessed: 2018-05-1.

[18] Zatylny, P. Hand Painted Forest Environment Free Sample. `https://assetstore.unity.com/packages/3d/environments/hand-painted-forest-environment-free-sample-35361`, accessed: 2018-05-8.

[19] Incorporated, A. S. RTS camera. `https://assetstore.unity.com/packages/tools/camera/rts-camera-43321`, accessed: 2018-05-1.

[20] Sylkin, D. Mixamo characters and animations. `https://www.mixamo.com/`, accessed: 2018-03-2.

Appendix **A**

Contents of enclosed CD

```
 readme.txt ....................... the file with CD contents description
 exe ................................... the directory with executables
 src ................................... the directory of source codes
   project .................................. implementation sources
   thesis ............ the directory of LaTeX source codes of the thesis
 text ...................................... the thesis text directory
   thesis.pdf ......................... the thesis text in PDF format
```