Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Games and Graphics

Game Development Demos - Game Physics

Bachelor's Thesis

Daniel Jiřík

Branch of study: Open Informatics
Supervisor: Doc. Ing. Jiří Bittner, Ph.D.

Prague, May 2022

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Jiřík** Jméno: **Daniel** Osobní číslo: **492393**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Dema pro výuku herního vývoje - herní fyzika

Název bakalářské práce anglicky:

Game Development Demos - Game Physics

Pokyny pro vypracování:

Zmapujte existující metody pro řešení fyzikální simulace v herních enginech. Vytipujte nejméně pět různých fyzikálních problémů v oblasti dynamiky tuhých těles (např. kolize dvou těles, nakloněná rovina, odstředivá síla, apod.), na kterých budete ilustrovat principy fyzikální simulace v herním enginu Unity.
Implementujte jednoduché výukové programy (dema), které budou přehledně ilustrovat vybrané fyzikální problémy a způsob jejich simulace v enginu. Soustřeďte se na vizualizaci důležitých veličin simulovaného systému, která umožní dobře pochopit interní princip fyzikální simulace. Vytvořené programy a jejich uživatelské rozhraní podrobte základnímu uživatelskému testu a zakomponujte výsledky testu do upravené verze programů.

Seznam doporučené literatury:

[1] Millington, Ian. Game physics engine development. CRC Press, 2007.
[2] Parberry, Ian. Introduction to Game Physics with Box2D. CRC Press, 2017.
[3] Jason Gregory. Game Engine Architecture (3rd edition). CRC Press, 2018.
[4] Michelene T. H. Chi and Ruth Wylie. The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes, Educational Psychologist, 49(4), 219–243, 2014.
[5] Papinčák, Marek. Zátěžové testy fyzikální simulace v herním enginu. Bakalářská práce, ČVUT FEL, 2018.
[6] Machovský, Štěpán. Sada výukových nástrojů pro kurz herního vývoje. Bakalářská práce, ČVUT FEL, 2020.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

doc. Ing. Jiří Bittner, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Declaration

I hereby declare I have written this Bachelor thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, May 2022

....
Daniel Jiřík

Abstract

Physics is a common problem in current games. However, if you would have asked an average player a simple question, how does the in-game physics work, he/she would probably have a hard time finding the answer. Therefore, this thesis was created to help answer this question. To achieve this, web pages were created which consist of 6 teaching demos. Demos have a uniform, clear, intuitive UI and were tested by eleven users.

Keywords: Teaching demos, Physics engine, Game engine, Physics, Unity.

Abstract

Fyzika je běžným problémem současných her. Pokud byste se však průměrného hráče zeptali na jednoduchou otázku, jak vlastně ta fyzika ve hře funguje, odpověď by nejspíše hledal těžko. S odpovědí na tuto otázku pomáhá tato bakalářská práce. Za tímto účelem byly vytvořeny webové stránky, které se skládají ze 6 výukových dem. Dema mají jednotné, jasné, intuitivní uživatelské rozhraní a byly testovány jedenácti uživateli.

Keywords: Výuková dema, Fyzikální engine, Herní engine, Fyzika, Unity.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor Jiří Bittner, associate professor in the Department of Computer Graphics and Interaction, for the continuous support and guidance. I would also like to thank my family and friends who helped me to accomplish everything in time and gave me useful tips on how to improve my work further.

List of Figures

1.1 G-Switch 3 [1] - Change of the direction of gravitational force. 1

2.1 Euler method illustration, s_i = position in current frame, s_{i+1} = approximated position in next frame, s_e = exact solution, the red line is the slope at the point s_i of the green function, the length of the yellow dashed line is the error. 6
2.2 Moving boxes from left to right. The top box is run on 2 FPS and bottom box is run on 3 FPS. 11
2.3 Types of colliders. 12
2.4 Mesh colliders. 13
2.5 Fixed Joint. Purple line indicates connection between the object and its anchor. 15
2.6 Hinge Joint. The object can rotate around the pre-defined axis. 15
2.7 Spring Joint. The purple line behaves like a rubber band with pre-defined parameters. 16
2.8 Character Joint. It is possible to limit rotation around each axis. Axis are represented as orange arrows. 16
2.9 x_0, y_0 and x_1, y_1 are known points, x is a point we want to estimate a value for, and y is the estimated value. 18
2.10 Temperature approximation - graphical visualization. 19

3.1 A panel with important parameters in Destructible Objects demo. 22
3.2 UI. 22
3.3 Lever demo. 23
3.4 The green cube's force is larger and will overweight the red cube. 24
3.5 Seesaw illustration. 25
3.6 Conservation of Angular Momentum Demo. 25
3.7 Moving hands changes moment of inertia. 27
3.8 Tunneling Demo. 27
3.9 Undetected collision of a fast moving object. 29
3.10 The box is approaching the wall from the left. Despite a collision being registered, the box still moves through the wall. 29
3.11 Collisions of Two Objects demo. 30
3.12 Collision. 31
3.13 Accuracy of collision. 31
3.14 Difference between the continuous speculative and the discrete collision detection mode. 32
3.15 Solar System - Centripetal Force demo. 32
3.16 Destructible Objects demo. 35
3.17 Broken fixed joints are not visualized. 36
3.18 The visualization of a cup. 37

List of Tables

3.1 Lever modifiable parameters. 24
3.2 Conservation of Angular Momentum modifiable parameters. 26
3.3 Tunneling modifiable parameters. 28
3.4 Collisions of two objects modifiable parameters. 31
3.5 Solar System modifiable parameters. 35
3.6 Destructible objects modifiable parameters. 37

4.1 General Questions. 39
4.2 Question No. 1 - Did you learn anything new from this demo? 40
4.3 Question No. 2 - Have you noticed any change in behaviour of selected objects after adjusting given parameters? 41
4.4 Question No. 3 - Is there anything interesting you came across or learned while using this demo? 42
4.5 Question No. 4 - Is there anything you would improve on this demo? 43
4.6 Question No. 5 - What was your overall experience with this demo? 44
4.7 Summarizing question No. 1 - What was your overall experience? 45
4.8 Summarizing question No. 2 - Did you find information button in each demo helpful? 45
4.9 Summarizing question No. 3 - Did you have any trouble using the user interface? 45
4.10 Summarizing question No. 4 - Have this set of demos gave you an overall view on how physics in game engines work? 45
4.11 Summarizing question No. 5 - Is there anything you would like to say about the project? 46

Contents

Abstract iv

Abstract v

Acknowledgements vi

List of Figures vii

List of Tables viii

1 Introduction 1

2 Game Physics 3
2.1 Physics Engines . . . 3
2.1.1 Popular Physics Engines . . . 3
2.2 Physics Simulation . . . 4
2.2.1 Physics Step . . . 4
2.2.2 Types of numerical integration . . . 5
2.3 Unity Physics . . . 9
2.3.1 Rigidbody . . . 9
2.3.2 Force Modes . . . 10
2.3.3 Time in Unity . . . 10
2.3.4 Unity colliders . . . 12
2.3.5 Joints . . . 14
2.3.6 Collision Detection Modes . . . 16
2.3.7 Interpolation . . . 17

3 Physics Demos in Unity 21
3.1 User interface - UI . . . 21
3.2 Lever . . . 23
3.3 Conservation of Angular Movement . . . 25
3.4 Tunneling . . . 27
3.5 Collision of Two Objects . . . 30
3.6 Solar System - Centripetal Force . . . 32
3.7 Destructible Objects . . . 35

4 Results and Discussion 38
4.1 Testing the demos during the development . . . 38
4.2 User Test . . . 39
4.2.1 General information part . . . 39

4.2.2 Separate demos part 39
4.2.3 Final thoughts part 44
4.2.4 Future changes 46

5 Conclusion 48

Bibliography 50

A User manual 51

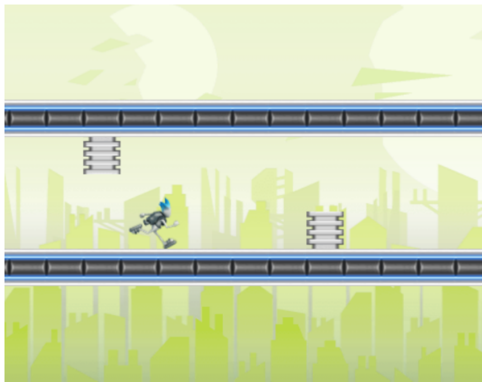
B DVD contents 52

Chapter 1

Introduction

Nowadays, there are continuously increasing demands on video games. Video games have to be visually appealing, they have to have a unique storyline, good gameplay, interesting main idea, realistic physics, and much more. Most of these qualities must be met for video games to succeed.

I would like to focus on physics in video games. Video games can be made in 2D or 3D and all of them might have different demands for physics. Some games try to have as realistic physics as possible, others might want to alter the physics to achieve unique gameplay features. For instance, the 2D game G-Switch 3 is a game where your character runs forward on a track and dodges obstacles by changing the direction of gravitational force (see Figure 1.1).



(a) G-Switch 3 - gravitational force pointing downwards.

(b) G-Switch 3 - gravitational force pointing upwards.

Figure 1.1: G-Switch 3 [1] - Change of the direction of gravitational force.

Among things we can achieve with physics also belong different types of simulations. For example, simulating the destruction of a building, visualizing water impact on the terrain, examining an outcome of an avalanche, or creating ragdoll effects. There are many other things we may achieve with physics. On the other hand, it is not always better to use physics to simulate everything in our game. When physics is not implemented correctly, it can leave

a bad impression on a player. For example, it is usually better to use animation to move your in-game 3D character rather than simulating the movement, because physically simulated behavior is chaotic and unpredictable [2].

Realistic physics in large games might be computation-heavy. Because of that, game developers try to achieve the middle point between physics that is as close to reality as possible and physics that is fast to compute.

There are many physics engines, which tackle physics in their way. For example, some of the most well-known are PhysX, Havok, Box2D, and Bullet. These physics engines are usually integrated into game engines. The majority of game engines use PhysX like Unreal Engine, CryEngine, and Unity. Unity uses the PhysX engine for 3D physics and the Box2D engine for 2D physics. In this project, I decided to use the Unity engine to illustrate how game physics in this engine is implemented.

Chapter 2

Game Physics

2.1 Physics Engines

A physics engine [3] is software that can simulate real-life physics in computer programs. However, these simulations are not one hundred percent accurate and serve as a precise approximation.

A physics engine can simulate all sorts of physics. For example, rigid body dynamics, fluid dynamics, or soft body dynamics.

It is possible to divide physics engines into two groups. The first group is called real-time and the second is high-precision. Real-time physics engines are less accurate than high-precision physics engines, but they are not as computational heavy. Real-time physics engines are used in software, typically in computer games, where the computation speed is more important than precision. High-precision engines are typically used in science software or animated movies.

2.1.1 Popular Physics Engines

Box2D

Box2D [4] is a free open-sourced real-time physics engine written in C++. This engine is used to simulate rigid body simulation in 2D software. For example, the well-known android game Angry Birds was created with this engine. Unity uses Box2D for its 2D physics.

Bullet

Bullet [2] is a free open-sourced real-time physics engine that simulates rigid and soft body dynamics and collisions. Bullets' soft body dynamics support simulations with cloth, deformable objects, and ropes. For example, Google uses this engine for game development or virtual reality.

PhysX

PhysX [2] is a free, real-time multithreaded physics engine. The versions PhysX 4.1 and lower are also open-source and their source codes can be accessed on GitHub. PhysX is developed by Nvidia and is one of the most used physics engines today. Unity uses PhysX for 3D physics. PhysX is supported on a wide range of platforms, for example on Microsoft Windows, macOS, Linux, Playstation 4, Xbox One, iOS, or Android. This engine supports both rigid and soft body dynamics. It also provides volumetric fluid simulation. PhysX can use GPGPU (General-purpose computing on graphics processing unit). GPGPU is the use of a graphics processing unit to compute tasks that would have normally been computed on a central processing unit in order to get better performance.

VisSim

VisSim is a high-precision physics engine that simulates dynamics. It is used for creating virtual dynamic systems.

2.2 Physics Simulation

Physics simulation [5] is used to simulate real-world physics problems. Physics simulation in physics engines is divided into multiple steps. Physics engines compute final forces such as velocity, acceleration, or torque.

2.2.1 Physics Step

In game engines, physics is usually calculated in a fixed amount of time per second. For example, in Unity fixed delta time step is set to 0.02 by default. This means that the physics step is calculated every 0.02 seconds, thus 50 times per second. Calculating physics for a fixed amount of times per second results in deterministic outcomes, which stay the same regardless of system performance.

The physics step may be divided into multiple parts. It begins with updating and applying forces. To calculate the outcome of these forces, numerical integration is used. Numerical integration is a method that enables computers to compute integrals. There are various types of numerical integration, some of them are described here 2.2.2.

After numerical integration, collisions are solved. Collision detection is divided into two parts – broad and narrow phases. The broad phase creates a BVH-boundary volume hierarchy. This tree-like hierarchy of nodes contains objects that are most likely to collide. The broad phase is fast and removes all non-possible collisions, although it may produce false-positive

collisions that have to be tackled in the narrow phase. The narrow phase solves all possible collisions, it is accurate but computationally expensive.

If there were not any collisions, the physics step may continue updating physics objects. Otherwise, penalty forces must be applied and collisions calculated again.

2.2.2 Types of numerical integration

Numerical integration is a way that enables computers to compute definite integrals.

Numerical integration is also a great way to approximate the results of ordinary differential equations. An ordinary differential equation is an equation including a function containing one independent variable and the derivatives of this function. [2]

In physics engine we mostly want to calculate Equations of Motion [6]. These equations are Newton's second law

$$\vec{F} = m\vec{a}$$

and Rotational force

$$\vec{M} = I\vec{\alpha}$$

.

Differential equations of motion can sometimes be solved analytically. However, finding an analytical solution to most physics differential equations is a hard task, thus closed-form solution remains unknown. Furthermore, we can't always predict positions and velocities with closed-form solutions, because physics forces in our game may change over time. An example of when a closed-form solution is suitable is when we want to predict a trajectory of an arrow, that has been shot from a bow.

There are many methods of numerical integrations e.g., Verlet, Explicit Euler, Runge-Kutta methods. Arguably the most widely used is the Verlet method. Euler is the easiest to implement. On the other hand, Runge-Kutta is complex but more precise in most cases.

Explicit Euler method

Explicit Euler is one of the simplest numerical integration methods [4]. In this method we assume that velocity is constant during one frame and because of that, we are able to calculate the position of an object in the next frame. It works like this:

$$s_{i+1} = s_i + v(t_i)\Delta t$$

$v(t_i)$	velocity in current frame
s_i	position in current frame
s_{i+1}	position in next frame
Δt	duration of the frame

However, as we can see in Figure 2.1, the distance between the exact and approximated solution is relatively large. For the larger time between each time step, this method can get inaccurate.

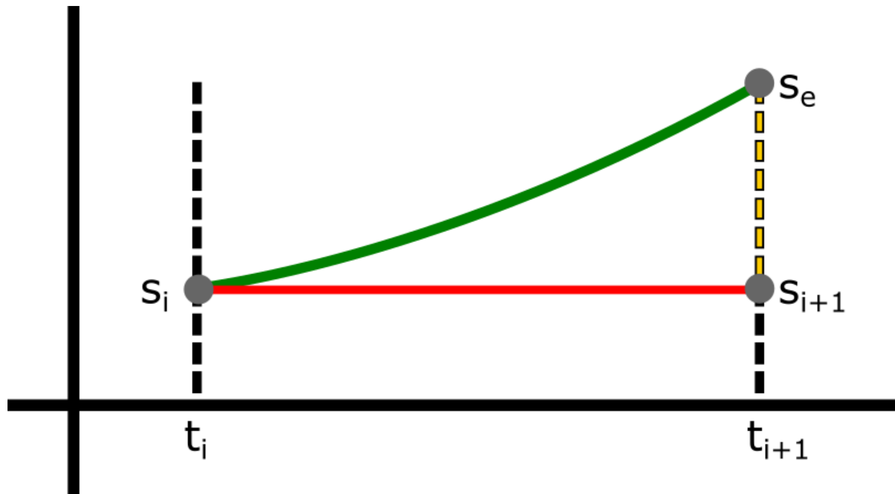


Figure 2.1: Euler method illustration, s_i = position in current frame, s_{i+1} = approximated position in next frame, s_e = exact solution, the red line is the slope at the point s_i of the green function, the length of the yellow dashed line is the error.

Verlet method

This method's advantage is that it is easy to implement constraints like angles and lengths. That means, the Verlet method is suitable for soft body dynamics like cloth or ragdoll physics. [4]

Verlet method works like this. We know that position in next time step is equal to:

$$\Delta s_{i+1} = v_i \Delta t + \frac{a_i \Delta t^2}{2}$$

We may substitute $v_i \Delta t$ for Δs_i as it is good enough approximation:

$$\Delta s_{i+1} = \Delta s_i + \frac{a_i \Delta t^2}{2}$$

and then because $\Delta s_{i+1} = s_{i+1} - s_i$ and $\Delta s_i = s_i - s_{i-1}$ we may substitute further, leaving us with

$$s_{i+1} = 2s_i - s_{i-1} + \frac{a_i \Delta t^2}{2}$$

a_i	acceleration
s_i	position in current frame
s_{i+1}	position in next frame
s_{i-1}	position in previous frame
Δs_i	distance between s_i and s_{i-1}
Δt	duration of the frame

The final equation indicates that it is sufficient to know the only current and previous positions, acceleration, and duration of the frame to calculate the next position.

Runge-Kutta method

Family of Runge-Kutta methods [7] use the Euler method approach multiple times on different points and calculating the average value. To clarify, the Euler method is the first order Runge-Kutta method.

I would like to introduce you the Runge-Kutta method of the fourth order (RK4). This method's initial parameters are any function $\frac{dy}{dt} = f(y,t)$ and initial value of $y_0 = y(t_0)$. Both y_0 and t_0 are known beforehand. The purpose is to approximate the value of the parameter y based on the change of the parameter t. In the case of the equation of motion, we calculate next position based on time where:

t_0	initial time,
y_0	is value of the function at initial time.

In order to calculate the value of y in next step, we need to set the step size h. The smaller the step size the more accurate the approximation of the function:

$$y_{i+1} = y_i + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{i+1} = t_i + h$$

t_{i+1}	time in next step,
y_{i+1}	value of y in next step,
h	step size.

However, in the first equations appear unknown parameters $k_i, i \in 1,2,3,4$ these are slopes of the function calculated in different points:

k_1	slope of the function at y_0 and t_0,
k_2	slope of the function at mid point of $y_0 + k_1$,
k_3	slope of the function at mid point of $y_0 + k_2$,
k_4	slope of the function at end point of $y_0 + k_3$.

Formulas to calculate $\forall k_i, i \in 0, 1, 2, 3, ...$

$$k_1 = f(y_i, t_i)$$
$$k_2 = f(y_i + k_1\frac{h}{2}, t_i + \frac{h}{2})$$
$$k_3 = f(y_i + k_2\frac{h}{2}, t_i + \frac{h}{2})$$
$$k_4 = f(y_i + k_3 h, t_i + h)$$

Let me demonstrate it on simple equation $f(y, t) = y^2 t$. First, we set our initial value $y_0 = 2$, time $t_0 = 0$ and step size $h = 0.1$

Now we need to calculate $\forall k_i, i \in 1, 2, 3, 4$

$$k_1 = f(2, 0) = 0$$
$$k_2 = f(2 + 0\frac{0.1}{2}, 0 + h/2) = 0.2$$
$$k_3 = f(2 + \frac{1}{5}\frac{0.1}{2}, 0 + h/2) = 0.202005$$
$$k_4 = f(2 + k_3\frac{1}{10}, 0 + h) \doteq 0.408121006$$

Then we can calculate desired y_1 like:

$$y_1 = 2 + \frac{1}{6}\frac{1}{10}(k1 + 2k_2 + 2k_3 + k4) \doteq 2.020202183$$

However, this is not an exact solution. If we want to find the correct answer, we must solve this differential equation:

$$\frac{dy}{dt} = y^2 t$$

To begin, we divide both sides with y^2 and multiply with: dt

$$\frac{1}{y^2}dy = tdt$$

Then we integrate left side with respect to y and the right side with respect to t:

$$\int \frac{1}{y^2}dy = \int tdt \Rightarrow \ -\frac{1}{y} = \frac{t^2}{2} + C$$

We may simplify the result further:

$$y = -\frac{2}{t^2 + C}$$

Substitude our initial values of y_0 and t_0 to calculate C:

$$2 = -\frac{2}{0^2 + C} \Rightarrow C = -1$$

Finally, we can compute the exact solution for y_1:

$$y_1 = \frac{2}{0.1^2 - 1} = 2.02020202$$

In this case, Runge-Kutta method's error $\doteq 0.00000008\%$. As we can see in this instance, the error for one step is close to 0%.

2.3 Unity Physics

Unity [8] provides tools to handle physics. It has two built-in physics engines for that. The first one is the Box2D physics engine 2.1.1 that handles 2D physics and the second one is the PhysX physics engine 2.1.1 that deals with 3D physics. Additionally, it is possible to install two physics engines for the Data-Oriented Technology Stack projects, the Unity physics engine, and the Havok Physics engine. Data-Oriented Technology Stack consists of three main parts - entity component system, C# job system, and burst compiler. The entity component system declares how to organize your code, the C# job system is then able to run the application in multiple threads by giving each thread "jobs" to do, and finally, the burst compiler converts those jobs into highly optimized machine code. All three components combined result in a much better performance of your Unity project.

2.3.1 Rigidbody

The rigidbody is a solid body, whose particles retain the same distance between each other during any motions of the body. Simply said, a rigidbody can not be deformed.

In Unity, the main component that is needed to bring life (physics) to objects is the rigidbody component. After attaching rigidbody components to an object. The object is controlled by forces. The default force that affects the rigidbody is gravity. Gravity has a default force that can be modified in Unity settings.

Using an object with a rigidbody component as if it was not affected by forces, for example, updating the object's position by a fixed distance every physics step can be achieved by enabling the rigidbody property Is Kinematic. By toggling this property on, the physics engine loses control over the object's behavior.

Having a lot of rigidbody components in your scene can be computationally heavy. This is because, for every object, the physics step has to be done a fixed amount of times per second.

The default is 50 times per second. This problem is partially solved by introducing sleep mode to rigidbodies. After rigidbodies' linear or rotational speed fall under a certain threshold, rigidbodies go to sleep mode. In this mode, objects do not move until they are awakened. To awake sleeping rigidbody component, the object has to be set in motion again. Sleep threshold may be manually changed in Unity settings or script individually for each rigidbody component.

2.3.2 Force Modes

To move a rigidbody object, we add force to the object with the function AddForce(). This function applies the size of the force in a direction of a chosen vector.

In Unity exists four modes [9] in which we may add force to a rigidbody. The purpose of these modes is to change the velocity of a rigidbody. Every mode has its characteristics and is used in different instances.

Force

This mode adds a continuous force physics step to a rigidbody along one second considering its mass.

Acceleration

Acceleration is similar to force with only one difference, it does not take the rigidbody mass into account.

Impulse

Impulse applies the force instantly in one physics step. The force is also changed based on the rigidbody mass.

VelocityChange

As the name suggests, this mode changes velocity instantly in one physics step. Rigidbody mass does not alter the size of the force.

2.3.3 Time in Unity

Time is an important part of any game engine. However, dealing with time is not an easy task. How to measure time? Every motherboard has built-in system clocks which applications can access through API. Thanks to these clocks, most game engines measure time between frames to get a sense of time.

Delta time

In Unity and many other game engines, we use deltaTime. Calling deltaTime in our script returns a time that has passed between the previous and the current frame. To be more precise, deltaTime is the number of seconds it took to process the previous frame. To calculate the deltaTime's value, Unity uses the system's internal clock to get time at the beginning of the previous frame and time at the start of the current frame. Subtracting these two values yields deltaTime.

Nonetheless, applications created in game engines are made to be run on devices with different hardware. Some hardware is faster than the other and performance can differ a lot on each machine. This causes programs that run on better hardware to have smaller deltaTime steps than programs that are run on old hardware. Is that a problem? For example, when we move an in-game character by a fixed distance each second:

$$character.transform.position += Vector3.forward * Time.deltaTime;$$

Because of the multiplication by the deltaTime, this line of code in Unity moves a character in direction of the Z-axis by one unit per second despite the hardware. The only difference that might be noticeable on the screen on different hardware is the distance traveled between each frame. On slower hardware with low frame rate distances are larger and can be seen with the naked eye. To summarize, deltaTime ensures constant speed of our application regardless of frames rate 2.2.

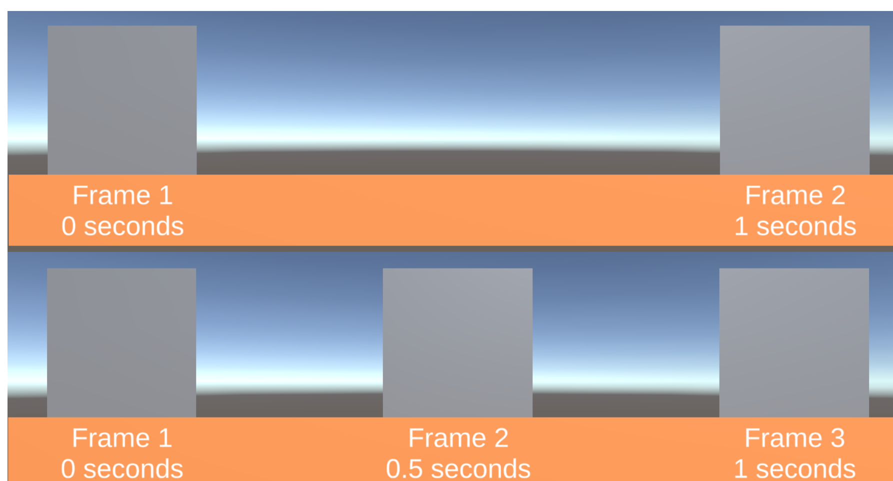


Figure 2.2: Moving boxes from left to right. The top box is run on 2 FPS and bottom box is run on 3 FPS.

Fixed update

However, what if at the position at the time of 0.5 seconds were an obstacle 2.2? This might be a problem, a slower hardware might not register collisions the same as faster hardware. This

behavior produces non-deterministic results and may provide advantages or disadvantages in various games.

Thankfully, there is a solution to this in Unity called fixed update. The fixed update ensures that a code inside the fixed update function will be executed a fixed amount of times per second. This guarantees us, that the code inside the fixed update function will act the same on different devices. Inside this fixed update we should mainly put a code that deals with physics.

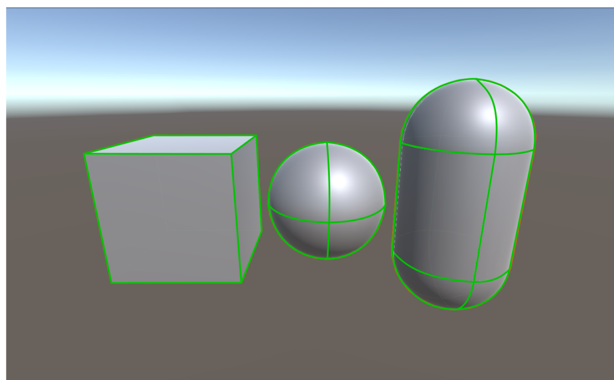
How does it work [10]? Suppose we have two time variables, a sum of passed delta times - T and a sum of passed fixed delta times - FT. Both of these variables are initialized to zero. At the beginning of the frame, Unity adds delta time to the T. Then Unity checks whether $FT < T$. If yes, Unity proceeds to do a fixed update and add fixed delta time(0.02 is the default in Unity) to FT. After that, check again if $FT < T$, if that is the case, do the fixed update, add fixed delta time to FT again and repeat this process until $FT > T$.

2.3.4 Unity colliders

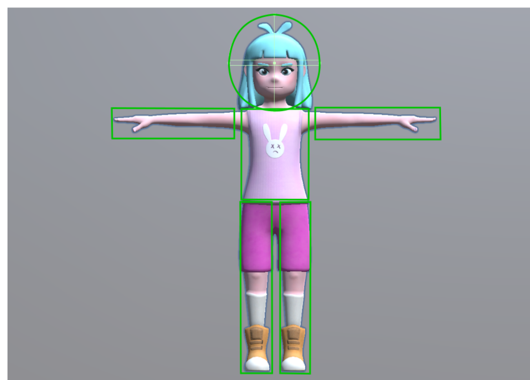
In Unity, objects might have a collider component attached [11]. This component wraps around the object and registers collisions. Objects with any colliders and rigidbody attached react to collisions. For example, crashing a car with both of these components into another object with any collider moves the car based on the laws of physics and properties of the colliders.

Compound collider

Compound collider 2.3b is made from multiple primitive colliders to represent the shape of an object as close as possible while maintaining low computational complexity. There are three primitive colliders in Unity - box, sphere, and capsule collider 2.3a.



(a) From Left to Right - Box, Sphere, Capsule Collider.



(b) Compound Collider made of five box colliders and one capsule collider [12].

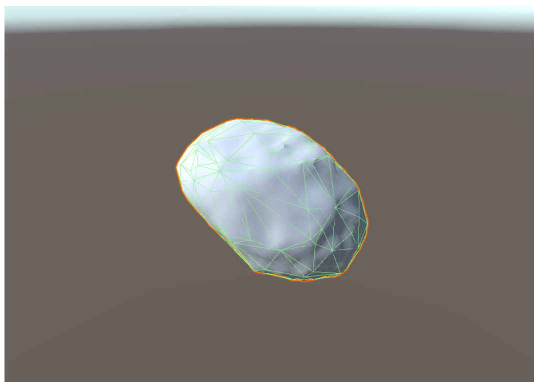
Figure 2.3: Types of colliders.

Mesh collider

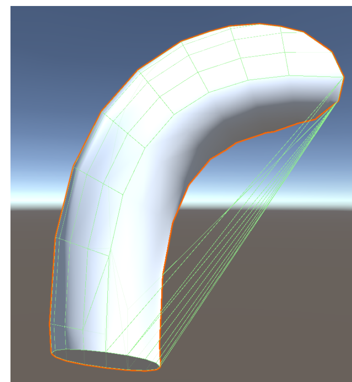
When primitive or compound colliders are not enough to represent the complex shape of an object, we may use a mesh collider. The shape of the collider is the same as the mesh of the object.

There are two main reasons why mesh colliders shouldn't be used very often. The first reason is that mesh colliders don't collide with each other. If we were to put a mesh collider on every object in our project, we wouldn't have any collisions registered. The second reason is that calculating collisions for this collider is a computationally heavy task and using a lot of them may cause significant performance drops.

However, to solve the first problem, we may set our mesh collider as convex. By doing so, our object can collide with other objects that have mesh collider attached. On the other hand, convex mesh colliders aren't as accurate and when representing difficult shapes, the convexity of the collider causes unwanted results, for example 2.4b.



(a) Convex mesh collider on a rock.



(b) Inaccurate convex mesh collider.

Figure 2.4: Mesh colliders.

Static collider

A static collider serves as a collider that can be used on static objects. For instance, when we have a firm wall in our project and we attach a collider to it, the collider is referenced as a static collider. To summarize, a static collider is a collider on an object without a rigidbody component.

Rigidbody colliders

If the object has a rigidbody component, the collider is called a dynamic collider. We use this type of collider when we want to simulate real physics on our object.

Kinematic rigidbody colliders

A collider attached to a rigidbody with IsKinematic property enabled is called a kinematic rigidbody collider. For example, we may use it for an elevator in our game that is moved after pressing a button.

Interaction of colliders

Not every overlap of colliders triggers collision events. To clarify, here is a table of registered collisions between two types of colliders.

Registered collisions table			
	Rb collider	Kinematic rb coll.	Static collider
Rb collider	✓	✓	✓
Kinematic rb coll.	✓	✗	✗
Static collider	✓	✗	✗

Triggers

In some cases, we might want to detect when a collision would occur, rather than resolving collisions normally. For this purpose, triggers are used. In Unity, we may check three states - OnTriggerEnter, OnTriggerStay, and OnTriggerExit. OnTriggerEnter occurs on the physics step when a collider, which has trigger property on, touches another collider. OnTriggerStay executes every physics frame as long as these two colliders are overlapping and finally, OnTriggerLeave is called once when these two colliders stop overlapping.

2.3.5 Joints

Joints in Unity serve as a connection component between two rigidbodies. Joints may also connect a rigidbody with a fixed point in space (anchor) [13]. In Unity exist five types of joints - character joint, configurable joint, fixed joint, hinge joint, and spring joint. All of these joints can be broken when an excessive force or torque is applied to the connected objects. These properties can be modified in joint properties.

Fixed joint

When connecting a rigidbody to another rigidbody or fixed point by a fixed joint. We restrict the movement of the rigidbody to mimic the movement of the connected rigidbody or fixed point. Fixed joints can be used to simulate destructible objects, or to keep a fixed distance between connected objects.

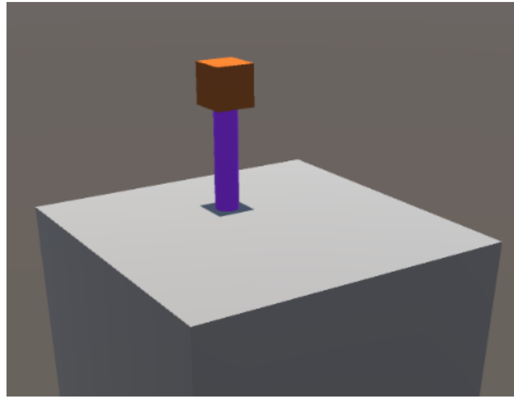


Figure 2.5: Fixed Joint. Purple line indicates connection between the object and its anchor.

Hinge joint

The hinge joint allows only one rotation around a chosen axis. This type of joint can be useful to simulate doors, seesaws, and more.

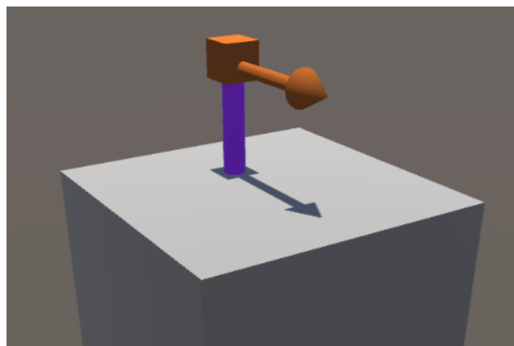


Figure 2.6: Hinge Joint. The object can rotate around the pre-defined axis.

Spring joint

A spring joint is similar to a fixed joint except for the fact that the distance between connected objects is not fixed and may stretch slightly when forces are applied. We can imagine it as if the objects were connected by a rubber band.

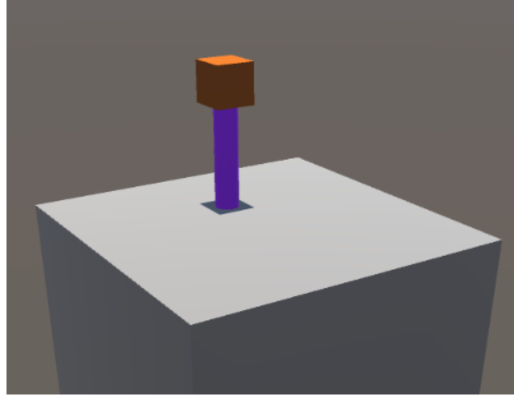


Figure 2.7: Spring Joint. The purple line behaves like a rubber band with pre-defined parameters.

Character joint

As the name suggests, this joint is used to simulate joints in the character's body, like hips, shoulders, knees.

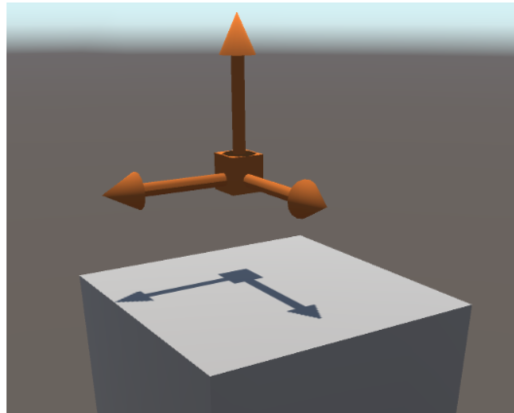


Figure 2.8: Character Joint. It is possible to limit rotation around each axis. Axis are represented as orange arrows.

Configurable joint

This is the most configurable joint in Unity. Using this type of joint we can emulate every other joint. This joint is used when we have very specific demands on the joint.

2.3.6 Collision Detection Modes

In Unity exist 4 collision detection modes [14]. Each of them has its pros and cons. We should always use the most suitable mode for our objects. For example, when we have large, slowly moving objects the majority of the time, a discrete collision system is sufficient.

Discrete

Discrete is the fastest collision detection mode in unity. Using this mode may not detect collisions of fast-moving objects. For example, a bullet can go through the wall without registering any collisions.

Continuous

Continuous detection mode prevents objects with this detection mode to pass through other objects with static colliders. Although it is still possible for these objects to pass through other objects with continuous colliders.

Continuous dynamic

Continuous dynamic mode solves this problem and prevents two objects with continuous dynamic colliders to pass through each other. Nevertheless, it is still possible for continuous dynamic colliders to pass through discrete colliders.

Continuous speculative

Continuous speculative mode collides with both static and dynamic objects. It is best to detect collisions of spinning objects. However, collisions might be a little inaccurate.

2.3.7 Interpolation

Interpolation serves as a tool to find the approximate value in an interval for a parameter. In other words, we use our knowledge of known discrete data points and estimate new data points from it [15]. For example, in graphics, we use interpolation to create interpolation splines. Interpolation splines are curves that pass through all control points.

Linear interpolation

Linear interpolation is a method of curve fitting with a first-degree polynomial. A first-degree polynomial can be imagined as a straight line in the graph. The formula of a first-degree polynomial is

$$p(x) = ax + b$$

where $a, b \in \mathbb{R} \wedge a \neq 0$ How do we interpolate between two known points? Let the first point be x_0, y_0 and the second one x_1, y_1. Thanks to these two points, we are able to estimate any $y(x)$ value where $x \in (x_0, x_1)$. The formula to get unknown value y is as follows:

$$\frac{y_1 - y_0}{x_1 - x_0} = \frac{y - y_o}{x - x_0}$$

This formula is based on the fact, that a straight line between two points has the same slope value on its whole interval. Then we are able to compute y value

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

We can visualize it as finding a point on a straight line between these two points 2.9

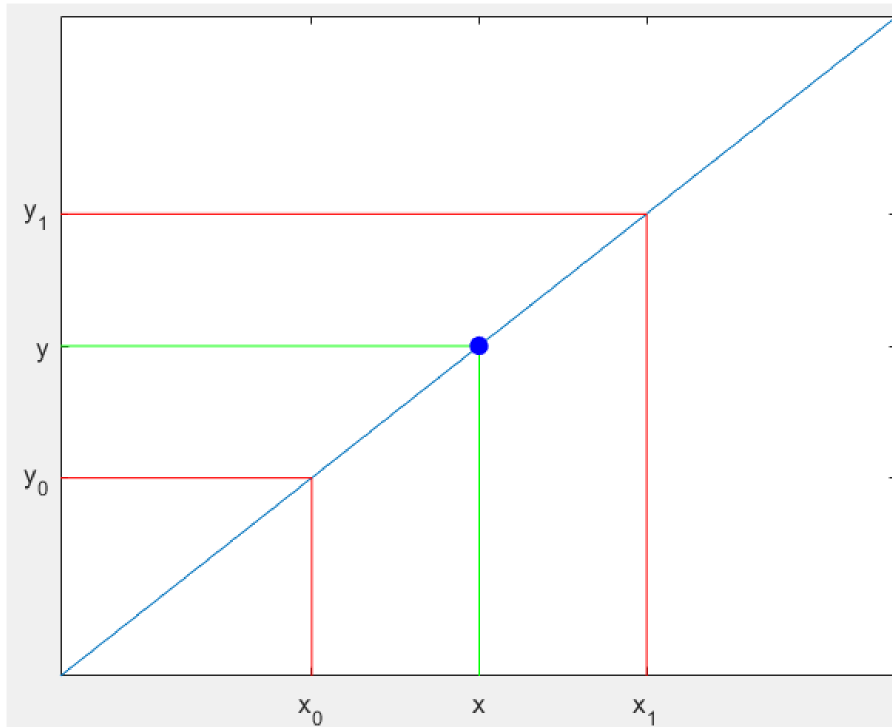


Figure 2.9: x_0, y_0 and x_1, y_1 are known points, x is a point we want to estimate a value for, and y is the estimated value.

Suppose we have data on the average temperature on Monday and Wednesday and we want to know the estimated temperature on Tuesday. The average temperature (y value) on Monday was 10°C and on Wednesday 20°C. Tuesday is in-between these days, thus we can assign numbers (x value) to each day as follows Monday = 0, Tuesday = 1, Wednesday = 2 and use the previous formula:

$$y = 10 + (1 - 0)\frac{20 - 10}{2 - 0} = 15$$

According to linear interpolation, the average temperature on Tuesday was 15°C. Graphic visualization of the previous example. 2.10

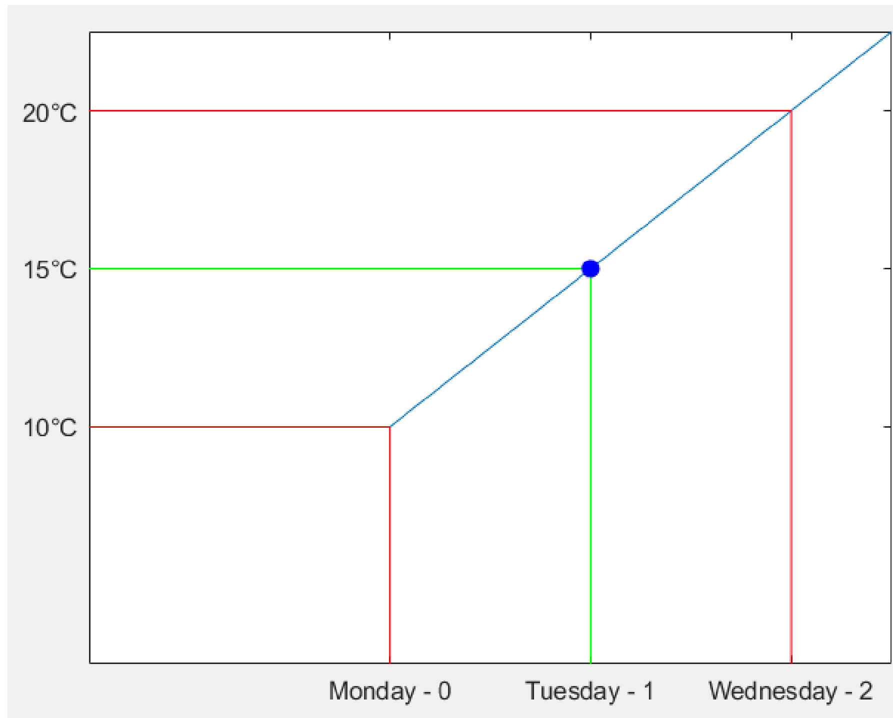


Figure 2.10: Temperature approximation - graphical visualization.

Rigidbody interpolation

The rigidbody component has interpolation property. Interpolation is used when we need to smooth out the movement of an object. For example, when we slow down the time in Unity. We might not have enough physics steps and output can look jittery. When we run computer games at high frames per second, physics that is run a small number of times per second might be out of sync with rendered graphics and may cause the same jittery effect.

For this purpose, rigidbody components can be interpolated. Interpolation solves this rendered graphics and physics out-of-sync problem. In Unity, we can interpolate or extrapolate. The difference between these is that interpolating is always a little bit delayed but can be smoother than extrapolation. Extrapolation predicts the position of the rigidbody based on current velocity. However, it may produce artifacts. For instance, the fast-moving object can seem to go through a wall for one frame and then move back to the previous position in the next frame.

Unity Lerp

Lerp is an abbreviation for linear interpolation [16]. Lerp in Unity is used to smooth out animations, move objects between two points on a line, and so on. This function takes 3 parameters - minimal value, maximal value, and interpolation point. The interpolation point stands between 0 and 1. Thanks to this point, it is possible to calculate the value between minimal and maximal value. We usually use the lerp function to interpolate vectors, scales,

colors, rotations, and more. Using this method, we may create animations like a button that is constantly getting bigger and smaller in a fixed period.

Chapter 3

Physics Demos in Unity

I decided to use Unity because it is easy to use, well-documented game engine. I divided my project into scenes where every scene represents one physics problem. Interactivity is provided through the user interface. Users can change multiple parameters to influence the outcome of the simulation.

I created three physics teaching demos to demonstrate how some well-known physics problems in the real-world work in the Unity engine and three physics teaching demos that showcase physics problems that are closely related to the game engine.

All demos can be tried online on a website [17] which I created as a part of my project.

3.1 User interface - UI

An important part of making teaching demos is to make a clear and intuitive user interface. The user interface stands between the user and demos and provides communication between them.

The template of the user interface is the same throughout all demos. Small layout changes were made to maintain the desired functionality of all demos.

Every demo has a panel in the top left corner with the most important parameters that the user should observe. For example, in the Destructible Objects demo the user should be mainly paying attention to parameters as shown in 3.1. The only panel that is always the same is the panel in the top right corner. This panel manages the state of simulation with start/pause/restart buttons, hides UI, and also provides information about each demo 3.2a. The last type of UI panel is a panel with parameters that users can modify. The number of parameters vary and depend on the selected demo. For example, in Conservation of Angular Movement demo 3.2b, the user can tweak simulation values with the help of a Unity UI component - slider.

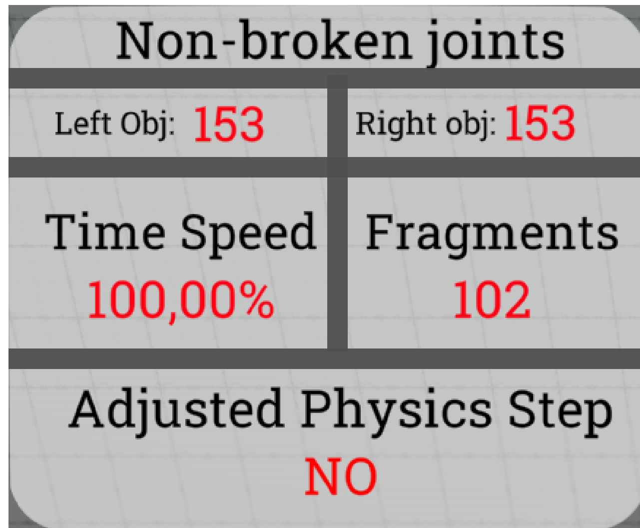
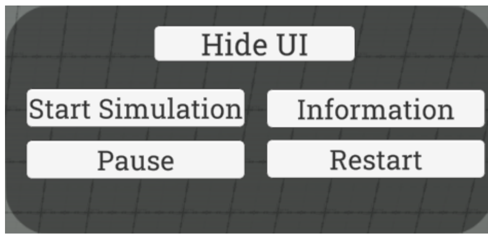
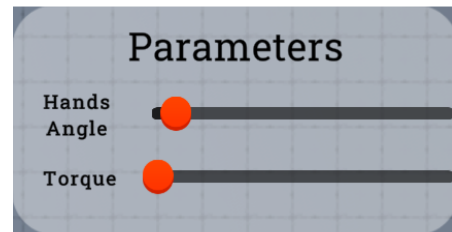


Figure 3.1: A panel with important parameters in Destructible Objects demo.



(a) A panel which manages the state of the simulation.



(b) A panel with parameters user can change.

Figure 3.2: UI.

3.2 Lever

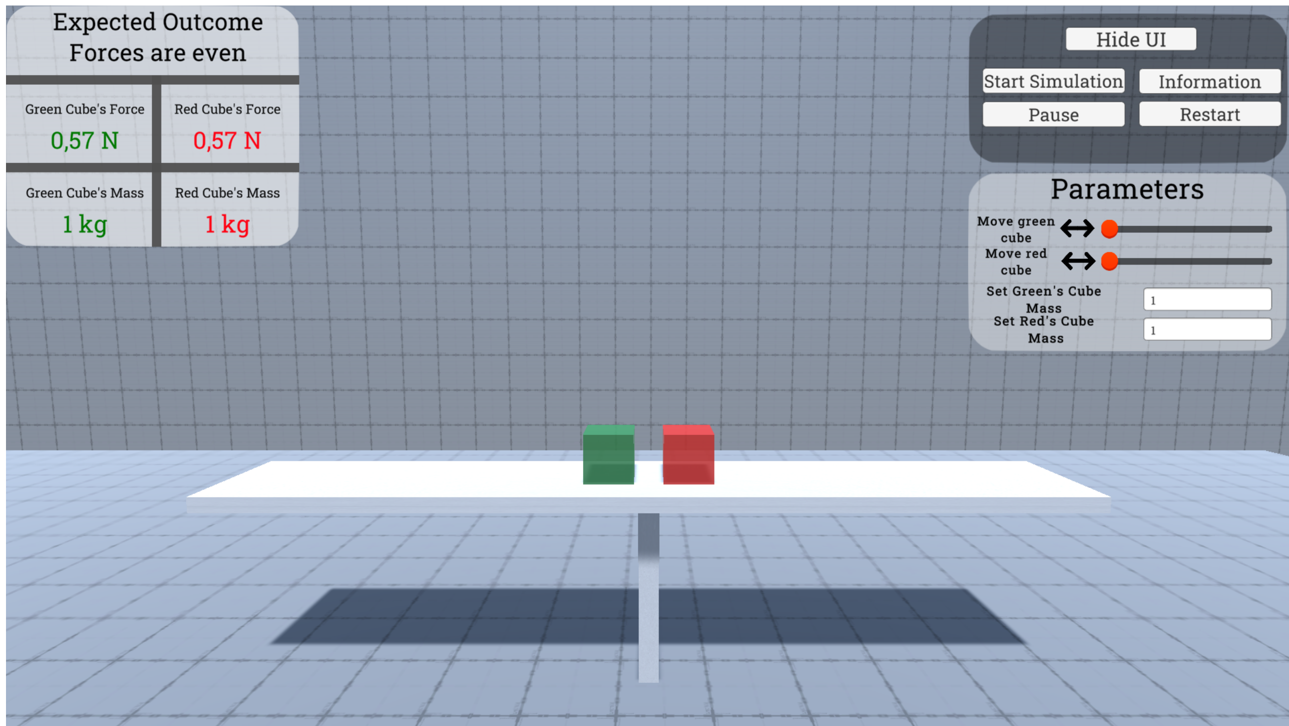


Figure 3.3: Lever demo.

The first demo represents a well-known physics problem lever. Levers can be used to lift heavy objects at one end of the lever by applying small force over a larger distance at the other end of the lever. Let me illustrate it on a seesaw 3.5. In the picture, forces F_1 and F_2 are not equal and the green cube will fall. Suppose forces to be equal:

$$F_1 = F_2$$

substitute $F_1 = m_1 d_1 \wedge F_2 = m_2 d_2$

$$m_1 d_1 = m_2 d_2$$

solve for m_1

$$m_1 = \frac{m_2 d_2}{d_1}$$

any cube with m, $m > m_1$ will be able to lift the second cube if placed on first cube position

F_1, F_2	Forces of the cubes
m_1, m_2	Masses of the cubes
d_1, d_2	Distances of the cubes from the middle point

Demo Figure 3.3 contains a seesaw on which are green and red cubes. These cubes can be moved left-right by using sliders. Cubes are connected to the seesaw with a fixed joint that is

Modifiable parameters	
Set the cube's mass	Set the position of the selected cube in KG
Move the cube	Set the position of the selected cube

Table 3.1: Lever modifiable parameters.

impossible to break. In Unity, fixed joint properties break force and break torque are set to infinity. The movement of the seesaw is simulated with a hinge joint which is also unbreakable. The user can define the mass of the cube. Default mass is set to 1.

The green and red cube force label changes color accordingly to the expected output of the simulation. When the green cube force is greater than the red cube force, the green cube force label turns green and the expected output is that the green cube lifts the red cube 3.4. When forces are too close to each other, the expected output might be inaccurate because of the sleep threshold. The sleep threshold in Unity is a threshold below which Unity objects fall into a sleep mode. That means Unity stops processing sleeping rigidbodies. In this demo, the seesaw stops the motion and the output is the same as if the forces of each cube were equal even though they are a little different.

Figure 3.4: The green cube's force is larger and will overweight the red cube.

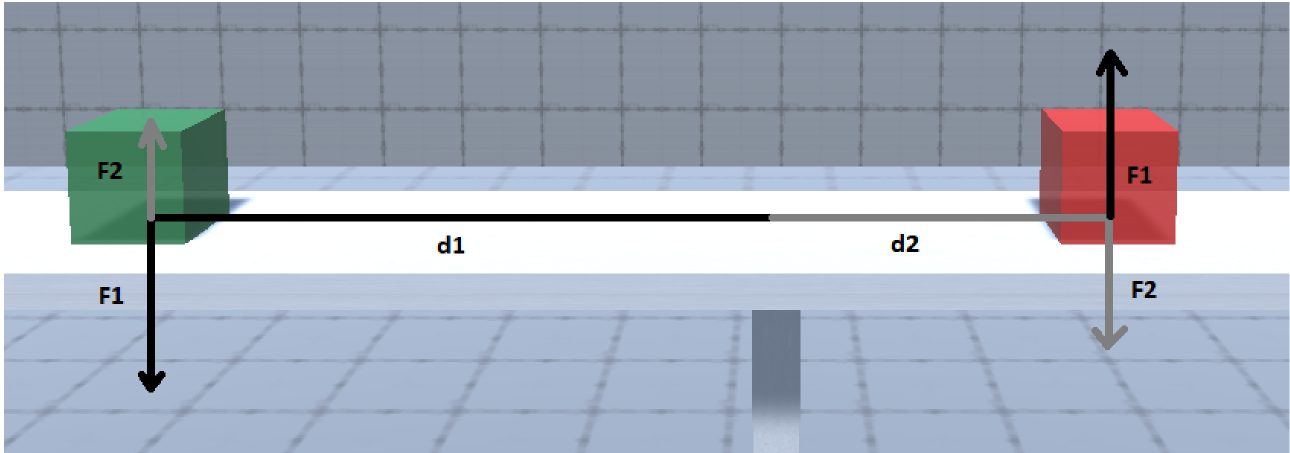


Figure 3.5: Seesaw illustration.

3.3 Conservation of Angular Movement

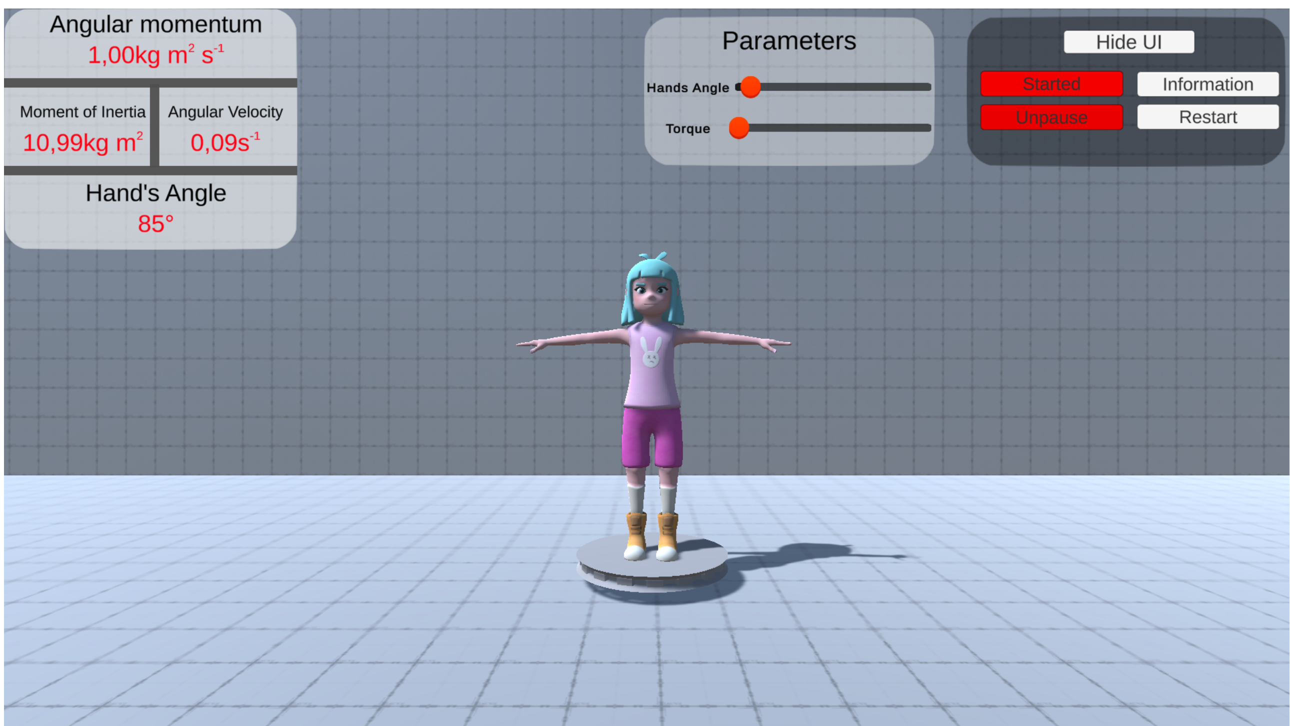


Figure 3.6: Conservation of Angular Momentum Demo.

This demo simulates a less-known physics problem – conservation of angular momentum. Angular momentum [18] is vector quantity that describes an object in circular motion. It is calculated as a product of moment of inertia and angular velocity.

$$L = I\omega$$

Modifiable parameters	
Hands Position	A slider to move the hands of the statue
Torque	A slider to change the amount of torque applied to the statue

Table 3.2: Conservation of Angular Momentum modifiable parameters.

L	Angular Momentum
I	Moment of Inertia
ω	Angular Velocity

Angular velocity is the speed at which an object rotates around the rotational axis and the moment of inertia determines how much torque is needed to achieve particular angular acceleration. The larger moment of inertia the more difficult it is to set the object in motion. Furthermore, the larger the moment of inertia the harder it is to slow down the object. Objects with the majority of the mass located further away from the rotational axis have a larger moment of inertia than objects that have the majority of the mass located close to the rotational axis.

Because the angular momentum is conserved, by changing the moment of inertia, the angular velocity is also changed to preserve angular momentum. This demo illustrates it in a simple example Figure 3.6. The girl statue is rotating on the stone stand. Users can move statue's hands closer to or further away from the body and change default torque. When hands get closer to the body, the moment of inertia gets lower, and thus the angular velocity must get higher and the statue rotates faster.

When moving hands, rigidbody property - inertia tensor is changed. In Unity, the inertia tensor determines only how much torque is needed in order to achieve a given angular acceleration. A change of the inertia tensor does not have an immediate effect on the angular velocity. To achieve realistic behavior while moving the hands of the statue, every physics frame, angular velocity had to be set to zero and then torque had to be added as an impulse.

(a) Moment of inertia is larger - mass of hands is further from the axis of rotation.

(b) Moment of inertia is smaller.

Figure 3.7: Moving hands changes moment of inertia.

3.4 Tunneling

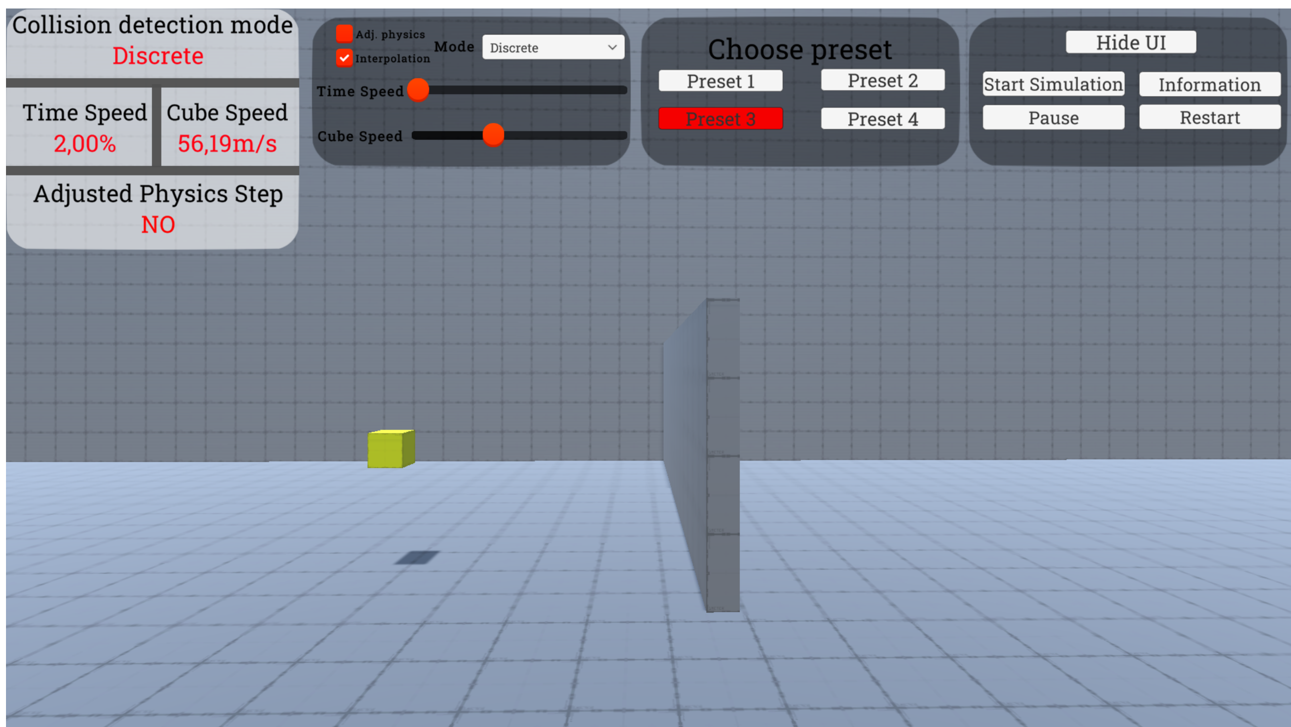


Figure 3.8: Tunneling Demo.

This demo tackles the problem of the tunneling effect. The tunneling effect happens when a collision is not registered where we would expect it to be detected. In games, it mostly happens with fast-moving objects that collide with thin objects, for example, a bullet going through a wall.

In Unity default collisions are handled discretely. That means in every physics frame all objects with rigidbody colliders are checked for collisions with other objects. However, if physics

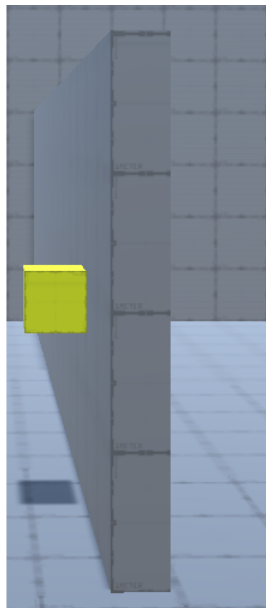
Modifiable parameters	
Mode	Set the collision detection mode of the cube
Cube speed	Set the velocity of the cube
Time speed	Set the speed of time
Interpolation	Toggle the interpolation of the cube
Presets	Choose between multiple presets
Adjust physics	Add more physics steps in relation to the time speed.

Table 3.3: Tunneling modifiable parameters.

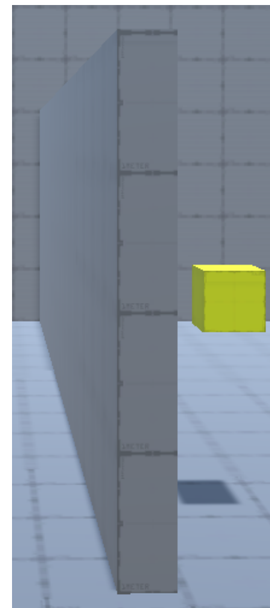
frames are too far apart from each other, collisions might be missed entirely, as illustrated on 3.9.

The tunneling effect is mainly solved by introducing continuous collision detection. This mode of collision detection calculates the time of impact to register collisions between frames.

There are four presets representing different problems in this demo. The first preset showcases the tunneling effect. The second one shows how the change of collision detection mode fixes the problem of tunneling. The third preset illustrates the difference between rendering and physics frequency when interpolation is turned on. The box moves into the wall because the collision is not registered at the surface of the wall but inside of the wall. The last fourth preset address the problem where a collision is registered but wrongly interpreted. This happens when the pivot point of the box is inside the wall collider during the collision. Zero penalty forces are applied, and the moving object passes through the wall as if no collision happened 3.10.



(a) First frame.



(b) Second frame.

Figure 3.9: Undetected collision of a fast moving object.

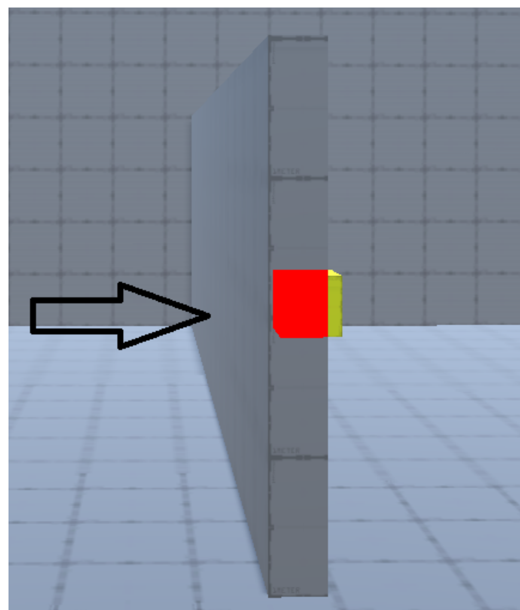


Figure 3.10: The box is approaching the wall from the left. Despite a collision being registered, the box still moves through the wall.

3.5 Collision of Two Objects

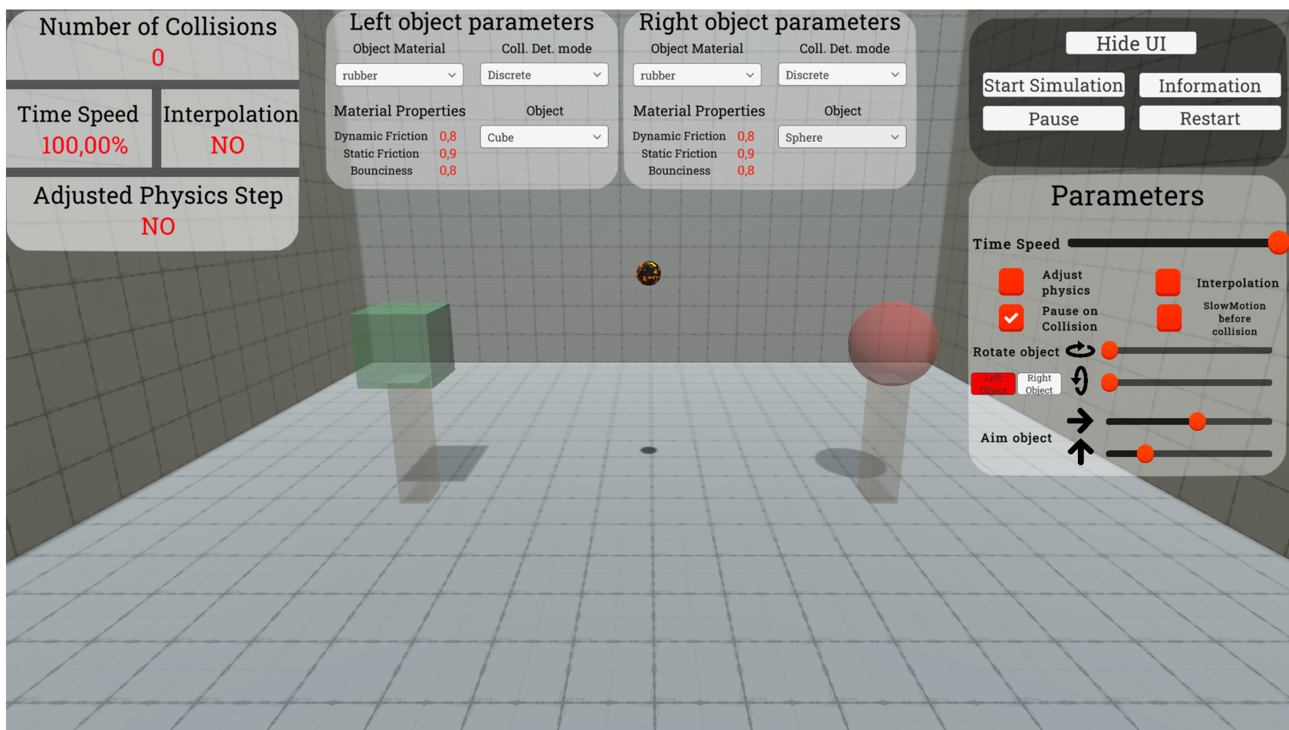


Figure 3.11: Collisions of Two Objects demo.

This demo focuses on collisions in Unity. The demo consists of two objects that can be changed by the user into a cube, sphere, or capsule. In the middle is a sphere on which both objects are shot at when the simulation is started.

By default, the simulation is paused when any collision occurs. At every collision, collision points and normals are created 3.12. Black vectors are collision normals and the orange vector is a velocity vector of the object. Collision normals are being used in order to calculate impulses after a collision occurs.

Simulating at a normal time scale produces unrealistic collision contact points as can be seen here 3.13. Inaccurate collision is the result of a large velocity of colliding objects combined with a low amount of physics steps in a second 3.13a. However, by adding more physics steps we can achieve a more precise collision 3.13b.

Turning the interpolation on may cause the same problems as in the tunneling demo. When slowing down time objects will be seen as fluid on-screen. However, objects will sublime into each other before a collision occurs because of the lack of physics steps.

Objects can be assigned different physics materials. Users can read its properties and experiment with it. Every object can be given a different collision detection mode. However, collision detection modes have little to no visible effect in the simulation apart from the continuous speculative mode which tends to detect collisions earlier and further apart from the

Modifiable parameters	
Material	Physics material of the selected object
Collision detection	Collision detection mode of the selected object
Object	Select an object to simulate
Time scale	Set time speed in percents
Interpolation	Turn on/off interpolation of both objects
Adjust physics	Add more physics steps concerning the time speed.
Rotate	Rotate selected object
Slow motion	Slow time shortly before collision
Aim object	Change position of aim object
Pause on collision	Simulation is paused at every collision

Table 3.4: Collisions of two objects modifiable parameters.

expected outcome. The illustration of the difference can be seen here 3.14

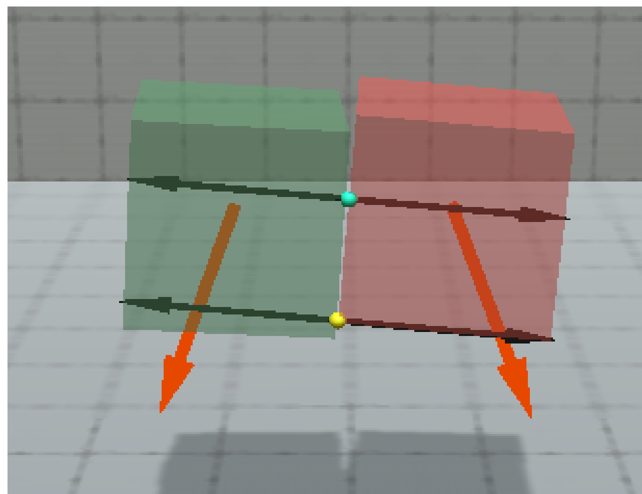
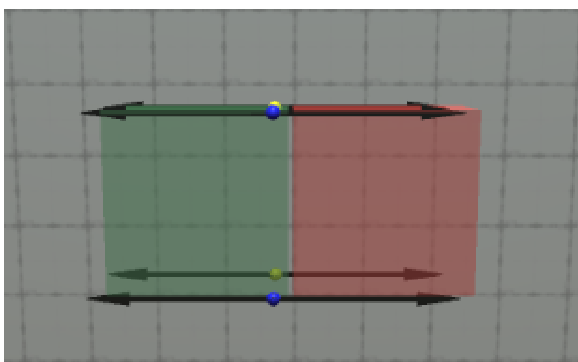
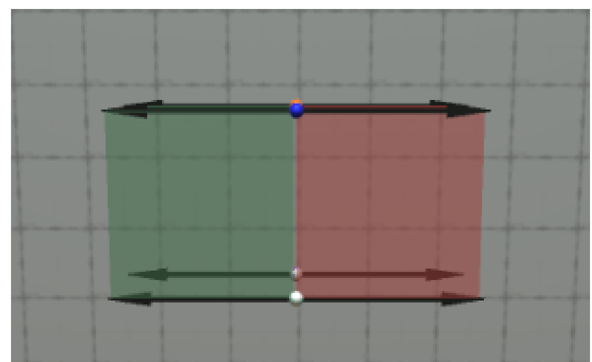


Figure 3.12: Collision.

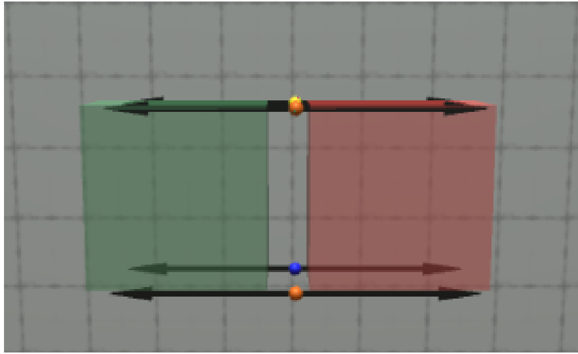


(a) Inaccurate collision.

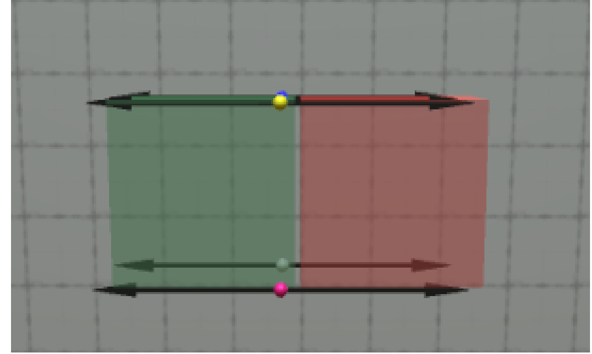


(b) Accurate collision.

Figure 3.13: Accuracy of collision.



(a) Speculative continuous detection.



(b) Discrete detection.

Figure 3.14: Difference between the continuous speculative and the discrete collision detection mode.

3.6 Solar System - Centripetal Force

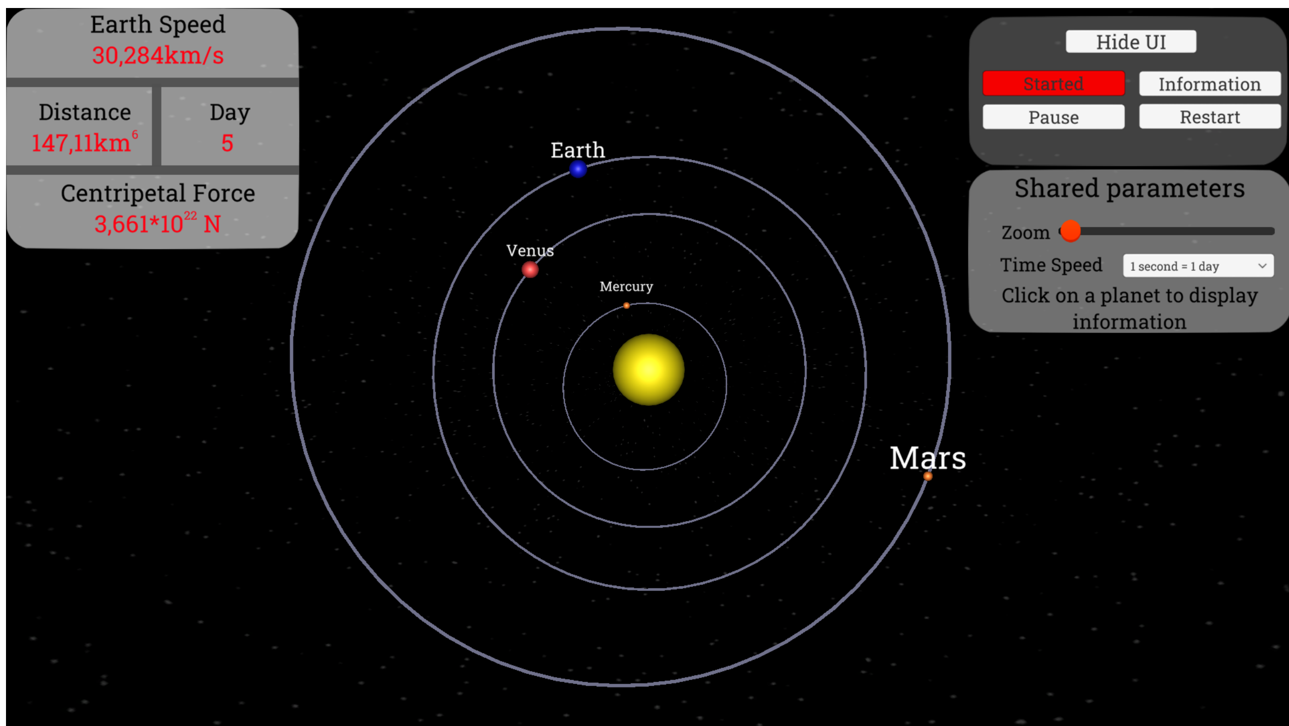


Figure 3.15: Solar System - Centripetal Force demo.

In this demo, the user can observe our Solar System. The Solar System consists of eight planets - Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune.

The Solar System is viewed from the top and can be zoomed in and out. The default speed of simulation is that one second equals one day in the simulation. The scale of time can be changed in the UI. The simulation keeps the track of the current day (the simulation starts from day zero). Apart from the initial positions of planets and the size of the Sun (the Sun

is 27.85 times scaled down), the simulation is realistic and real values from NASA [19] were used. The size of the Sun was diminished in order for other planets to be visible. By clicking on a planet, information in the top left corner about current speed, distance from the Sun and centripetal force will be displayed.

Centripetal force is a force that makes a body follow a curved path [20]. Centripetal force is always orthogonal to the velocity vector and is orientated towards the center of the curved path. Centripetal force can be visualized in many ways. In the Solar system, the centripetal force can be imagined as a string connecting the sun with other planets.

To create the simulation, it is possible to simply use the formula of gravitational force with a given initial velocity and direction.

Formula of gravitational force:

$$F = G\frac{m_1 * m_2}{r^2}$$

G	gravitational constant,
m_1	mass of first celestial object,
m_2	mass of second celestial object,
r	distance between celestial objects.

We calculate the gravitational force between each pair of celestial objects and then at every physics step, those forces are added to the velocities of celestial objects. To prevent all planets from falling into the Sun at the beginning of the simulation, we have to give all Planets initial velocities. One possibility is to give them initial force according to the "circular orbit instant velocity formula":

$$V = \sum_{n=1}^{i} \sqrt{\frac{G * m_i}{r}}$$

V	initial velocity,
G	gravitational constant,
m_i	mass of i_{th} celestial object,
r	distance between i_{th} celestial object and object we calculate V for.

This is everything that is needed to achieve an orbit around the Sun. However, this simulation is not realistic by any means. Even if real parameters such as mass, distances, size are used, the orbit can be inaccurate due to imprecise physics calculations. Also, it would be difficult to increase/decrease time speed. As in Unity, you can speed up the time only one hundred times and in connection to the Solar System the maximal speed up is negligible. To create a precise replica of our Solar System, it is better to use Keplerian elements and solve the simulation analytically. By using this method, it is easier to change time speed and by using real values from NASA [19] it is possible to make the simulation more realistic. The analytical

approach advantage is that the planets will always stay on the correct orbits regardless of the Unity engine settings.

First, it is necessary to calculate anomalies. Mean, eccentric and true anomalies are angular parameters that give us information about the position of the orbiting object in an elliptic orbit. Mean anomaly [21] is an angle between the orbits center and a point on a circle with a radius of a semi-major axis. If the orbit was circular, the point would stand for the position of the orbiting object. To calculate mean anomaly we use this formula:

$$M = M_0 + n(t - t_0)$$

M	mean anomaly,
M_0	mean anomaly at t_0,
t_0	reference time,
n	mean angular motion,

where n is computed as:

$$n = \sqrt{\frac{\mu}{a^3}}$$

μ	standard gravitational parameter,
a	lenght of semi-major axis.

To calculate eccentric anomaly [22] we use formula:

$$M = E - esinE$$

M	mean anomaly,
E	eccentric anomaly,
e	eccentricity.

This equation cannot be directly solved for E and root-finding algorithm had to be used. In this case Newton's method. After computing eccentric anomaly, we can finally compute true anomaly [23]:

$$v = 2arctan\left(\sqrt{\frac{1+e}{1-e}}tan\frac{E}{2}\right)$$

v	true anomaly,
E	eccentric anomaly,
e	eccentricity,

and distance of the orbiting object to the Sun [22]

$$r = a(1 - ecosE)$$

Modifiable parameters	
Zoom	Zoom closer to or further from the Sun
Time Speed	Set the scale of time
Click on a planet	Display information about the selected planet

Table 3.5: Solar System modifiable parameters.

r distance,

E eccentric anomaly,

e eccentricity.

After this, we can calculate the final position of planets [24] in relation to time. Other elements that are needed to ensure realistic simulation consist of semi-major axis, eccentricity, inclination, the longitude of ascending node, and argument of periapsis. Factual values were used based on NASA observations [19].

3.7 Destructible Objects

Figure 3.16: Destructible Objects demo.

This demo Figure 3.16 represents a way in which destructible objects in Unity can be created. Users can choose the left and the right objects (Cup, Chair, Table). Every object is created from fragments that are connected with fixed joints. Fixed joints work as an invisible firm line

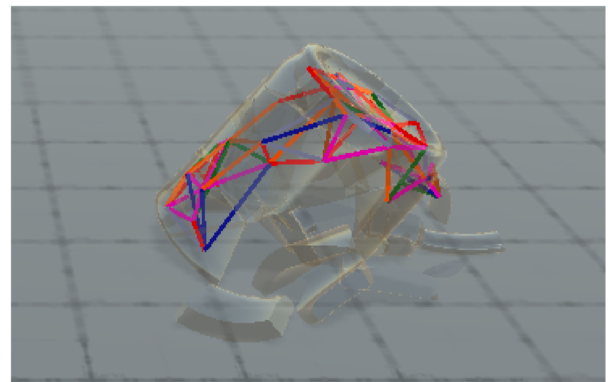
between two objects that keep their position towards each other the same. Fixed joint connections can be destroyed by applying higher force than the force set in fixed joints' properties. The default break force between these joints is set to fifty but it can be altered through the input field separately for the left and the right object.

Fixed joint connections are visualized with random colored lines 3.18a. These lines render as long as the connection between two objects isn't destroyed and visualization is toggled on 3.17. Fragments are visualized in a way that every fragment has random color 3.18b. It is also possible to toggle on/off this visualization.

For each object, the user can decide whether the rigidbody of the object is interpolated or not. When interpolating objects have fluid movement even when time is slowed. On the other hand, checking the interpolation option does not make the movement of visualized lines between shards fluid, because interpolation in Unity is only possible on objects that have rigidbody component attached to them.

Users can control the speed of time by moving the TimeScale slider. For instance, 100% is normal time, 200% is two times faster and 50% is two times slower. The option to adjust physics will make every movement of objects and visualized lines fluid, but the outcome of the simulation will be different for different time scales. This is because in Unity fixed delta time interval is not affected by the time scale and by adjusting fixed delta time manually, we add more physics steps to make every movement fluid and by doing so we increase the number of physics calculations and that makes the outcome non-deterministic for different time scales.

(a) Semi-broken cup without visualization.

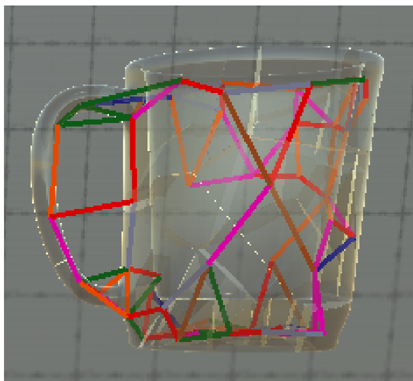


(b) Semi-broken cup with visualization.

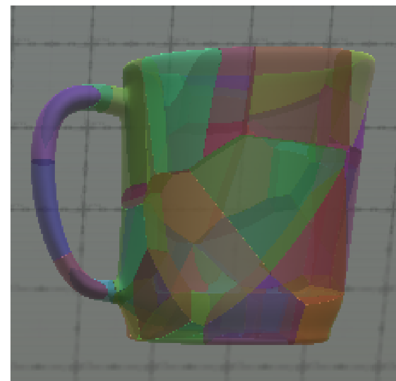
Figure 3.17: Broken fixed joints are not visualized.

Modifiable parameters	
Adjust physics	Add more physics steps in relation to the time speed.
Time Speed	Set time speed in percent
Interpolation	Turn on/off interpolation of selected object
Rotate	Rotate selected object
Break force	The force needed to break a connection between a joint and the connected fragment
Mass	Set mass of each fragment of the selected object
Visualize joints	Visualize connections between shards (between fixed joints)
Visualize shards	Visualize each fragment of object with a different color
Texture	Change the texture of the selected object
Object	Select a type of the object to be destroyed
Joints per fragment	Number that indicates how many other fragments is one fragment connected to

Table 3.6: Destructible objects modifiable parameters.



(a) Visualized fixed joints of the cup.



(b) Visualized fragments of the cup.

Figure 3.18: The visualization of a cup.

Chapter 4

Results and Discussion

This chapter will discuss a questionnaire that collected feedback on the user experience of the demos.

Demos were implemented in the Unity version 2021.2.13f1. Demos were built using WebGL and uploaded online to the website [17].

Demos were internally tested in Unity editor by me on a laptop with the GPU - GTX 1650 and the CPU - Intel Core i5–9300H

The questionnaire respondents tested all the demos on their computers and the internet browser they selected.

4.1 Testing the demos during the development

The questionnaire was not the only interaction between users and the developer during the time of the making of this project. The progress was continuously discussed with the thesis's supervisor or with my friends and family.

Some of the properties and features that were in previous versions of the project, but were later changed, based on the discussion with users, consist of different background, lightning, parameters, or readability of the UI.

The background in demos was initially made out of wood. Point lights and spotlights were illuminating the scene. However, the light combined with the material made it difficult to see the functionality of the demos and had to be changed.

In breakable objects, different textures of objects and the option to add more connections between fragments were added.

4.2 User Test

To collect user feedback on teaching demos, a questionnaire on Google Forms was created. At the beginning of the questionnaire, fifteen participants were requested to try out every demo online [17] After that, they were asked to answer the following questions of each part of the questionnaire. The questionnaire consisted of three parts. Overall, eleven out of fifteen participants filled out the questionnaire and in the upcoming section, their answers will be discussed.

4.2.1 General information part

General Questions		
Question	Yes	No
Are you familliar with technologies?	6	5
Are you familiar with programming?	1	10
Are you interested in physics?	4	7
Do you have an idea how physics-engines work	3	8
Do you have any previous experience with educational tools?	5	6
Have you heard of Unity game engine before?	6	5

Table 4.1: General Questions.

In this part, yes/no questions, that can be seen in the table 4.1, were asked to get a general idea about respondents.

Based on answers provided by respondents. We may see that the majority of them are not familiar with programming. Only one respondent out of eleven is familiar with programming. This is important to bear in mind in the next parts of the questionnaire as respondents might have general knowledge about physics, technologies, or Unity. However, they lack the general idea of how are the demos tackled in the code.

4.2.2 Separate demos part

General questions were followed by questions that were specifically aimed at each physics demo.

To be able to compare all the demos effectively, five identical questions for each demo were asked. Questions were not required and thus the number of respondents on demos sometimes differs.

Question I.

Did you learn anything new from this demo?		
Demo	Yes	No
Destructible Objects	9	1
Collisions of Two Objects	8	2
Tunneling	6	4
Conservation of Angular Momentum	6	4
Lever	5	6
Solar System	10	1

Table 4.2: Question No. 1 - Did you learn anything new from this demo?

Answers, that can be seen in the table 4.2, turned out as expected regarding respondents' knowledge. Respondents learned more from demos that tackled physics problems in game engines and less from demos that dealt with common real world physics problems.

However, there were two exceptions, namely Tunneling and Solar System demos. Many respondents did not learn anything new from the tunneling demo, which was surprising to me as it is a problem about something that the majority of people not interested in game physics not know. Even though the basic functionality of the Solar System is known by the majority of people, in-depth understanding is not and that might have been the reason why ten out of eleven respondents learned something new.

Question II.

Have you noticed any change in behaviour of selected objects after adjusting given parameters? (If yes, write below)			
Demo	Yes	No	Feedback
Destructible Objects	9	1	7
Collisions of Two Objects	8	2	6
Tunneling	6	4	4
Conservation of Angular Momentum	6	4	5
Lever	5	6	5
Solar System	10	1	5

Table 4.3: Question No. 2 - Have you noticed any change in behaviour of selected objects after adjusting given parameters?

The second question was a yes/no type question with the possibility of describing the answer in detail. The feedback in the table 4.3 is the number of respondents that answered yes and also provided feedback in a form of a text answer.

In Destructible objects, respondents often noticed the connection between the durability of the objects and the break force of fixed joints.

As for the collision of two objects demo, one respondent said: "Behaviour depends a lot on the object's shape and material" or another said: "If I rotate, the object gets a big rotation after the impact, it's dense". Both of these answers imply that respondents tend to notice changes after making only slight adjustments to parameters.

In the tunneling demo, users noticed the difference in collision detection modes and adjusted physics. For example, one respondent said: "Turning on adjustable physics made object move far more smoothly, collision detection mode made the biggest difference", followed by another respondent saying: "Adjusted physics and continuous/discrete mode have a big impact on behavior at the time of the wall collision"

Regarding the demo of Conservation of Angular Momentum, users observed the change in the speed of the rotation when adjusting the angle of the statue's hands. One respondent concisely said: "Speed of the rotation".

In the lever demo, respondents were mostly aware of the change of the force based on the distance from the middle point of the seesaw.

Finally, in the Solar System demo, users noticed the change in the speed of the simulation when adjusting the time parameter.

Question III.

Is there anything interesting you came across or learned while using this demo?	
Demo	Feedback
Destructible Objects	8
Collisions of Two Objects	4
Tunneling	6
Conservation of Angular Momentum	6
Lever	4
Solar System	6

Table 4.4: Question No. 3 - Is there anything interesting you came across or learned while using this demo?

In this question 4.4, respondents were asked to provide feedback on anything interesting they came across while trying the demos. Most respondents gave a feedback on the first demo. For example, one user said: "It's fascinating, that every destruction happens a bit differently, even though the same parameters are in use." Which is an excellent observation of inconsistency in Unity physics.

Concerning the Tunneling demo, one respondent said: "How much difference collision detection modes can make. They decide, whether the object goes through or not." Which is exactly what the demo was supposed to show and explain.

As for the Solar System demo, respondents mostly said that it was interesting to find out real values of the speed of planets. For instance, one respondent said: "It was interesting comparing the speed of movement of different planets, comparing those in the inner part of the solar system to the outer planets. I really loved the function of clicking on each planet to show more information about its speed, etc."

Question IV.

Is there anything you would improve on this demo?			
Demo	Yes	No	Responses
Destructible Objects	7	2	9
Collisions of Two Objects	7	2	9
Tunneling	5	3	8
Conservation of Angular Momentum	3	4	7
Lever	3	2	5
Solar System	5	2	7

Table 4.5: Question No. 4 - Is there anything you would improve on this demo?

This open-ended question 4.5 collected more responses than the other open-ended questions. Probably because a lot of answers were simply stating: "No".

As for the first demo, one respondent said: "Parameters range and better explanation of their meaning." Where parameters range is a reasonable observation. Because, for example, when a mass parameter is set to a high number it is hard to estimate the value of other parameters that would affect the simulation.

In the collisions of two objects demo, one user suggested: "I would like to be able to change the initial speed/force of objects" and another user said: "When a collision is detected I would appreciate some kind of camera function, which lets you take look at the result from different angles." Both are good catches that could be implemented in future versions to amplify the user experience.

As for the Lever demo, one participant recommended adding a second wall: "Second wall and the option to change parameters after crossing the first one".

An interesting piece of advice regarding the lever demo was: "Make it possible to move cubes by writing numbers" Which would be a reasonable addition as moving cubes via sliders isn't as accurate as giving cubes fixed positions.

Question V.

What was your overall experience with this demo?					
Demo	Excellent	Exceeds exp.	Avg.	Below exp.	In need of major improv.
Destructible Objects	5	4	2	0	0
Collisions of Two Objects	7	1	3	0	0
Tunneling	5	3	2	1	0
Conservation of Angular Momentum	6	3	2	0	0
Lever	4	5	2	0	0
Solar System	10	0	1	0	0

Table 4.6: Question No. 5 - What was your overall experience with this demo?

This table 4.6 represents the number of respondents that rated their overall experience with particular demos. Users could choose from options - Excellent, Exceeds expectations, Average, Below average, In need of major improvements.

Overall, none answers would propose major improvements to the demos and there was only one participant who had below-average experience with the Tunneling demo.

All in all, the most uninteresting demos to respondents were the Tunneling and the Lever demos. Probably because the Tunneling demo was not providing many adjustable parameters and the Lever is a well-known physics problem that they already knew.

The most liked demo was the Solar System. Ten out of eleven participants found the experience excellent. Presumably, the simulation was straightforward, intuitive, and visually appealing.

4.2.3 Final thoughts part

At the end of the questionnaire, respondents were asked five summarizing questions.

What was your overall experience?				
Excellent	Exceeds exp.	Avg.	Below exp.	In need of major improv.
7	4	0	0	0

Table 4.7: Summarizing question No. 1 - What was your overall experience?

Overall, every respondent was satisfied with the demos. As can be seen in the table 4.7, seven out of eleven found it excellent and the rest had their expectations exceeded.

Did you find information button in each demo helpful?	
Yes	No
11	0

Table 4.8: Summarizing question No. 2 - Did you find information button in each demo helpful?

This was the only question 4.8 in the questionnaire where the answer was unanimous that the information button was helpful for every respondent.

Did you have any trouble using the user interface?	
Yes	No
3	8

Table 4.9: Summarizing question No. 3 - Did you have any trouble using the user interface?

Most of the respondents were not having trouble using the user interface 4.9. However, some users had problems with the full-screen mode.

Have this set of demos gave you an overall view on how physics in game engines work?	
Yes	No
10	1

Table 4.10: Summarizing question No. 4 - Have this set of demos gave you an overall view on how physics in game engines work?

This question had 5 possible answers, although three of them began with yes and the remaining two with no, the table 4.10 reduces the answers to yes and no. Six respondents said

that they understand most of the demos, but there are things, which they don't apprehend. Two answered that they have got a basic understanding of physics in game engines. The other two participants said that they understand in-depth, how physics works in every demo. Only one person stated that demos were entertaining, but he/she didn't get any understanding of physics in game engines.

To wrap up, 10 participants said that they got some level of understanding and only one did not.

Is there anything you would like to say about the project?
Responses
7

Table 4.11: Summarizing question No. 5 - Is there anything you would like to say about the project?

As a final question 4.11, respondents were asked if there is anything they would like to mention about the project.

Some of the responses clearly outline the experience with the demos: "It simply and clearly shows some relatively complex physical principles", "It's nice that in Unity is possible to create such interesting simulations, definitely continue with more projects!", "It was quite fun going through each of the demos and playing around with the parameters, etc. Thank you!"

4.2.4 Future changes

Regarding the six already existing demos, the Tunneling demo could also showcase collisions between two dynamic objects. In the destructible objects demo, objects could be rotated by mouse drag.

The most important changes prompted by the users in the questionnaire were the change in the range of parameters and different camera modes.

Concerning the range of parameters. There could be fixed ranges for certain parameters as it might not always be intuitive for everyone, for instance in the Destructible Objects demo 3.16, setting the break force to a high number makes other parameters have a little to no effect. Also creating multiple presets (similar to presets in the Tunneling demo 3.8) in more demos could be a possibility.

As for the camera modes, one option could be a free camera mode. The free camera could give better angles to examine the properties of each demo. For example, in the Collisions of Two Objects demo 3.11, users could see the collision from different angles and distances to get a better understanding of the internal work. In the Solar System demo 3.15 the free camera

could help to understand how spacious is the Solar System in reality.

Other possible future changes were also already discussed here 4.2.2

Chapter 5

Conclusion

In the first part of this thesis, I introduced readers to popular physics engines and explained briefly how they work. Then I talked about how is numerical integration used and its function. Afterward, I presented basic parts of Unity physics.

In the second part, I described in-depth the purpose and function of each demo. In the last chapter, I discussed the results of the user questionnaire.

To conclude, six unique demos were made. These demos target a broad audience to help them understand some of the basic physical concepts in Unity. I tried to shed some light on basic concepts of how physics engines work and I also described basic concepts of physics in Unity.

The first three demos represent common physics problems. These simulations indicate that real-world physics can be replicated in game engines. However, they might not always be accurate. The other three demos show a little bit more about collisions which are very important in today's video games.

In the end, this work might be used as an educational tool to teach basic concepts of physics, or it can serve as a closer look at Unity physics for curious users.

Bibliography

[1] *G-switch 3, serious games*, https://www.seriusgames.com/G-Switch3.html, last accessed on 04/24/22.

[2] J. Gregory, *Game engine architecture*. AK Peters/CRC Press, 2018.

[3] *Physics engine*, https://en.wikipedia.org/wiki/Physics_engine, last accessed on 04/16/22.

[4] I. Parberry, *Introduction to Game Physics with Box2D*. CRC Press, 2013.

[5] *How does a physics engine work?*, https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview, last accessed on 01/14/22.

[6] *Algorithms in game engine development*, https://www.haroldserrano.com/blog/algorithms-in-game-engine-development, last accessed on 04/11/22.

[7] *Runge-kutta method*, https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method, last accessed on 04/09/22.

[8] *Unity*, https://docs.unity3d.com/Manual/PhysicsSection.html, last accessed on 01/14/22.

[9] *Force modes*, https://docs.unity3d.com/ScriptReference/ForceMode.html, last accessed on 04/10/22.

[10] *How to use fixed update in unity*, https://gamedevbeginner.com/how-to-use-fixed-update-in-unity/, last accessed on 04/11/22.

[11] *Colliders*, https://docs.unity3d.com/Manual/CollidersOverview.html, last accessed on 04/10/22.

[12] *Amy character model*, https://www.mixamo.com/?page=1&type=Character, last accessed on 04/10/22.

[13] *Unity joint types*, https://docs.unity3d.com/Manual/Joints.html, last accessed on 04/11/22.

[14] *Collision detection modes*, https://docs.unity3d.com/ScriptReference/Rigidbody-collisionDetectionMode.html, last accessed on 04/11/22.

[15] *Interpolation*, https://www.britannica.com/science/interpolation, last accessed on 04/11/22.

[16] *Unity interpolation*, https://gamedevbeginner.com/the-right-way-to-lerp-in-unity-with-examples/, last accessed on 04/11/22.

[17] *Web pages with teaching demos*, https://dcgi.fel.cvut.cz/home/bittner/demos/PhysicsDemos/, last accessed on 04/11/22.

[18] *Angular momentum*, https://www.britannica.com/science/angular-momentum, last accessed on 04/13/22.

[19] *Planet fact sheet*, https://nssdc.gsfc.nasa.gov/planetary/factsheet/, last accessed on 04/16/22.

[20] *Centripetal force*, https://www.khanacademy.org/science/physics/centripetal-force-and-gravitation/centripetal-forces/a/what-is-centripetal-force, last accessed on 01/14/22.

[21] *Mean anomaly*, http://www.csun.edu/~hcmth017/master/node14.html, last accessed on 04/16/22.

[22] *Eccentric anomaly*, https://mathworld.wolfram.com/EccentricAnomaly.html, last accessed on 04/16/22.

[23] *True anomaly*, https://en.wikipedia.org/wiki/True_anomaly, last accessed on 04/16/22.

[24] I. P. Williams and N. Thomas, *Solar and Extra-Solar Planetary Systems: Lectures Held at the Astrophysics School XI Organized by the European Astrophysics Doctoral Network (EADN) in The Burren, Ballyvaughn, Ireland, 7–18 September 1998*. Springer Science & Business Media, 2001, vol. 577.

Appendix A

User manual

The project can be tried online on web pages that were created simultaneously with the project https://dcgi.fel.cvut.cz/home/bittner/demos/PhysicsDemos/

There are six different demos on this web page. By clicking on the demo's name you will get to the page with detailed information about the demo and with the "Try it yourself" button. This button starts the selected demo. To get the best user experience, the demos should be run in full-screen mode and ideally on a display with 1920x1080 resolution.

Demos were built using WebGL. Although the demos may run on mobile devices with android or iOS, they are not fully supported and may not work properly. To get the best user experience it is recommended to use a computer with Windows OS and one of these browsers - Chrome, Firefox, Internet Explorer, Opera, or Safari.

Information on how to use all the demos can be found on the web pages or in the information button in each demo.

Appendix B

DVD contents

Contents of the enclosed DVD are stored in this manner:

- readme.txt
  - Description of the DVD content.
  - The link to download the source code of the Unity project.
  - The link to the web pages.
- unity_scripts.zip
  - Extracted scripts from the Unity project.
- imgs.zip
  - Six FullHD images of the demos.
- web_pages.zip
  - Zip with the source code of the web pages.