Ville Tamminen

Combat hit detection system for 3D action role-playing games in Unity

Bachelor's thesis

Information and Communication Technology

Game Programming

2022

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Ville Tamminen
Thesis title	Combat hit detection system for 3D action role-playing games in Unity
Commissioned by	South-Eastern Finland University of Applied Sciences, XAMK Gamelab
Time	October 2022
Pages	54 pages, 1 appendix page
Supervisor	Niina Mässeli, Pekka Vilpponen

ABSTRACT

The objective of this thesis was to create a combat hit detection system in the Unity game engine that can be used in 3D action role-playing games. The thesis also explored game mechanics that require hit detection such as weapon damage and weak spots. This thesis emerged from a lack of tutorials that would explain the implementation of a hit detection system while utilising multiple colliders in characters. Modern games use multiple colliders to achieve more accurate hit detection. The hit detection system created in this thesis aspires to be similar to the Dark Souls franchise games.

The start of the thesis, a simple hit detection system was implemented for characters using one hurtbox, and this was later elaborated into the use of multiple hurtboxes. Hit detection has multiple steps, with each step improving the previous one by including more combat mechanics. Also, features of characters such as damage accumulation, weak spots and target differentiation were implemented.

This thesis presents two solutions for the creation of a hit detection system that relies on Unity's collision detection components. The first solution passes information of a weapon attack to a character which will make a final decision if the attack hit should count. The second solution is a rework of the first one with hit detection functionality integrated to the weapon's code. In this thesis, damage and character weak spots were also implemented, and a means to distinguish enemies from friendly characters during attack hits was created.

Keywords: hit detection, game programming, Unity, C#, action role-playing game

CONTENTS

ABBREVIATIONS

3D-model	Three-dimensional model
Box2D	2D Physics engine that is utilised by Unity's 2D physics
C#	A scripting language used in Unity
Collider	A component that provides collision detection
Dynamic	Collisions and movement are driven by a physics engine
GameObject	A container where components are gathered to create functionality
GUID	Globally Unique Identifier
Hitbox	A trigger collider used in things that can hit
Hurtbox	A trigger collider used for things that can get hurt
Kinematic	Collisions only happen with dynamic GameObjects, and movement must be handled by scripts
Mecanim	Unity's animation system
Mesh	Skin geometry made of vertices to represent a 3D model
Nvidia	A tech company specializing in graphics technology
PhysX	Physics engine made by Nvidia that is utilised by Unity's 3D physics
Prefab	A reusable asset of a GameObject
Ragdoll	A state where the character's body is completely limb
Rigidbody	A component that enables GameObject to be affected by physics
RPG	Role-playing game
Static	A GameObject with a collider but no Rigidbody. No collisions apply to it and the physics engine does not consider it to be moving.
Trigger Collider	Collider that doesn't physically react to collisions
Unity	A game engine made by Unity Technologies

1 INTRODUCTION

This thesis explores ways to implement a hit detection system that can be used during character fights in action role-playing games. The focus is on producing scripts that form a functioning hit detection system in the Unity game engine. Mechanics tied to hit detection, such as applying damage, are also explored in the context of hit detection working with multiple colliders.

The thesis was commissioned by XAMK Gamelab, located on the Kotka Campus of Southeast Finland University of Applied Sciences. This thesis aims to help students to understand the process of creating a functioning hit detection system for action RPG games.

The inspiration for this thesis came from action RPGs, notably the Dark Souls game franchise. There are many tutorials available that show how to make a hit detection system, but they only use one hurtbox per character. Modern games require more realistic and accurate hit detection, which can be achieved by using multiple smaller hurtboxes in character bodies.

The implementation of hit detection consists of multiple parts where each part is a direct continuation of the previous one. Each part addresses a problem that emerges in the previous part and provides a solution for it. The implementation process presented in Chapters 4 and 5 describes each aspect in detail.

2 FRAMEWORK OF THESIS

2.1 The objective of the thesis

The aim of this thesis is to produce a functioning hit detection system in Unity that utilises multiple colliders in characters. The system aimed for action role-playing games that usually involve a third-person view combat. There are many tutorials available that show how to make a hit detection system, but they use only one collider per character. This results in the most basic and rough hit detection. If only one collider is used for the character's hit detection, it is often shaped like a cylinder and so big it covers the whole character. The problem in this can be illustrated by imagining a sword striking forward under

a character’s arm without hitting him. Despite not visually hitting the character, the sword does go through the collider and therefore results in a hit. This will not feel right for the player. Modern require more realistic and accurate hit detection that uses multiple smaller colliders in characters' bodies. In addition to the aforementioned phenomenon, this thesis examines also other mechanics related to hit detection such as dealing damage upon hit.

2.2 Research methods

Primarily, this thesis aims to present information about the common practices in how physics and collision detection function in video games, and the components and methods Unity offers to implement a hit detection system. This thesis is mainly based on the Game Physics Engine Development by Ian Millington, Real-Time Collision Detection by Christer Ericson, and official Unity documentation. Optimization in games is always appreciated, so programming solutions that require the least processing power in a computer should be favored during implementation.

2.3 Research problems

The objectives this thesis tries to answer are 1. Process of creating a hit detection system that uses multiple colliders in Unity engine 2. Process of implementing damage dealing mechanics which are related to hit detection. In order to achieve this objective, possible ways to implement damage dealing must be taken into consideration when creating a hit detection system. The solution is expected to have levels of complexity depending on the requisites of an action role-playing game. These needs can be e.g. the number of characters fighting at the same time against each other, the number of attacks can they do at once, and the nature of the character to decide whether or not they should receive hits during attacks.

2.4 Characteristics of action role-playing games

Action role-playing games (RPG) are games that combine both action-orientated combat and role-playing. Unlike traditional RPGs where combat is turn-based or menu-based, action RPGs take combat to another level with fights played in real-time. The other RPG mechanics such as character levels/stats,

weapon upgrading, and trading with NPCs are presented in the traditional manner.

The developer studio of the Dark Souls game franchise, FromSoftware, released a new action RPG called Elden Ring in March 2022. The game has been overwhelmingly positively reviewed. Previously, all Dark Souls games have been reviewed positively on the Metacritic website which compiles reviews of games. Elden Ring received an average of 96/100 metascore on Playstation 5 and 94/100 metascore on PC (Metacritic 2022). The Dark Souls series is the source of inspiration for this thesis, and the final hit detection system is expected to resemble it to some degree.

Elden Ring’s combat mechanics can be summed up as attacking, blocking, and dodging. The Player character's attacks and actions can be halted by enemy attacks or by the environment. Receiving hit will result in staggering while by dodging the player becomes momentarily immune to receiving hits. The Player's melee attacks will go through the enemy's 3D-models, while hitting a wall will usually make attack animations halt, making the attack movement seem more realistic. Weapons include different kinds of melee weapons, bows, magic staffs, or throwable bombs/daggers.

Elden Ring is by no means the first game to utilise realistic hit detection. Video games have used physics and collision detection since the first games were created (Millington 2007, 1). Elden Ring’s combat style shares many similarities with Severance: Blade of Darkness game released in 2001, which has accurate body hit detection and even body-part slicing mechanics (Kasavin 2001).

2.5 Physics

Physics is a vast field with hundreds of subfields. Many aspects of physics can be utilised in games. For example, optics can be used to simulate how light travels and bounces, which is illustrated in the use of ray tracing. However, this is not what is meant when the reference is to game physics. Game physics relate to classical mechanics with gravity and other forces pulling or pushing objects. Mass, inertia, bounce, and buoyancy give life to objects to achieve

realistic simulations. As processing power increases, possibilities grow. Objects can be stacked, and walls can collapse into smaller pieces. These phenomena are part of Rigidbody physics which can also make softer objects such as clothes and ropes have realistic movement. Human characters can trip due to ragdoll physics which makes realistic joint movements possible. (Millington 2007, 2.)

Instead of coding physics into the game, some games use physics engines. Physics engines have two advantages: saving time and improving quality. Building physics into a game from scratch takes time, so sometimes it is better to use a premade physics engine. Having all physics run under one program is better than having multiple different programs running separately, as even if they may work flawlessly on their own, combining them might be difficult. (Millington 2007, 3–4.)

Physics engines also have disadvantages. It takes a significant amount of processing power to run a general-purpose physics engine. Because it is general, it cannot make assumptions about the kinds of objects it is simulating. If the game environment is very simple, processing power is wasted, which is particularly a problem for mobile devices. The physics engine operates by receiving data and creating a simulation based on that data. It may require data that is not relevant in a game but essential for the engine to function. This means it can sometimes be simpler to create a specific code for physics rather than use a physics engine. (Millington 2007, 4.)

Collisions are needed for objects to interact with each other in a simulation. Collision physics can vary depending on the types of colliding objects. Particle collisions are easiest to implement, as a basic particle system can be programmed in only a hundred lines of code (Millington 2007, 3). 3D objects, on the other hand, can be much more complex, varying from simple shaped spheres and boxes to complex shapes such as humanoid characters.

A 3D object's geometry is made of vertices. Together, vertices form polygons, and polygons form a mesh. Building objects from polygon meshes is one of the most common methods for creating geometrical models in games. Polygonal objects are said to have explicit representation. Implicit objects refer to

spheres, cones, cylinders, and other geometric primitives that can be defined by a mathematical expression. (Ericson 2005, 9.)

2.6 Collision detection

The process of collision detection can be the most time-consuming part of physics simulation. Any object in a game may collide with another object, and each event of collision must be examined. A game with hundreds of objects may require hundreds of thousands of verifications. Even worse, each verification must comprehend the geometry of the two objects, which may include thousands of polygons. This enormous workload cannot be finished in the fraction of time available between frames. (Millington 2007, 232.)

The collision detection system is responsible for calculating any geometrical properties, such as when and where two objects are in contact, and the contact normal between them. There are many different algorithms used for identifying contact points. Some collision detection algorithms can predict the likelihood of future collisions by looking at how objects are moving. (Millington 2007, 111.)

The process of identifying intersection between moving objects at specific moments in time is known as static collision detection. Each time this occurs, the objects are managed as though they were stationary with zero velocities. In contrast, dynamic collision detection considers the continuous motion of objects over a specified duration. Dynamic collision testing typically reports the precise moment of collision and the initial points of contact. Although static tests are less expensive than dynamic tests, the duration between tests must be brief enough for the movement of the objects to be less than their spatial extents. Otherwise, the objects can just pass one another from one time step to the next without a collision occurring. This phenomenon is called tunneling. (Ericson 2005, 17.)

In order to improve collision detection, the number of possible collisions should be decreased, and collision verifications should be made less expensive. In order to reduce the number of verifications, a two-step process can be utilised. First, groups of objects that are likely to be in contact with one another

are searched in the game. This eliminates the vast majority of possible collision verifications, and it is known as coarse collision detection. Inside said groups, objects are examined to determine whether they are colliding. This is known as fine collision detection. (Millington 2007, 232.)

A bounding volume is a 3D shape that contains an object. A simple shape, usually a sphere or a box, is used for coarse collision detection. Size of the shape should be sufficient to contain the whole object, and ideally as close-fitting as possible, as this further decreases unnecessary collision verifications. If two objects have bounding volumes that overlap, then these objects are likely in contact and will proceed to undergo fine collision detection. Fine collision detection takes into account the real shapes of objects to determine if they are in contact. Contact points are needed for Rigidbody physics. The geometry of an object is typically simplified to reduce the time required for verification. (Millington 2007, 233.)

Polygonal meshes and bounding volume hierarchies are regarded as the industry's golden standards for collision detection, but they both have drawbacks. Meshes must be extremely detailed to achieve high geometric precision, which slows down mesh-based algorithms, and bounding volume hierarchies are implemented for very quick, low-accuracy proximity queries by design (Gonçalves 2015, 57). This, however, is not as serious a problem in games as it is in robotics where the highest possible precision is required.

Coarse collision detection is not limited to using only bounding volumes. Many different approaches use spatial data structures. A bounding volume hierarchy groups objects together based on their relative positions and sizes. The hierarchy moves if the objects move. For different sets of objects, the hierarchy will have a quite different structure. An advantage of hierarchies is that whole branches can be excluded. A spatial data structure is locked to the world. An object found at some location will be mapped to one position in the data structure. No matter what objects are inserted into a spatial data structure, it remains the same structure, which makes it much easier to build. The distinction between bounding volume hierarchy and spatial data structure is blurred, and a combination of both techniques is sometimes used. It is worth noting that

even when bounding volume hierarchies are not used, it is common to use bounding volumes around objects. (Millington 2007, 251.)

3 UNITY ENGINE

Unity is a game engine developed by Unity Technologies. Unity was chosen as a game engine for this thesis for its familiarity and popularity. Unity uses C# coding language, and it has a large documentation manual. Contents of Unity Documentation may slightly differ depending on which Unity version is used, and this thesis uses the currently most recent long-term support version of Unity, which is 2021.3.0f1 from March 2022. Long-term support versions offer a stable base for game projects.

This chapter describes the operation of Unity's physics engine and methods or components that can be utilised in a hit detection system. The main contribution to the creation of hit detection in this thesis comes from Rigidbodies, Colliders, and collision/trigger-orientated methods.

3.1 Unity engine components and related terms

Everything in Unity's scenes is made of GameObjects: characters, items, scenery, lights, cameras, and special effects. They act as containers for components which give GameObjects their functionality. GameObjects always have an unremovable Transform component on them which dictates their position, rotation, and scale. GameObjects can have multiple components on them, and using different combinations can help achieve the desired outcome. Unity has several different built-in components, and making new components is possible with Unity's scripting API. (GameObjects 2022.)

In order to group GameObjects together, Unity uses the concept of parent-child hierarchies or parenting. Linking GameObjects together makes it easier to move, scale, or transform a group of them. Changing the Transform of the top-level GameObject, or parent GameObject, changes all child GameObjects. Other GameObjects can inherit the properties of a parent GameObject. The top-level GameObject is also referred to as root GameObject if it has child GameObjects. (Hierarchy 2022.)

Rigidbody is the most fundamental physics component in Unity. It allows GameObjects with Colliders to be affected by gravity and collisions or even forces via scripting. If the Rigidbody is set to be kinematic, it will not physically react to collisions, and it can receive trigger messages. If it is not kinematic, Rigidbody receives collision messages. Being kinematic also allows Rigidbodies to switch control from physics to animations. Rigidbody has a collision detection mode, which has four different modes: discrete, continuous, continuous dynamic, and continuous speculative. Discrete collision detection is the default setting and it is used against all other Colliders with Rigidbodies in the scene. Continuous collision detection is for GameObjects that can be expected to collide with fast-moving GameObjects, and continuous dynamic collision detection is meant for those fast-moving GameObjects. Continuous speculative collision detection is for static Colliders, and it tends to be less expensive than sweep-based continuous collision detection. (Rigidbody 2022.)

The Character Controller is a component meant to be added to characters. It makes it easier to implement movement that uses Colliders and gives basic Collider responses without having to use a Rigidbody component. It moves only when the Move function is called, and it is not affected by forces. The main advantage it provides is the amount of control over characters, but it also requires that everything is done with code. (IronEqual 2017.)

A collider is a bounding volume, an invisible perimeter that is used for collisions by a physics engine. Although it has a geometric look, it is not made from mesh data (Rodrigues 2018). Colliders are algorithmic and geometrically optimized verifications that use only the minimum required amount of data (Rodrigues 2018). They can have a simple primitive shape such as a box, sphere, or capsule, or it can be a complex MeshCollider which builds its collider based on a 3D model's mesh asset (Mesh collider 2022). Collider type will affect the number of contact points that occur during collisions. A SphereCollider usually has one contact point and a BoxCollider usually has one, two, or four (BEST PRACTICES FOR RIGID… 2014). Mesh colliders can have hundreds of contact points, so they provide more accurate collision detection at the cost of higher performance. Static Colliders are GameObjects that have colliders but no Rigidbody component attached to them. A common mistake is to move a static collider as this would mean that the whole game

world must be recalculated by the physics engine (Macek n.d.). Compound Colliders are made from multiple single colliders.

A hitbox is a collider with a trigger setting. A Trigger Collider lets other GameObjects pass through it since it does not physically react to collisions (Learning unity3d eBook, Chapter 7: Collision). A hitbox is designed to be used on weapons and items that deal damage. A hurtbox is similar to a hitbox, but it is designed to be used on anything that suffers damage, such as characters, to form a shape where damage is received (Ranney 2017). The distinction in names aims to make it less confusing when referring to weapon hitboxes hitting character hurtboxes. The use of the word “box” in these terms stems from the fact that in the early days of game programming, box shape was the least math-intensive way to calculate collisions.

GUID is short for Globally Unique Identifier and is also known as UUID (Universally Unique Identifier). A GUID is a 128-bit integer number used for unique identification (Guid struct 2022). The purpose of generating identifiers is to minimise the chance of having duplicates. An example of GUID in Unity can look like this: 9d28a64e-5e45-4971-80e0-1afb6ce75985. Letters appear in GUID because it uses hexadecimal numbers. Identifiers are important since GameObjects can have identical names.

Collision layers are used for layer-based collision detection. They can help determine which GameObjects should collide with which layer. If a layer is set to collide only with itself, the GameObjects on that layer can only collide among themselves. For example, if multiple colliders are used for characters, it could be useful to make sure hurtboxes do not collide with physics colliders.

3.2 Physics collisions

Unity has two built-in physics engines for object-oriented physics: Nvidia PhysX and Box2D. Unity’s built-in 3D physics uses Nvidia PhysX engine integration, and 2D physics uses Box2D engine integration (Unity Physics 2022). 3D physics will generally interact with meshes, while 2D physics will interact with 2D sprites which are only rectangular images. 3D physics have Rigidbody and Collider components, and 2D physics have corresponding Rigidbody2D

and Collider2D components. Unity even has a 3D physics simulation for clothing, which makes movement of clothes look realistic. For data-oriented projects, Unity offers the possibility to install the "Unity Physics" package or the "Havok Physics for unity" package. The implementation part of this thesis is done with PhysX.

Rigidbody physics is becoming more extensively used in games as technology advances and software capacity grows, as it allows for more varied and realistic simulation. It is important to keep the game's scale in mind when starting to make a physics-based game. PhysX can be heavy to process on mobile devices and, in general, an optimized game is more satisfying for players. In order to reduce the time and resources that physics simulation takes, it is advisable to simplify colliders. Mesh colliders can be expensive as they take the exact shape of GameObject’s 3D-model mesh. More complex mesh colliders can be substituted with primitive or simplified colliders, such as box colliders or sphere colliders to approximate the original shape. (Goldstone 2009, 42; Krogh-Jacobsen 2021.)

The aforementioned colliders will be used by either a Rigidbody or CharacterController components for movement. Collisions are needed to prevent GameObjects from falling through floors or going through walls. CharacterController will allow movement constrained by collisions without dealing with a Rigidbody. If physics are used in the game, a Rigidbody must be present to use the following collision verification methods: OnCollisionEnter(), OnCollisionExit(), OnCollisionStay(), OnTriggerEnter(), OnTriggerExit() and OnTriggerStay(). While the way movement is achieved can vary depending on which component is used, it will not matter as the same results can be attained by both CharacterController and Rigidbody (IronEqual 2017). This thesis is focused on trigger methods since the inspiration originally came from the Dark Souls game franchise in which weapons can go through enemy characters instead of physically reacting to their 3D-models. That is why a Rigidbody component is necessary for character GameObjects.

Physics.OverlapBox() is another component that is used to detect touching or overlapping colliders, similar to OnTrigger methods. It computes and stores all

colliders that are touching the box or are inside it. It allocates an array of colliding colliders on every frame, which generates garbage. OverlapBoxNonAlloc fixes this problem by storing collision data into a provided buffer. It does not attempt to increase the buffer if it runs out of space, which can be problematic. OverlapBox could be a viable alternative to OnTrigger methods, but this thesis is only focused on trigger methods.

Collisions have one parameter in their methods which is a class of collision type. The collision class holds information that describes the characteristics of the collision, such as contact points and impact velocity. Triggers, on the other hand, have one parameter in their methods which is a type of collider. This collider is a reference to the other collider that entered the trigger area. This makes it possible to manipulate the other collider's GameObject.

Depending on the Rigidbody arrangements of the colliding GameObjects, a variety of script events can occur upon collision. Figure 1 shows which event routines are called. The general rule is that physics will not be applied to a GameObject that does not have a Rigidbody component attached to it. Exceptions to this are combinations that only affect one of the two GameObjects. (Introduction to collision 2022.)

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		collision			trigger	trigger
Rigidbody Collider	collision	collision	collision	trigger	trigger	trigger
Kinematic Rigidbody Collider		collision		trigger	trigger	trigger
Static Trigger Collider		trigger	trigger		trigger	trigger
Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger
Kinematic Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger

Figure 1. Collider interaction matrix.

In a collision between two non-trigger colliders, the physics engine calls specific physic methods in involved GameObjects. The OnCollisionEnter() method is activated on the first physics update when a contact is detected. Next, OnCollisionStay() is called during updates while contact between colliders is

maintained, and OnCollisionExit() signals that the contact has been severed. If a collision happens between trigger colliders, similar methods are called. Triggers detect collisions like other colliders, but they simply do not react to them. There are similar functions for 2D physics, such as OnCollisionEnter2D(). (Introduction to collision 2022.)

Raycasting means shooting an invisible ray from an origin point to a direction, and having a report of anything that it hits. While this can be used to verify collisions, it fails to detect a hit if its origin point is inside a collider. Therefore, raycasting is not considered in this thesis, as it would be rather problematic in a close-quarter melee fighting game. The raycast method is more suitable for shooters where bullets fly fast.

3.3 Physics execution order

Everything runs in a predetermined order in Unity. In initialization, Awake(), OnEnable() and Start() functions are called first in that order. Common game tasks a.k.a. game logic are performed in the Update() function, but before that, another function called FixedUpdate() is actuated. While Update() is called once per frame, FixedUpdate() is often called more frequently. FixedUpdate()'s calling rate is determined by a fixed timestep and it can be called multiple times in a frame if necessary. All physics calculations run immediately after FixedUpdate(). FixedUpdate() is called on reliable time independent of the game's frame rate, which means movement calculations do not need to be multiplied by Time.deltaTime inside it. (Order of execution for event functions 2022.)

The upper part of Figure 2 illustrates Unity’s physics cycle, and it can be seen that FixedUpdate() is called first. Triggers and collisions run at the physics cycle’s end. The lower part of Figure 2 presents the game logic cycle, where Update() is called first, followed by WaitForSeconds() and coroutine functions. LateUpdate() is called last in the game logic cycle, and it is most commonly used to update camera position and rotation after character movement. After LateUpdate() come different rendering functions followed by decommissioning function which processes quitting, disabling, and destroying. Animation updates are present in both physics and game logic cycles.

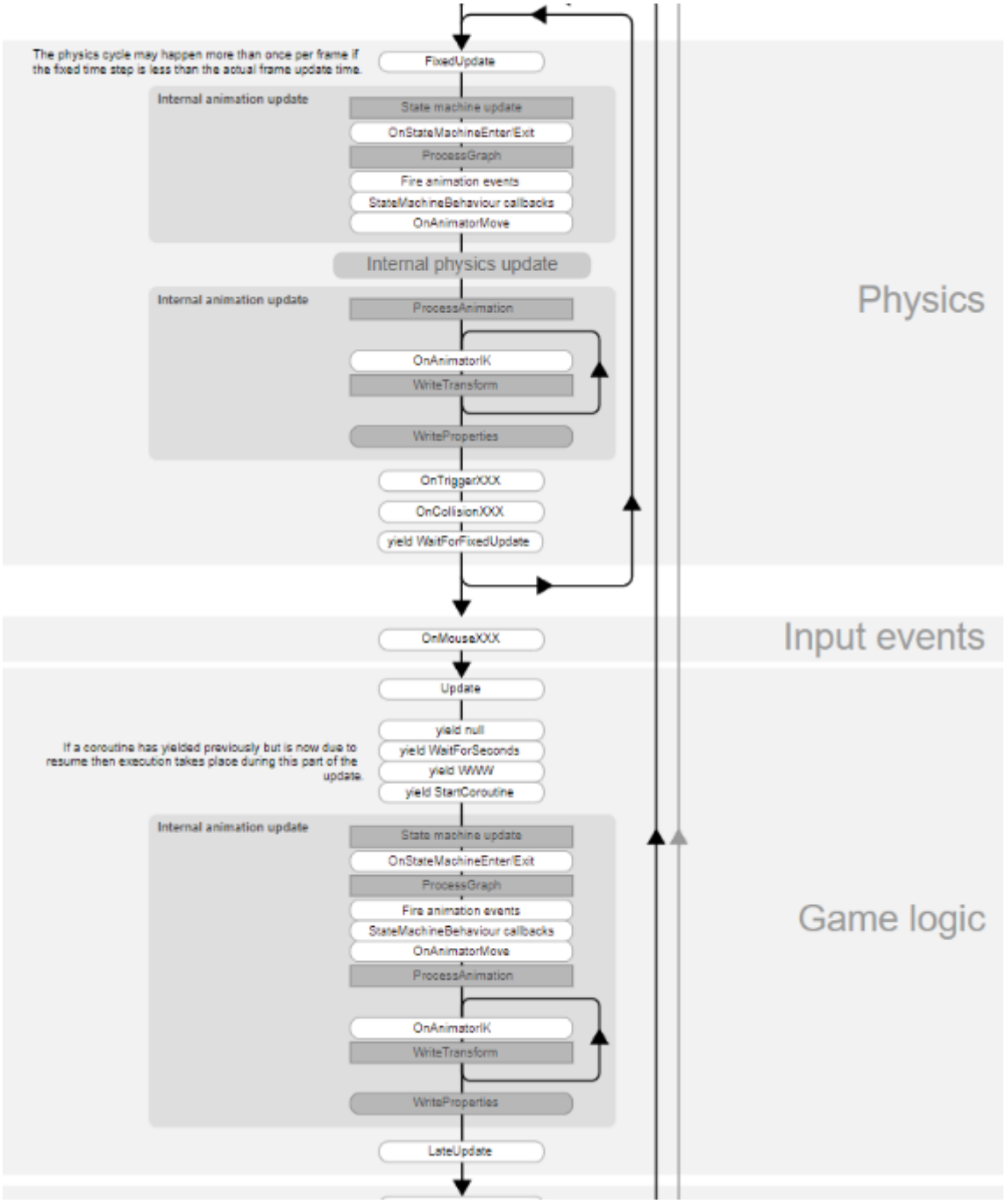


Figure 2. Unity's physics and game logic.

3.4 Finding and calling methods

Calling a method inside a class is as simple as calling its name, but calling a method from another class can be done in many ways. First, the other class must be found. If the class script is in the same GameObject, it can simply be assigned to a public field in Editor or be discovered with GetComponent() method.

Because there are numerous versions of the GetComponent() method, each with a different performance cost, it is best to use the one with the lowest performance cost. The available alternatives are GetComponent(string), GetComponent(), and GetComponent(typeof(T)). Because numerous optimizations have been made to these methods over the years, the fastest version depends on whatever version of Unity is used; nonetheless, in all later versions of Unity 5 and the first release of Unity 2017, it is better to utilise the GetComponent() variant. (Dickinson 2017, 44.)

This thesis uses Unity version 2021.3, thus GetComponent() variant will be used if necessary. Overall, these are micro-optimizations, and the changes will not give any perceivable performance difference during runtime (Unity performance tests 2015).

If the class script is in another GameObject, it must be found so that its public methods can be called. Collision methods, such as OnTriggerEnter(), know when a collision occurs with a GameObject that possesses a collider. Child GameObjects can also trigger OnTriggerEnter() methods but only if the parent GameObject with OnTriggerEnter() method has no colliders. The place of trigger method scripts in the hierarchy must be planned accordingly if child colliders are also used. The number of colliders on a GameObject does not affect trigger methods. OnTriggerEnter() can be used with GetComponent() to acquire the desired class script reference from the GameObject with which the collision occurs. Next, public methods can be called from the other collider's class. This can be done directly by calling the class name paired with the method name.

3.5 Mecanim animation system

Unity has its own animation system which is sometimes referred to as Mecanim. Mecanim has a visual editing window similar to a flowchart where single animation clips are joined together to form an animation state machine. (Animation State Machine 2022.)

There are multiple types of animations in Unity: Rigidbody animations, rigged/bone-based animation, Sprite animation, physics-based animation,

morph animation, video animation, particle animation, and programmatic animation. When animating humanoid characters, animals or monsters, rigged animation a.k.a. bone-based animation is required. Rigged animation does not alter the GameObject's position, rotation, or scale, but the movement and deformation of its internal parts between keyframes. It uses special bones to simulate the skeleton of the mesh, which allows independent control of the mesh geometry. This is very useful for animating limb/head/mouth movements. Usually rigged animation is made as a complete animation sequence in 3D modeling software and then imported to unity with a mesh. (Thorn 2015, 5.)

4 HIT DETECTION IMPLEMENTATION

The implementation of hit detection consists of multiple solutions that each have multiple parts in them. Each part is a direct continuation of the previous one. Every part answers a problem that emerges in the previous part and provides a solution for it while trying to keep Elden Ring's playstyle and mechanics in mind. In order to make explaining simpler, the character that attacks with a weapon will be referred to as the attacker, and the character that receives hits will be referred to as the target.

4.1 Solution 1

Solution one of hit detection implementation starts by creating a simple hit detection code that exchanges information of collisions between weapons and characters, and expands it further to make it suitable for Elden Ring combat mechanics.

4.1.1 Setup preparations

This thesis uses 3D-models from "POLYGON Knights - Low Poly 3D Art by Synty" which can be bought from Unity Asset Store. Animations are obtained from mixamo.com which has a wide variety of free animations. Using ready-made assets allows more time to focus on writing code. When knight models are viewed in Unity, an Animator component is already attached in the 3D-models GameObject. If a model without Animator is used, then one must be

attached to the model. New Animator controllers are created for two characters, and different animations are added to them. With these in place, a testing setup was made where the attacker is given a continuous looping sword attack animation while another one stands idle and takes hits (Figure 3).

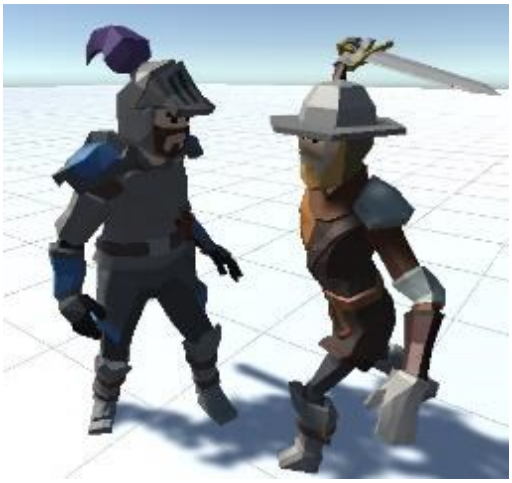


Figure 3. 3D-model setup screenshot.

A hitbox is added to the weapon GameObject which will be used with a script. The script needs to be on the same GameObject that uses the Animator component, so it will be set on the attacker's top-level GameObject. In Unity, a hierarchy window is used to display GameObjects in a scene. In Figure 4, the top-level GameObject's name is Character 1, the visible 3D-model is named Character_soldier_02, GameObject called Root refers to the root of all bones used by Character 1, and the weapon with a hitbox is named SM_Web_Broadsword_01. As for the latter, while a simpler name would be more memorable, this weapon came with POLYGON Knights pack, and it is left as it was so that its origin is not forgotten. From **Window>Animation>Animation,** an Animation window can be opened where animation events can be added to animation clips. These animation events will call a script (Figure 5) to activate and deactivate the sword hitbox during attack animations. This way the weapon hitbox cannot hit anything when it is off.

A minor problem occurred when animation events could not be added to animations because FBX-skin (skinned Mesh format supported by Unity) was attached to the imported animations as well. If this happens, the animation clip can be disconnected from the FBX-skin by duplicating the clip or by re-importing the animation without skin.

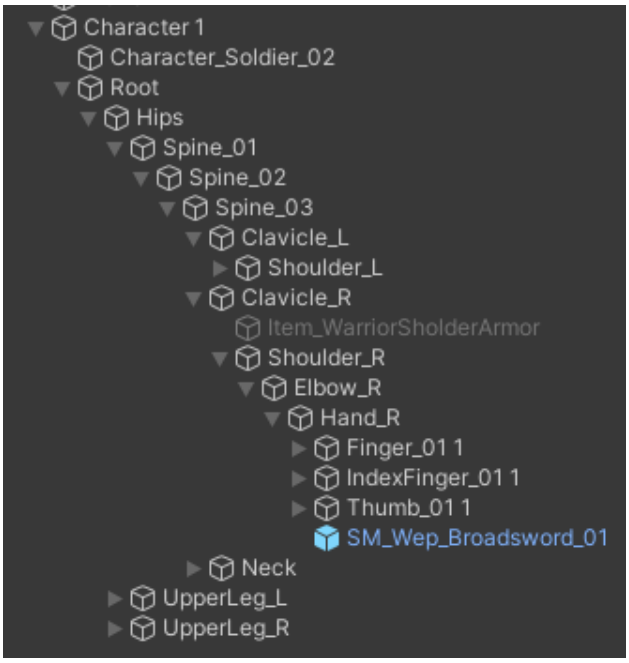


Figure 4. Screenshot of Character 1 hierarchy.

```
//Set from Animation event
0 references
public void SetWeaponHitboxOn()
{
    if (weapon != null)
    {
        weapon.GetComponent<BoxCollider>().enabled = true;
    }
}
//Set from Animation event
0 references
public void SetWeaponHitboxOff()
{
    if (weapon != null)
    {
        weapon.GetComponent<BoxCollider>().enabled = false;
    }
}
```

Figure 5. Weapon hitbox animation events code.

4.1.2 Hit detection with one hurtbox

Using one hurtbox is a generic way to implement hit detection with ease that many tutorials favor. From the setup in Chapter 4.1.1, the attacker's weapon already has a hitbox. Next, a simple hit detection mechanic is implemented that detects hits from triggers. All characters need a Rigidbody, capsule collider, and short script that uses OnTriggerEnter() method. Using the OnTriggerEnter() makes more sense than using OnTriggerStay() or OnTriggerExit() methods, as attack hits should count immediately when a collision occurs between a weapon hitbox and a character's hurtbox. The capsule collider can be

turned into a trigger by checking an IsTrigger checkbox. As only one collider is used, whether the hurtbox is a non-trigger or a trigger will only matter for the character's movement system. The checkbox is left unchecked, which allows the Rigidbody to use the capsule collider for physical collision detection. Rigidbody collision detection has four different modes: discrete, continuous, continuous dynamic, and continuous speculative. For most optimized results, continuous dynamic collision detection mode should be used for fast moving GameObjects and continuous collision detection mode for other scenarios (Collision detection mode 2022). In this thesis, characters use continuous collision detection.

A script with OnTriggerEnter() method can either be placed into characters or weapons. It will not matter for hit detection since characters use only one hurtbox. OnTriggerEnter() provides a reference to other colliders, whether it is a target's hurtbox or a weapon’s hitbox. If the script is placed on the character's GameObject, it would need to fetch the weapon's script in the attacker’s GameObject, which could be difficult and unnecessarily complex to find. The weapon could be anywhere in the attacker’s GameObject hierarchy, so it makes more sense to place the OnTriggerEnter() method script into the weapon. This script will be named Weapon_hit_detection class.

4.1.3 Basic multi-hurtbox hit detection

The problem with using only one hurtbox is that it would not only cover the whole character but also unwanted space between a torso, legs, and arms. In a case where a sword is swung close to the target and despite not visually hitting the target, the sword might go through the hurtbox and result in a hit. This will not make hit detection feel right for the player. The use of multiple hurtboxes allows more accurate hit detection and player satisfaction in terms of combat fighting. They can be added to the character's bones to follow their movement. All colliders can be viewed in Unity’s scene with a physics debugger which allows inspection of the collider geometry (Physics debug visualization 2022). Figure 6 shows the character's hurtboxes in yellow. The character also has a non-trigger collider since a hypothetical movement system could require one.

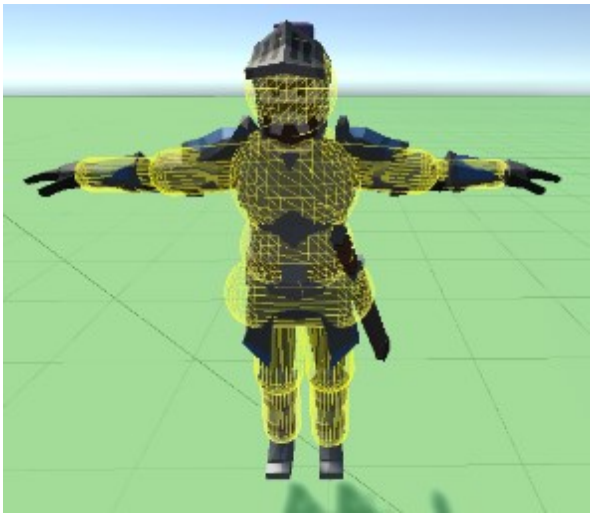


Figure 6. Screenshot of character hurtboxes (yellow) and ground box collider (green) shown in a scene with physics debugger.

At this point, layer collision matrix is updated with new layers. Ground, Character, Hitbox, and Hurtbox layers are added (Figure 7). Hitbox and Hurtbox layers can only collide with each other. Hurtbox layer might be needed to collide with Default and Ground layers if hurtboxes are used with physical collisions. Character layer is reserved for 3D-model GameObjects that have a non-trigger collider in case a movement system is created, and it can collide with Default, Character, or Ground layers. Ground layer is meant for floors, and it has similar layer settings as Character layer. Layer collision matrix already has layers as a default before any new layers are added to it.

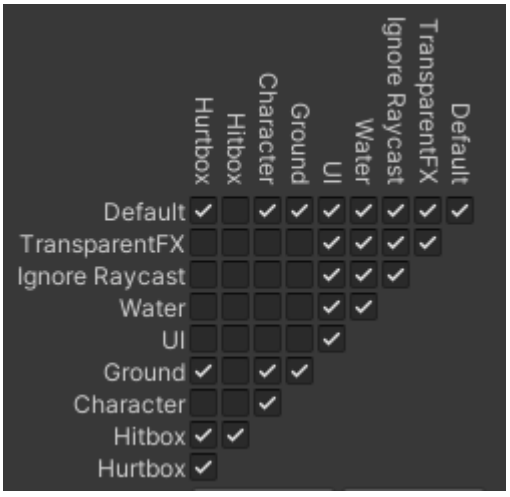


Figure 7. Layer collision matrix.

If multiple hurtboxes are used, multiple hits can occur during a single attack. A new method is required to ignore these duplicate hit messages. This method

needs an identifier from the attacker and information on the time left in the attacker's attack animation. The identifier can either be a GUID or instance ID which is created by Unity for GameObjects at the beginning of the session. Since both of them change when a new session is started, it will not matter which one is used (GetInstanceID 2022). This thesis uses a GUID for character identification. A new class named Attack_info is created for this information (Figure 8). While Attack_info is not imperative as information can be passed as multiple variables to functions, using a class makes it easier to manage this information.

```
public class Attack_info
{
    //guid of the attacker for identifying it if it attacks again
    public System.Guid attacker_guid = System.Guid.NewGuid();
    //time left in attacker's animation
    public float animation_time_left;

    1 reference
    public Attack_info(System.Guid _attacker_guid, float _animation_time_left)
    {
        attacker_guid = _attacker_guid;
        animation_time_left = _animation_time_left;
    }

}
```

Figure 8. Attack_info class code.

Attack_info is passed from the weapon's OnTriggerEnter() method. If OnTriggerEnter() method were on a character, it would need to request information from the attacker and then the attacker would need to send this information back. Therefore, it is easier to place OnTriggerEnter() method on a weapon because less information must be passed between the attacker and the target.

Since characters will have other colliders besides hurtboxes, they must be differentiated from each other. An easy way to do this is with a tag. GameObjects will have a tag named "HurtBox" on any of their child GameObjects connected with hurtboxes. In this setup, characters have hurtboxes in their bones so all bones will receive this tag. Now OnTriggerEnter() method of Weapon_hit_detection class can first verify whether the other collider has a "HurtBox" tag. Then, a verification for null value is performed to ensure that the target has a script in its root GameObject that will receive hits. This script

will be named Character_hit_detection and its purpose is to decide whether a hit from the attacker should be counted or ignored.

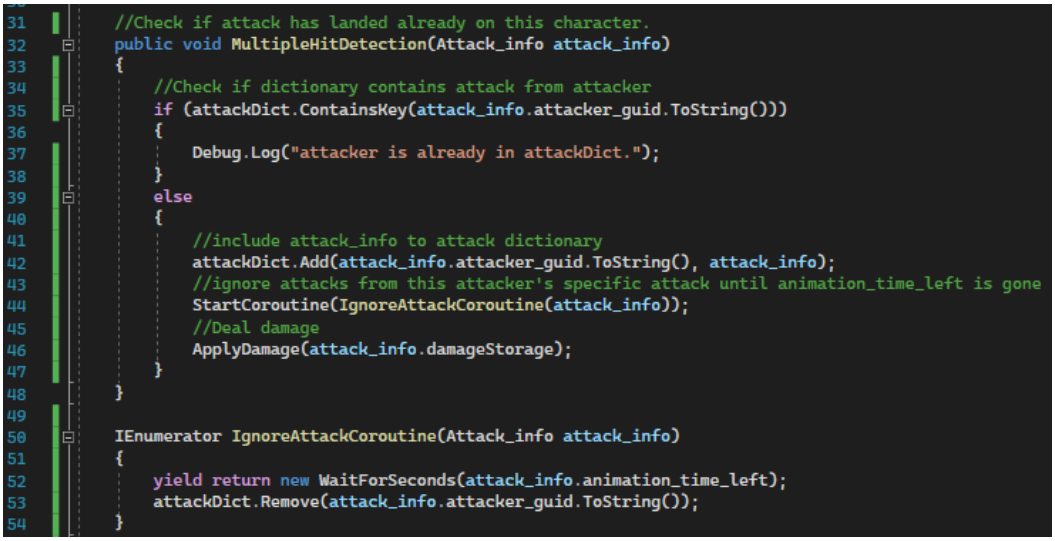
Next, necessary information is fetched. attacker_guid can be acquired from the root GameObject's Character_hit_detection class, and animation_time_left is then calculated. Passed animation time is calculated in Figure 9 in float value named animation_time_passed. The normalized time of AnimatorStateInfo informs how much this animation has progressed on a scale from zero to one with a value of one marking the end of the animation. The value zero in AnimatorStateInfo(0) indicates the animation's layer index number. If only one layer is used, the layer index is zero. If attacks are grouped on a separate layer, then that layer's layer index is used. If attacks are on multiple different layers, a verification is necessary to determine from which layer the attack animation clip is. This layer index verification is not executed in this part of the thesis as only one animation layer is in use but it is implemented in Chapter 4.1.6. By taking a decimal percentage of the passed animation and multiplying it with the length of the clip, the time that has passed in animation is defined. Reducing this amount of time from the clip length allows to see the time that is left in animation in the form of a float variable named animation_time_left. New Attack_info is created with attacker_guid and animation_time_left and sent into the target's MultipleHitDetection() method of Character_hit_detection class as seen at the end of Figure 9. The entirety of the OnTriggerEnter() method is shown in Figure 9.

```
private void OnTriggerEnter(Collider other)
{
    if (other.tag.Contains("HurtBox"))
    {
        //Find other collider's root with Character_hit_detection script and check that it exists
        if (other.transform.root.gameObject.GetComponent<Character_hit_detection>() != null)
        {
            //Attack_info needs this attacker's guid and time left in it's current animation
            System.Guid attacker_guid = this.transform.root.gameObject.GetComponent<Character_hit_detection>().attacker_guid;

            //Calculate passed time in current animation clip.
            currentClipInfo = animator.GetCurrentAnimatorClipInfo(0);
            float animation_time_passed = (animator.GetCurrentAnimatorStateInfo(0).normalizedTime % 1) * currentClipInfo[0].clip.length;
            //Calculate remaining time in current animation
            float animation_time_left = currentClipInfo[0].clip.length - animation_time_passed;

            //Create new Attack_info
            newAttack = new Attack_info(attacker_guid, animation_time_left);
            other.transform.root.gameObject.GetComponent<Character_hit_detection>().MultipleHitDetection(newAttack);
        }
    }
}
```

Figure 9. OnTriggerEnter() method in Weapon_hit_detection class code.



```
//Check if attack has landed already on this character.
public void MultipleHitDetection(Attack_info attack_info)
{
    //Check if dictionary contains attack from attacker
    if (attackDict.ContainsKey(attack_info.attacker_guid.ToString()))
    {
        Debug.Log("attacker is already in attackDict.");
    }
    else
    {
        //include attack_info to attack dictionary
        attackDict.Add(attack_info.attacker_guid.ToString(), attack_info);
        //ignore attacks from this attacker's specific attack until animation_time_left is gone
        StartCoroutine(IgnoreAttackCoroutine(attack_info));
        //Deal damage
        ApplyDamage(attack_info.damageStorage);
    }
}

IEnumerator IgnoreAttackCoroutine(Attack_info attack_info)
{
    yield return new WaitForSeconds(attack_info.animation_time_left);
    attackDict.Remove(attack_info.attacker_guid.ToString());
}
```

Figure 10. MultipleHitDetection() and IgnoreAttackCoroutine() in Character_hit_detection class code.

In the Character_hit_detection class, the received Attack_info will be saved into a dictionary that is called attackDict. A simple verification is performed to see if a new Attack_info is already contained there with the same unique identifier. If this is the case, any new hit is ignored. If Attack_info is not in attackDict, it is added there, and a coroutine is started to ignore duplicate hits. WaitForSeconds(X) method suspends coroutine execution for X seconds using scaled time (WaitForSeconds 2022). Using a coroutine and WaitForSeconds() together, Attack_info can be stored in attackDict as long as necessary. WaitForSeconds() method will use the float value of animation_time_left, which means that attacks from the attacker are ignored for the duration left in the attacker's ongoing attack. The coroutine code is shown in Figure 10.

4.1.4 Advanced multi-hurtbox hit detection

After implementing a working multi-hurtbox hit detection in Chapter 4.1.3, all hits from the attacker are ignored by using only attacker_guid and animation_time_left. If the attacker's attack can be halted, and they have more than one attack mode e.g., fast/heavy attacks or different weapons they can switch, the hit detection described in Chapter 4.1.3 is not sufficient. If the attack animation is halted and a different attack is started, this new attack should not be ignored even if the last attack's animation_time_left is not finished in the hit

detection code. In order to explain it in simpler terms, if a first attack's animation length is 5 seconds and it is interrupted, hit detection should not ignore the next attack even when the 5 seconds have not passed.

An additional piece of information must be passed in attacks to identify the attacker's attack. This information is named attack_id and it is part of Attack_info class. The attack_id does not need to be a GUID, since attacker_guid already differentiates hits between all attackers. Attack_id only needs to identify different animations from the Animator controller, so it can be a string instead e.g. "sword_fast_attack" or "sword_heavy_attack". The problem is, however, that both attacker_guid and attack_id are needed for the identification of an attack, but a dictionary can have only one key per entry, so the attacker_guid will not be a sufficient key anymore. If attacker_guid and attack_id are combined into a single string, this string as a key will be unique and usable for a dictionary (Figure 11). Attack_id is created from the attack animation clip's name (Figure 12).

It should be noted that the animation name that attack_id receives is not the name used in the Animator, but the name used in the assets. This means attack_id would receive the name melee_attack_1 mixamo.com underlined with blue rather than Melee_attack1 underlined with red as seen in Figure 13.

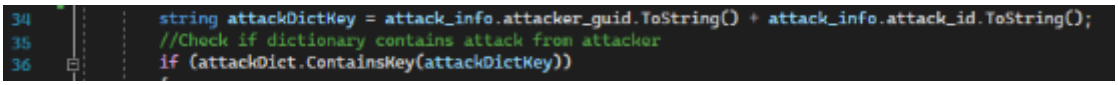


Figure 11. attackDictKey is formed by combining attacker_guid and attack_id code.

```
//give attack_id current animation name
string attack_id = this.animator.GetCurrentAnimatorClipInfo(0)[0].clip.name;
```

Figure 12. Attack_id creation from AnimatorClipInfo name code.

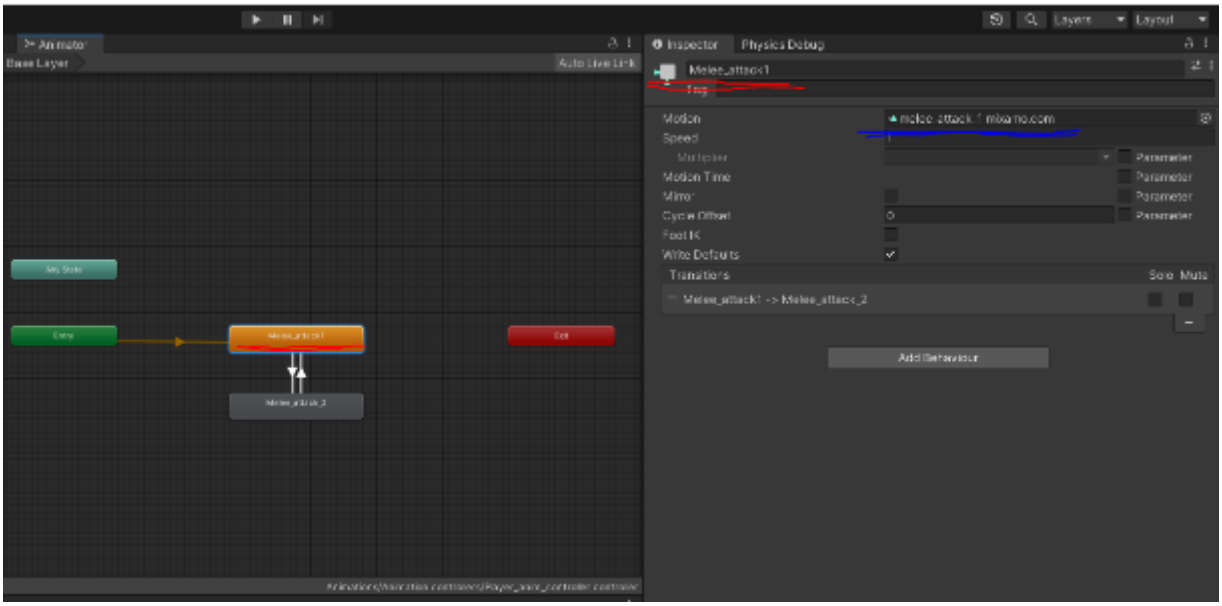


Figure 13. Animator animation clip with underlined names screenshot.

4.1.5 Chain-attacks

Not all attacks are necessarily performed with a single weapon swing. Games might have chain-attacks, where attacks are done one after another. This is suitable for the system described in Chapter 4.1.4, but attacks can also have one long complex animation with multiple moments where the weapon's hit-box activates and deactivates. An example of this could be a repeated thrust attack with a spear. Instead of doing multiple short attack animations in quick succession, one longer animation is done where the spear is repeatedly thrust forward as a special attack. It also prevents unnecessary button smashing for players. In such a case, the target should take more than one hit, which is not achieved with the code produced in Chapter 4.1.4. Cutting the animation into smaller pieces would allow the aforementioned hit detection code to comply with it. This, however, is not desirable as it requires work that is not directly related to coding, and the visual impact of halted animation can be quite disruptive.

A more proper way to proceed would be to tweak SetWeaponHitboxOff() method in Character_hit_detection class. By adding a float parameter to it, the time of the event length can be passed onto it, although this must be done manually in the animation window for every attack. With this new float value, the animation_time_left can be updated to present the correct time until the hitbox is deactivated. The event time is difficult to estimate in seconds, but

frames are easy to observe from the animation window's time bar. In Figure 14, time is calculated in seconds by multiplying given frames with Time.deltaTime, which is the time in seconds between the previous frame and the current one (Unity manual 2022). The seconds are forwarded to a new method named UpdateAnimationTimeLeft() (Figure 15) in Weapon_hit_detection class, which will be used to update a new float variable called animation_event_time_left. Variable animation_time_left is not updated directly in case it is in the middle of receiving a new value. Next, a verification is performed to see if animation_event_time_left is greater than zero, in which case animation_time_left is updated with animation_event_time_left (Figure 16). If not, then a standard calculation will proceed for animation_time_left.

```
//Set from Animation event. Frames are calculated into seconds.
0 references
private void SetWeaponHitboxOn(float frames)
{
    if (weapon != null)
    {
        float seconds = frames * Time.deltaTime;
        weapon.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(seconds);
        weapon.GetComponent<BoxCollider>().enabled = true;
    }
}
//Set from Animation event. No need for timeas weapon hitbox goes off.
0 references
private void SetWeaponHitboxOff()
{
    if (weapon != null)
    {
        weapon.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(0);
        weapon.GetComponent<BoxCollider>().enabled = false;
    }
}
```

Figure 14. Calculating seconds of animation event length and forwarding it to Weapon_hit_detection class code.

```
2 references
public void UpdateAnimationTimeLeft(float time)
{
    animation_event_time_left = time;
}
```

Figure 15. UpdateAnimationTimeLeft() method code.

```
float animation_time_left = 0;
if (animation_event_time_left > 0)
{
    //If Attacker's attack is a chain attack, their animation event passes the time of attack's lenght which is then updated to animation_time_left here.
    animation_time_left = animation_event_time_left;
}
else
{
    //Calculate passed time in current animation clip. Note that layerindex is 0 here! Should fecth current layerindex if more than one is used.
    currentClipInfo = animator.GetCurrentAnimatorClipInfo(0);
    float animation_time_passed = (animator.GetCurrentAnimatorStateInfo(0).normalizedTime % 1) * currentClipInfo[0].clip.length;
    //Calculate remaining time in current animation
    animation_time_left = currentClipInfo[0].clip.length - animation_time_passed;
}
```

Figure 16. New verification with animation_event_time_left code.

4.1.6 Use of multiple animation layers for attacks

Multiple animation layers might be used in a game. Separating movement and attack layer is somewhat standard since in many games characters can walk while performing other animations with their upper body. This will not pose a problem for hit detection if attacks are on the same animation layer. However, characters might have one weapon in each hand. This would make separating right-hand and left-hand attacks on different animation layers a practical solution, which also means the previously achieved hit detection described in Chapter 4.1.5 must be adjusted to manage two attack animation layers.

In order to achieve this, three new animation layers are added. One for right-hand, one for left-hand, and one for dual-wielding animations (Figure 17). By creating separate avatar masks for each layer, bone weight and distribution can be changed to affect only wanted bones (Figure 18). Override setting allows for animations on this layer to replace the animations on previous layers. The hierarchy is upside down on animation layers, as the ones on the bottom are higher on the hierarchical stage.

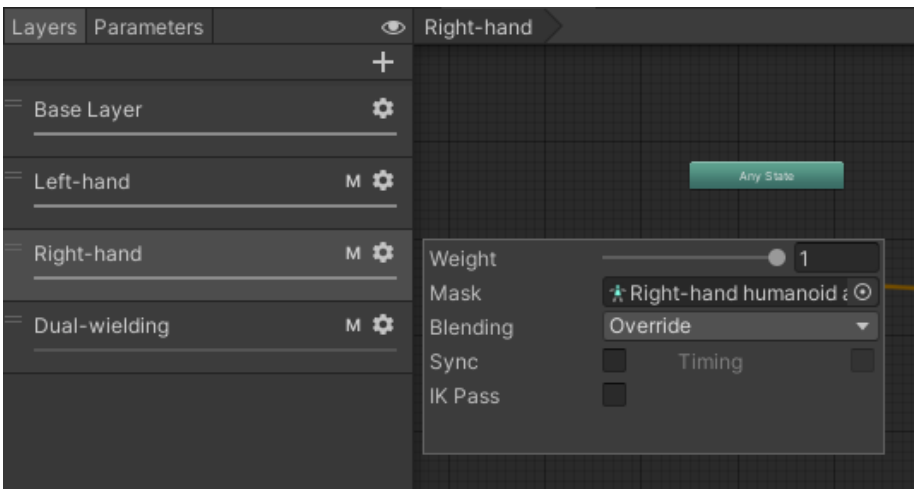


Figure 17. Animation layers.

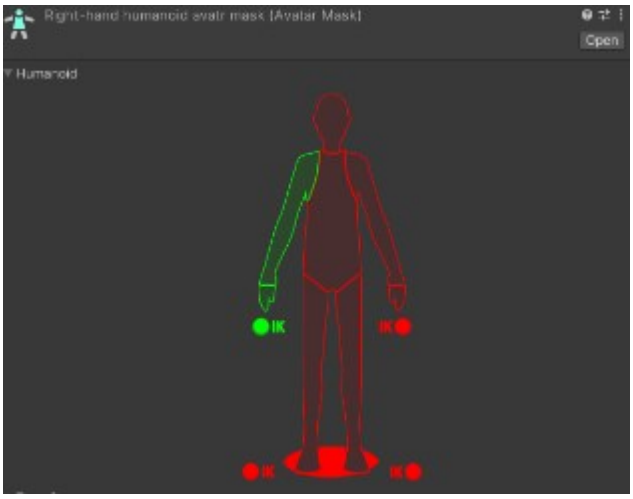


Figure 18. Avatar mask for right hand.

Creating a code to use multiple animation layers for attacks is problematic. The OnTriggerEnter() method does not know which hand's weapon is triggered during a hit. In the Elden Ring game, only one attack is possible to do at a time. If the right-hand attacks, the left cannot, and vice versa. The Elden Ring has dual-wielding attacks where both weapons can be used at the same time for greater damage, but it also means both weapons use the same animation. This would mean while attack animation can be distributed to multiple layers, only one animation layer can be functioning as an attack layer during an attack.

Since no player controller inputs or character inventory system are created in this thesis, a setup for testing multiple animation layers is required. Booleans will be used as a shortcut to determine if the attacker has a weapon in their right hand, left hand, or both. CheckWeaponAttackLayerAnimation() method verifies which animation layer is used by booleans made for this and returns a corresponding layer index integer (Figure 19). The layer index is then used to calculate animation_time_left with no changes. While this thesis utilises no dual-wielding animations, an option for a dual-wielding animation layer is also created. In a dual-wielding attack, both weapon's hitboxes are used, and attack animation times must be the same for both. This must be ensured when creating dual-wield animations. If animation times were different during dual-wielding, the animation state machine would become very complex to manage. When the animation layer is changed depending on the attack, the

weight of the animation layer can be changed with Animator.SetLayer-Weight(), so that correct animations will play. This feature will not be added to this thesis, as creating an animation state machine is not the focus.

```
public int CheckWeaponAttackAnimationLayer()
{
    if (leftHandWeaponAttack)
    {
        return 1;
    }
    else if (rightHandWeaponAttack)
    {
        return 2;
    }
    if (bothHandWeaponAttack)
    {
        return 3;
    }
    return 0;
}
```

Figure 19. CheckWeaponAttackAnimationLayer() method code.

A more serious problem is that animation events can use SetWeaponHit-boxOn() method only on one weapon, as there is only one reference to weapon GameObject in Character_hit_detection class. With no inventory system for characters, straight references for both weapons must be added to Character_hit_detection. SetWeaponHitboxOn() and SetWeaponHitboxOff() methods must be modified to allow both right and left hand weapons to update their animation_time_left for hit detection purposes (Figures 20 and 21). However, as the target will ignore the second weapon's hit after the first weapon hits it, either another identification parameter is required for ignoring hits, or damage should be doubled upon the first dual-wield hit to compensate for only registering one hit.

```
//Set from Animation event. Frames are calculated into seconds.
0 references
private void SetWeaponHitboxOn(float frames)
{
    float seconds = frames * Time.deltaTime;
    if (bothHandWeaponAttack && weaponRightHand != null && weaponLeftHand != null)
    {
        weaponRightHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(seconds);
        weaponRightHand.GetComponent<BoxCollider>().enabled = true;
        weaponLeftHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(seconds);
        weaponLeftHand.GetComponent<BoxCollider>().enabled = true;
    }
    else if (rightHandWeaponAttack && weaponRightHand != null)
    {
        weaponRightHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(seconds);
        weaponRightHand.GetComponent<BoxCollider>().enabled = true;
    }
    else if (leftHandWeaponAttack && weaponLeftHand != null)
    {
        weaponLeftHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(seconds);
        weaponLeftHand.GetComponent<BoxCollider>().enabled = true;
    }
}
```

Figure 20. Modified SetWeaponHitboxOn() that updates right and left hand weapons code.

```
//Set from Animation event. No need for time since weapon hitbox goes off.
0 references
private void SetWeaponHitboxOff()
{
    //implementation with Weapon_hit_detection to work with 2 weapons
        if (bothHandWeaponAttack && weaponRightHand != null && weaponLeftHand != null)
        {
            weaponRightHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(0);
            weaponRightHand.GetComponent<BoxCollider>().enabled = false;
            weaponLeftHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(0);
            weaponLeftHand.GetComponent<BoxCollider>().enabled = false;
        }
        else if (rightHandWeaponAttack && weaponRightHand != null)
        {
            weaponRightHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(0);
            weaponRightHand.GetComponent<BoxCollider>().enabled = false;
        }
        else if (leftHandWeaponAttack && weaponLeftHand != null)
        {
            weaponLeftHand.GetComponent<Weapon_hit_detection>().UpdateAnimationTimeLeft(0);
            weaponLeftHand.GetComponent<BoxCollider>().enabled = false;
        }
}
```

Figure 21. Modified SetWeaponHitboxOff() that updates right and left hand weapons code.

The parameters used to ignore melee hits are attacker_guid and attack_id. Dual-wield attacks share the same attacker_guid and attack_id, but the attack_id could be altered to differentiate between the left and right weapons. If an inventory system would be implemented for an action RPG, all weapons in inventory could have their own id, even duplicates. Since no inventory system is implemented in this thesis, the hit detection system will use weapon GameObject's instance ID, which Unity makes for every GameObject when a scene is started. By using the weapon's instance ID as a part of the attack_id string, both left and right weapons can be differentiated by hit detection code. This also means an animation clip name does not need to be part of attack_id

identification. However, because GameObject instance ID is different between runtimes, it is not suitable for being used in save files, and an inventory system with permanent IDs for weapons would be a preferable solution (GetInstanceID 2022).

4.2 Solution 2

Solution two of hit detection implementation is a direct continuation of multiple animation layer described in Chapter 4.1.7. The objective in solution two is to rework old code and remove dependencies on the attack_id and animation times of attacks.

4.2.1 Reworking the hit detection

The hit detection code developed in Chapter 4.1.3 leans on identifying not just the attacker, but also the attack itself. It requires attacker_guid, attack_id and animation_time_left since the information is passed from the Weapon_hit_detection to the Character_hit_detection. If hit detection was performed only in the Weapon_hit_detection, the necessary information required by hit detection could be reduced. By keeping a dictionary of hit targets, only their GUID must be stored since the attacker knows how long their own attack animation will take to complete. Also, storing animation_time_left might not even be necessary anymore, if a new method is done where the dictionary of hits is wiped clean every time a new attack is started.

When OnTriggerEnter() method detects a hit, instead of sending information to Character_hit_detection class, it will act similarly to MultipleHitDetection() method and proceed to verify hit targets. OnTriggerEnter() will store the target's GUID in a dictionary, which will first verify if the hit has occurred to the target before. Only one variable must be stored for a reference, so a dictionary can be substituted with a list. Since this work has no inputs for players or characters, a setup is made to test this new hit detection system. A boolean variable named startAttack is created and if it is set as true, a new attack will begin, and this will clear the list in OnTriggerEnter() (Figure 22). It should be noted that as damage implementation was processed at the same time, a code line referring to the ApplyDamage() method of the Character_hit_detection can be seen in Figure 22 on the last code line. Also, to preserve old code, all code

from Weapon_hit_detection was copied to a new class named Attack_hit_detection.

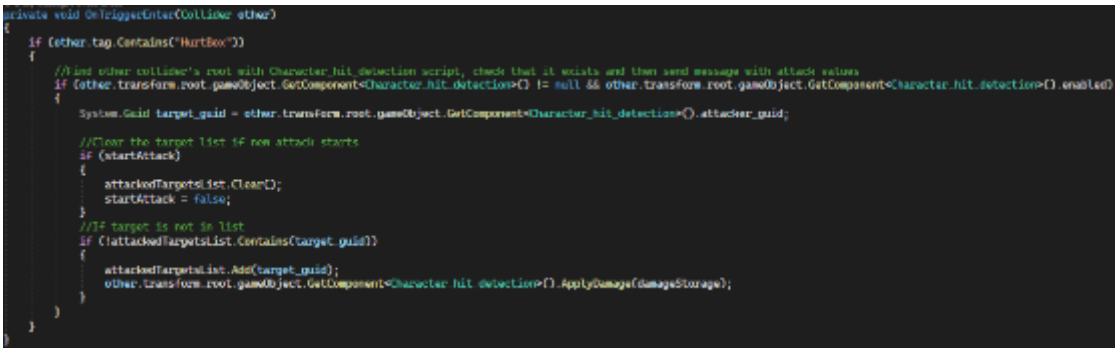


Figure 22. Reworked OnTriggerEnter() method code.

As a result, a hit detection method is completed that is equally good as the hit detection code from Chapters 4.1.3 and 4.1.4. As no attack_id is no more used, this rework fixes the dual-wielding problems related to attack_id from Chapter 4.1.6. Before this rework, only one dictionary was used by Character_hit_detection to store information, but now that lists are used in both left and right weapons, it does not matter that the target's GUID is passed to both lists. The rework does not, however, comply with the chain-attack implementation described in Chapter 4.1.5, where a weapon's hitbox is activated multiple times.

For chain-attacks to function, an input is required to activate new attacks in AttackHitDetection() when the weapon's hitbox activates. In the Character_hit_detection class, the methods that activate and deactivate weapon's hitboxes must be modified to not use UpdateAnimationTimeLeft() method in Attack_hit_detection class, as animation_time_left is not used anymore (Figures 23 & 24). Also, the startAttack bool must be set to true in SetWeaponHitboxOn() method since the work has no controller inputs for testing. With these modifications done, the hit detection functions with chain-attacks, and it is free from being bound to attack_id and animation_time_left. For future use, receiving an input should not automatically start a new attack. For a new attack to start, it should be inspected if there is any ongoing animation that could restrict an attack. For example, giving an attack input should not start an attack if the player character is climbing a ladder.

```
//Set from Animation event.
0 references
private void SetWeaponHitboxOn()
{
    //reworked implementation
    if (bothHandWeaponAttack && weaponRightHand != null && weaponLeftHand != null)
    {
        weaponRightHand.GetComponent<Attack_hit_detection>().startAttack = true;
        weaponRightHand.GetComponent<BoxCollider>().enabled = true;
        weaponLeftHand.GetComponent<Attack_hit_detection>().startAttack = true;
        weaponLeftHand.GetComponent<BoxCollider>().enabled = true;
    }
    else if (rightHandWeaponAttack && weaponRightHand != null)
    {
        weaponRightHand.GetComponent<Attack_hit_detection>().startAttack = true;
        weaponRightHand.GetComponent<BoxCollider>().enabled = true;
    }
    else if (leftHandWeaponAttack && weaponLeftHand != null)
    {
        weaponLeftHand.GetComponent<Attack_hit_detection>().startAttack = true;
        weaponLeftHand.GetComponent<BoxCollider>().enabled = true;
    }
}
```

Figure 23. Reworked SetWeaponHitboxOn() method code.

```
//Set from Animation event.
0 references
private void SetWeaponHitboxOff()
{
    //reworked implementation with Attack_hit_detection
    if (bothHandWeaponAttack && weaponRightHand != null && weaponLeftHand != null)
    {
        weaponRightHand.GetComponent<BoxCollider>().enabled = false;
        weaponLeftHand.GetComponent<BoxCollider>().enabled = false;
    }
    else if (rightHandWeaponAttack && weaponRightHand != null)
    {
        weaponRightHand.GetComponent<BoxCollider>().enabled = false;
    }
    else if (leftHandWeaponAttack && weaponLeftHand != null)
    {
        weaponLeftHand.GetComponent<BoxCollider>().enabled = false;
    }
}
```

Figure 24. Reworked SetWeaponHitboxOff() method code.

After reworking code, CheckWeaponAttackAnimationLayer() and UpdateAnimationTimeLeft() methods have become obsolete in Attack_hit_detection class, and both MultipleHitDetection() and IgnoreAttackCoroutine() methods have become obsolete in Character_hit_detection class. Separating the SetWeaponHitboxOn() and SetWeaponHitboxOff() methods away from Character_hit_detection class could be beneficial, as it would allow GameObjects without weapons to use Character_hit_detection for receiving hits.

4.2.2 Reworking animation events

The code introduced in Chapter 4.1.7 allows dual-wielding attacks, but these attacks must have the same hitbox activation and deactivation times. The

Elden Ring game, however, has dual-wielding attacks where weapon hitboxes can be activated at different times. SetWeaponHitboxOn() and SetWeaponHitboxOff() methods must be modified to allow animation events in the animation window to choose which weapon is activated or deactivated. This can be implemented by either adding a parameter to these methods or by making two separate activation and deactivation methods for each hand (Figure 25). If characters had an inventory with weapon slots, a parameter in weapon hitbox setting methods could reference those slots in the inventory. This would allow characters to use more than two weapons, which would be beneficial if the game uses characters with multiple limbs that can hold weapons.

In a case where weapons require more than one hitbox to form a proper shape of the weapon, all hitboxes will be looped through in weapon hitbox methods, where they will be enabled or disabled when necessary (Figure 25).

```
//Set from Animation event.
0 references
private void SetRightWeaponHitboxOn()
{
    if (weaponRightHand != null)
    {
        weaponRightHand.GetComponent<Attack_hit_detection>().startAttack = true;
        foreach (Collider col in weaponRightHand.GetComponents<Collider>())
        {
            col.enabled = true;
        }
    }
}
0 references
private void SetLeftWeaponHitboxOn()
{
    if (weaponLeftHand != null)
    {
        weaponLeftHand.GetComponent<Attack_hit_detection>().startAttack = true;
        foreach (Collider col in weaponLeftHand.GetComponents<Collider>())
        {
            col.enabled = false;
        }
    }
}
0 references
private void SetRightWeaponHitboxOff()
{
    if (weaponRightHand != null)
    {
        foreach (Collider col in weaponRightHand.GetComponents<Collider>())
        {
            col.enabled = false;
        }
    }
}
0 references
private void SetLeftWeaponHitboxOff()
{
    if (weaponLeftHand != null)
    {
        foreach (Collider col in weaponLeftHand.GetComponents<Collider>())
        {
            col.enabled = false;
        }
    }
}
```

Figure 25. Hitbox activation and deactivation methods for two different weapon hands.

4.2.3 Alternative ways to ignore hits

Previous hit detection implementations have been using a dictionary or a list to keep a tab of targets/attackers that should ignore further hits after the first hit lands from the attacker. These lists can be substituted with Physics.IgnoreCollision() method. This method makes collision detection ignore all collisions between two colliders (IgnoreCollision 2022). If IgnoreCollision() is used to loop through all hurtboxes in the target, no more collisions will occur. This can be reversed by looping through them again and setting the ignore parameter to false. However, to remember which hurtboxes are set to be ignored, a list must be again used to store references to those colliders, so that ignored collisions can be restored after the attack is finished. This solution would increase work which is against its purpose of reducing the usage of lists.

4.2.4 Ragdoll death mechanic

In the Dark Souls games, when characters die, their bodies go limb. In video games, this is called being a ragdoll. Rigidbodies create realistic motion through four different properties: mass, gravity, velocity, and friction (Goldstone 2009, 41). Unity has its own ragdoll wizard that makes creating a Rigidbody system fast. Ragdoll wizard can be found in a menu bar by choosing **GameObject > 3D Object > Ragdoll**. This wizard automatically adds a Rigidbody, Colliders, and Character joints to chosen character bones. Ragdoll state in characters can be activated by setting Rigidbody.IsKinematic off and preferably Rigidbody.UseGravity on for more realistic results. Colliders must be non-triggers so that they can be affected by gravity and physical collisions. Animator component must be disabled before the ragdoll state is started. All required steps to activate the ragdoll state are gathered in a new method named StartRagdoll() (Figure 26). In the same method, the hit detection must be disabled when the ragdoll state activates. During testing, the ragdoll body disappeared after few seconds if "Update when offscreen" -setting was not set as true in a character 3D-model's Skinned Mesh Renderer component. Using this setting requires more processing power, so it needs to be weighted if it is truly necessary for the game.

```
1 reference
private void StartRagdoll()
{
    animator.enabled = false;
    if (gameObject.GetComponent<Collider>() != null)
    {
        gameObject.GetComponent<Collider>().enabled = false;
    }
    foreach (Collider col in gameObject.GetComponentsInChildren<Collider>())
    {
        col.isTrigger = false;
        col.transform.gameObject.GetComponent<Rigidbody>().isKinematic = false;
        col.transform.gameObject.GetComponent<Rigidbody>().useGravity = true;
    }
    gameObject.GetComponent<Rigidbody>().isKinematic = false;
    gameObject.GetComponent<Rigidbody>().useGravity = true;

    this.enabled = false;
}
```

Figure 26. StartRagdoll() method code.

5 DAMAGE-RELATED MECHANICS IMPLEMENTATION

This chapter relates both to solutions one and two, and damage mechanics can be implemented with both solutions. The only difference is whether damage values are passed inside Attack_info class or sent directly into damage methods. Any changes to Attack_hit_detection class from solution two in this chapter can be made to Weapon_hit_detection class from solution one.

5.1 Damage implementation

Hitting characters is useless if they cannot be damaged. Solution one used Attack_info class to pass information on the attacker's current attack, and it can also be used to pass different damage values. An int or float variable can pass a damage value in method calls, which solution two uses, or they can be added to Attack_info class if solution one is used. RPGs can have multiple different damage types e.g. weapon damage, fire damage, poison damage, and magic damage. The possibilities for damage types are endless. An array can be used to store multiple values of the same variable type. Using a float array which will be named damageStorage, multiple float values can be passed in one parameter. The float array can be of any size if its size is first specified. In this thesis the size is not a factor, so a size three array is created. Since damage types in this thesis could be freely named, for simplicity these variables are named damageType1, damageType2, and damageType3 so that they can

represent any type of damage (Figure 27). Their damage values will not matter either. Adding damage types into the damageStorage array (Figure 28) is only required once so it is done inside the Awake() method.

```
public float damageType1 = 10;
public float damageType2 = 5;
public float damageType3 = 5;
```

Figure 27. Damage type floats are created, and values set code.

```
//save damage values to float[] array
damageStorage[0] = damageType1;
damageStorage[1] = damageType2;
damageStorage[2] = damageType3;
```

Figure 28. Adding damage types into damageStorage code.

Health points of a character will be represented by a float variable named health. RPG games have resistances for each damage type, so three corresponding damage resistance types are added: damageType1Resistance, damageType2Resistance, and damageType3Resistance (Figure 29). ApplyDamage() method is called in MultipleHitDetection() method of Character_hit_detection class when a hit should damage the target. The question of who the attack should damage is answered in Chapter 5.2. Final damage values are calculated in ApplyDamage() by decreasing the damage type resistance percent of each damage type. Next, a verification is performed to see if the damage value is negative. If the damage value is negative, it is set as zero because these damage values are subtracted from health and if they are negative then health would increase. The target should die if health reaches zero, but this feature is not implemented in this thesis. The full ApplyDamage() method is shown in Figure 30. It also includes an animator.SetTrigger() code to make characters do a faltering animation when they receive damage.

```
public float damageType1Resistance = 0.2f;
public float damageType2Resistance = 0.5f;
public float damageType3Resistance = 0.5f;
```

Figure 29. Damage type resistance floats are created, and values set code.

```
public void ApplyDamage(float[] damageStorage)
{
    //calculate damage when resistances are applied
    float damageType1 = damageStorage[0] - damageStorage[0] * damageType1Resistance;
    float damageType2 = damageStorage[1] - damageStorage[1] * damageType2Resistance;
    float damageType3 = damageStorage[2] - damageStorage[2] * damageType3Resistance;

    //no negative damage values allowed
    if (damageType1 < 0) { damageType1 = 0; }
    if (damageType2 < 0) { damageType2 = 0; }
    if (damageType3 < 0) { damageType3 = 0; }

    //check needed because player is not using this animator controller right now
    if (controller.name == "Character_anim_controller")
    {
        animator.SetTrigger("Taking damage");
    }

    //reduce damage from character health
    health = health - damageType1 - damageType2 - damageType3;

    if (health <= 0)
    {
        //character is dead, activate ragdoll
        bool startRagdollOnce = true;
        if (startRagdollOnce)
        {
            StartRagdoll();
            startRagdollOnce = false;
        }
    }
}
```

Figure 30. ApplyDamage() method in Character_hit_detection class code.

In action RPGs, weapon damage can be affected by weapon upgrades and character level. Two more variables would be needed to scale weapon damage with weapon upgrade level and character level. Adding a method that updates damage values with these variables can be added to any weapon script. Different games have different scaling for weapon damage, so there does not seem to be a universal scaling curve.

5.2 Differentiate targets

In RPGs, there are enemies that players can hit, but there are also friendly characters or destroyable objects. Attack_hit_detection class inflicts damage to everything with hurtboxes. Being able to hurt friendly characters might not be desirable for all games so the hit detection system needs a way to choose which GameObjects with hurtboxes are allowed to be damaged. First, differences must be made clear between all GameObjects with a HurtBox tag. This work will use three tags for hittable GameObjects: PlayerFriendly, Enemy, and

DestroyableObject. These tags will be used on parent GameObjects since verificating the parent GameObject's tag is sufficient.

A simple tag verification system can be implemented with Enum (short for enumeration). It allows the representation of named constants with numbers. This way, the tag of the target can be compared to the Enum list, where the names of tags that the attacker wants to hit are preselected. However, if joining tags together is desired, a Flags system is needed. Flags allow using Enum values that might appear in a combination. For example, if Enum values are compared without the Flags attribute, only the first matching value is returned, but if Flags is used, all possible matching combinations can be returned. This would allow mixing tags, which could be useful depending on the game's needs. If multiple GameObjects with different hittable tags are present or the target's damage sources could change during runtime, mixing tags could be helpful for future mechanics. This thesis will only show how to implement Flags tag system it but will not expand it.

The tag verification system will be built in Attack_hit_detection class. A new Enum type named Targets is created with same names that will be used as tags on hittable GameObjects (Figure 31). Nothing and Everything options are automatically added to Enum by Unity. Integer values of Enum names must be powers of two so that combined enumeration names do not overlap (Enum Class 2022). New Targets enum is added and named attacker_targets, which can be selected in the inspector window in Figure 32. Next, the target's tag must be compared with the attacker_targets Enum. A new method named CheckTargetTag() is created for this in Figure 33, which can be then called during OnTriggerEnter() method whenever a hurtbox is hit. Targets Enum is expandable, as long as new tag comparisons are added to CheckTargetTag().

```
[System.Flags]
7 references
public enum Targets
{
    //!!! numbers must be powers of 2 !!!
    PlayerFriendly = 1,
    Enemy = 2,
    DestroyableObject = 4
}
public Targets attacker_targets;
```

Figure 31. Enum Targets code.

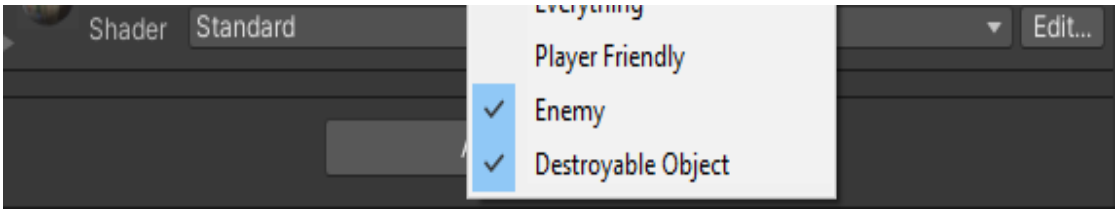


Figure 32. Enum Targets on inspector window screenshot.

```
private bool CheckTargetTag(string target_tag)
{
    //First is checked if my_targets has target and then if target_tag is in attacker_targets
    if (attacker_targets.HasFlag(Targets.PlayerFriendly))
    {
        if (target_tag.Contains(Targets.PlayerFriendly.ToString())) { return true; }
    }
    if (attacker_targets.HasFlag(Targets.Enemy))
    {
        if (target_tag.Contains(Targets.Enemy.ToString())) { return true; }
    }
    if (attacker_targets.HasFlag(Targets.DestroyableObject))
    {
        if (target_tag.Contains(Targets.DestroyableObject.ToString())) { return true; }
    }
    return false;
}
```

Figure 33. CheckTargetTag() method code.

5.3 Weak spots

In games, enemies often have spots where they take increased damage, such as the head. These are called weak spots or sometimes called weak points, and while they might not be as important in humanoid enemies, they could be more prominent in giant monsters. A weak spot in a dragon could be, for example, a head, a tail, or wings.

Characters can have multiple hurtboxes, and anyone of those could be tagged as weak spots. Since GameObjects can only have one tag, this could be implemented by adding a new tag named “HurtBox WeakSpot”. The hit detection code compares if a tag is precisely called “HurtBox” with CompareTag(). It is better to instead verify that the word weak spot is contained within the tag with Contains() method. This means the tag in hurtboxes does not need to be identical with the search word, as long as a weak spot word is contained within it. Later, changes will be easier to implement and expand when words can be added to tags. Verification for a weak spot tag is added to the OnTriggerEnter() method in Attack_hit_detection class, where an extra ten percent of damage is added to the damageStorage array (Figure 34).

```
if (other.tag.Contains("WeakSpot"))
{
    float extraDamage = 1.1f;
    damageStorage[0] = damageStorage[0] * extraDamage;
    damageStorage[1] = damageStorage[1] * extraDamage;
    damageStorage[2] = damageStorage[2] * extraDamage;
}
```

Figure 34. Weak spot code.

Since damage values are altered in the case of hitting a weak spot, to preserve original damage values, extra damage is applied straight into damageStorage values and not into damageType values. Values in damageStorage must be returned to their original values after they are sent to Character_hit_detection. This is simply done by setting each damageStorage value as equal to the corresponding damageType value (Figure 35).

```
//return original damage values, since hitting a weak spot alters them
damageStorage[0] = damageType1;
damageStorage[1] = damageType2;
damageStorage[2] = damageType3;
```

Figure 35. DamageStorage values are reset.

5.4 Projectiles

All objects that may harm targets might not have a Character_hit_detection class in them. Magic projectiles, crossbow bolts, throwable bombs/daggers, exploding barrels, or fire in the ground should inflict damage to targets, but they act independently of characters, meaning once they are created, they are not part of the attacker's hierarchy. Assigning the Character_hit_detection class to them would not make sense since they do not take damage. The exploding barrel is an exception as it should take damage, but it should have a different class as it is not a character. Projectiles and throwable bombs/daggers should be instantiated and then removed by destroying them after they hit something, be it a target or an environmental GameObject. Instantiation means a method of creating (also called spawning) GameObjects from a prefab template during runtime and it can be used to duplicate existing GameObjects in the scene (Goldstone 2009, 180).

Projectile removal could also be further optimized by making a pre-initialized pool of projectiles and then initializing them when they are needed. Then, instead of destroying a projectile, it is deactivated and returned to the pool. This approach is more useful for prefabs that are instantiated often, such as bullets in first-person shooter games. (Macek n.d.)

A new hit detection script is made for projectiles and other attacks that only hit once. This script is named Projectile_hit_detection, and its code is copied directly from Attack_hit_detection, which was produced in solution two. Only character tag verification and damage-related code are required. Damage will be applied to ApplyDamage() method in Character_hit_detection if targets are hit. Whether the projectile hits a character or something else, it is then destroyed with Unity's Destroy() method. In order to give the projectiles an actual flight arch, speed and direction variables should be added, and a new method would be required to calculate the projectiles' flight paths. There are multiple ways to achieve this. A mathematical calculation can be created for this, or a Rigidbody can be used with gravity and forces to make the projectile fly in an arch. A movement system will be omitted from this thesis, as only hit detection is in the focus.

When a projectile is instantiated, it uses a prefab of a projectile. The prefabs are instantiated by the attacker, which will provide the necessary parameters of hittable targets. Projectile prefabs must have a trigger collider attached to them so that OnTriggerEnter() method can be used. A speed variable should be universal for the same type of projectiles, but damage should be affected by either the attacker's level or the weapon's level. That is why damage variables are public so that they can be possibly updated later.

5.5 Invincibility

Dodging is an important mechanic in the Dark Souls games. A dodge provides a certain amount of invincibility frames, also known as i-frames, during which damage is completely ignored from all sources. Invincibility can either be implemented by deactivating all hurtboxes or using a boolean variable to verify if damage should be ignored. Using a boolean variable is more helpful, as it re-

quires less processing power than switching all hurtboxes. Whether a character is dodging or a special animation sequence requires an invincibility state, invincibility verification can be added to ApplyDamage() method in Character_hit_detection class, as some damage sources might directly send damage into it. If deactivating hurtboxes is more favored, it can be implemented by either deactivating all colliders with a HurtBox tag by looping through all child GameObjects or using a stored reference of all child hurtboxes.

6 RESULTS

The first objective of this thesis was to produce a functioning combat hit detection system in Unity game engine for 3D action role-playing games that utilises multiple colliders in characters. The second objective was to implement damage which is related to hit detection. All objectives were achieved. Results chapter of thesis shows conclusions and further development plans.

6.1 Conclusions from implementation

After considering and trying different possibilities, an accurate hit detection suitable for 3D action RPGs was achieved in Unity. Two solutions were created with scripts that took advantage of Unity’s physics by using Rigidbodies and multiple Colliders. Animation events and tags were also used as a part of core functionality.

The first solution focused on passing information used to identify an attack hit from the weapon script to the character script. The hit detection was done on both scripts, which proved to be unnecessarily complex. It created many dependencies that limited ways of creating fighting mechanics. The hit detection code was reworked in solution two, and a better way was implemented by moving all hit detection functionality into the weapon's code to reduce the complexity of exchanging information between script classes. This reduced the total amount of code and made a few methods in code obsolete. The overall complexity of code varies depending on how data is passed between the scripts and what the needs of the game are, such as how many characters are fighting, how many attack animations characters have, or how many weapons a character can use.

Other solutions were explored where all hit detection code was moved from weapon code to character code. This required a script for each character's child GameObject and while it worked, it brought no improvements. Replacing all identifications required in hit detection with one GUID was also tried. This did not improve code, but it allowed to exercise with a different coding strategy.

OverlapBox is another physics component in Unity that can detect collisions. It was not used in the implementation part of this thesis but it is very similar to triggers. It computes and stores all colliders that are touching the box or are inside it, as was the case in implemented solutions with triggers and lists. It allocates an array of colliding colliders on every frame, which generates garbage. OverlapBoxNonAlloc fixes this problem by storing collision data into a provided buffer. It does not attempt to grow the buffer if it runs out of space, so the buffer array should be made sufficient size to store a hypothetical maximum number of collisions in an attack. OverlapBox could have been a good alternative for what was eventually implemented. Whether a Unity game uses a trigger collider or an OverlapBox component, a system for multi-hit detection is achievable.

Mechanics related to damage that rely on hit detection were implemented as well. This required less effort compared to hit detection, as only damage values had to be passed forward to other scripts. Weapon attacks can apply different damage values during attacks to targeted characters. Targets are selected in the character’s code, which means friendly characters can be opted out of being hit during attacks. No movement was implemented, so projectiles do not fly forward, but their hit detection is a more simplified version of the weapon hit detection code.

6.2 Further development

Alone, the aforementioned codes do not create a coherent game, but they provide valuable solutions on how to create a hit detection system. Implementing more mechanics could produce a working 3D action role-playing game. Below is a list of suggestions that could be further developed concerning hit detection:

- Strategies for hit detection implementation
- Hit detection implementation for non-trigger collisions, such as a collision using the OverlapBoxNonAlloc component
- Increased attack mechanics

If the content of this thesis was evolved into a real game or imported to another game project, the following factors should be examined in order to create a complete game:

- Movement and camera system
- Inventory system
- Animation state machine
- Input controller for movement and attacks
- UI for character health bars
- Character and weapon levels

The hit detection scripts of both solutions produced in this thesis will be uploaded to GitHub. The whole content cannot be uploaded there because paid third-party assets were used as 3D-models. This minor setback, however, is quite insignificant as the focus was on producing scripts for hit detection.

The produced codes are stored in a GitHub repository, and a link to this repository is presented in Appendix 1.

7 SUMMARY

The first and most important objective of this thesis was to explore ways to implement a hit detection system that could be used for combat in action role-playing games in Unity. The second objective was to implement mechanics that were tied to hit detection. Damage, weak spots and projectiles implementation were explored after hit detection operated with multiple colliders.

This thesis was faced with many challenges. It was surprisingly difficult to find instructions on creating a hit detection system that utilises multiple colliders. This was also one of the reasons why this thesis was considered necessary and beneficial. Research material was difficult to directly utilise, as most of it only focused on the surface of Unity physics. The source code of PhysX and mathematical equations of collisions, on the other hand, were omitted from this thesis as it was deemed they would bring no additional information. The

definition of methods to implement hit detection was important, and while two solutions were created, more options would have allowed more versatility.

The results for the hit detection code were quite specific, as the thesis aspired to resemble the Dark Souls franchise style games. The first implementation worked so well that it was developed further to include more combat mechanics, such as chain-attacks and multiple weapons. This, however, posed later problems, as the complexity of the code increased as more mechanics were included. The second solution reworked the first one and made it simpler and more expandable, allowing more flexibility for later updates.

The code in this thesis was built from scratch so there was no inventory system to help manage weapons or input controller to help test character attacks. Public variables were used without restrictions to help with script testing, as they could be switched smoothly from the inspector window during Editor runtime. If the scripts were imported to another project for use, most of the public variables should be made private.

Designing and testing methods to find a functioning solution was a mix of trial and error. The experience was valuable, and will most likely be very useful for future game projects.

REFERENCES

Animation State Machine. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/AnimationStateMachines.html [Accessed 15 May 2022].

BEST PRACTICES FOR RIGID BODIES IN UNITY. 2014. Digital Opus. WWW document. Available at: https://digitalopus.ca/site/using-rigid-bodies-in-unity-everything-that-is-not-in-the-manual/ [Accessed 13 May 2022].

Collision detection mode. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Rigidbody-collisionDetectionMode.html [Accessed 12 May 2022].

Dickinson, C. 2017. Unity 2017 Game Optimization. 2nd ed. Birmingham: Packt Publishing.

Elden Ring. 2022. Metacritic. WWW document. Available at: https://www.metacritic.com/game/playstation-5/elden-ring [Accessed 3 May 2022].

Enum Class. 2022. Microsoft docs. WWW document. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.enum?view=net-6.0 [Accessed 20 May 2022].

Ericson, C. 2005. Real-time collision detection. San Francisco: Morgan Kaufmann Publishers.

GameObjects. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/GameObjects.html [Accessed 10 May 2022].

GetInstanceID. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Object.GetInstanceID.html [Accessed 28 May 2022].

Goldstone, W. 2009. Unity Game Development Essentials: Build Fully Functional, Professional 3D Games with Realistic Environments, Sound, Dynamic Effects, and More! Birmingham: Packt publishing.

Gonçalves, A. A. R. L. 2015. Efficient Contact Detection for Game Engines and Robotics.

Guid struct. 2022. Microsoft docs. WWW document. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.guid?view=net-6.0 [Accessed 20 April 2022].

Hierarchy. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/Hierarchy.html [Accessed 10 May 2022].

IgnoreCollision. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Physics.IgnoreCollision.html [Accessed 24 May 2022].

Introduction to collision. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/CollidersOverview.html [Accessed 25 April 2022].

Kasavin, G. 2001. Blade of Darkness Review. Gamespot.com. Article. Available at: https://www.gamespot.com/reviews/blade-of-darkness-review/1900-2690848/ [Accessed 5 July 2022].

Krogh-Jacobsen, T. 2021. Optimize your mobile game performance: Get expert tips on physics, UI, and audio settings. Blog.unity.com. Blog. Available at: https://blog.unity.com/technology/optimize-your-mobile-game-performance-get-expert-tips-on-physics-ui-and-audio-settings [Accessed 27 April 2022].

Macek, T. n.d. The 10 Most Common Mistakes That Unity Developers Make. Available at: https://www.toptal.com/unity-unity3d/top-unity-development-mistakes [Accessed 22 April 2022].

Mesh collider. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/class-MeshCollider.html [Accessed 12 May 2022].

Millington, I. 2007. Game Physics Engine Development. San Francisco: Morgan Kaufmann Publishers.

Order of execution for event functions. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/ExecutionOrder.html [Accessed 10 May 2022].

Physics debug visualization. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/PhysicsDebugVisualization.html [Accessed 15 May 2022].

Ranney, N. 2017. GameMaker Basics: Hitboxes and Hurtboxes. Blog. Available at: https://developer.amazon.com/blogs/appstore/post/cc08d63b-2b7c-4dee-abb4-272b834d7c3a/gamemaker-basics-hitboxes-and-hurtboxes [Accessed 15 May 2022].

Rigidbody. 2022. Unity documentation. Available at: https://docs.unity3d.com/Manual/class-Rigidbody.html [Accessed 14 April 2022].

Rodrigues, J. 2018. A Complete Guide to Fixing Collision Detection in Unity. Bladecast. E-magazine article. Available at: https://bladecast.pro/unity-tutorial/fix-my-collision-complete-guide-collision-trigger-detection-unity#trigger [Accessed 12 May 2022].

Time.deltaTime. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Time-deltaTime.html [Accessed 30 April 2022].

Trigger colliders. n.d. Learning unity3d eBook. WWW document. Available at: https://riptutorial.com/unity3d/example/19743/trigger-colliders [Accessed 30 April 2022].

Unity performance tests. 2015. Blog. Available at: https://snowhydra.wordpress.com/2015/06/01/unity-performance-testing-getcomponent-fields-tags/ [Accessed 10 May 2022].

Unity Physics. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/Manual/PhysicsSection.html [Accessed 10 May 2022].

Unity: CHARACTER CONTROLLER vs RIGIDBODY. 2017. IronEqual. WWW document. Available at: https://medium.com/ironequal/unity-character-controller-vs-rigidbody-a1e243591483 [Accessed 22 April 2022].

WaitForSeconds. 2022. Unity documentation. WWW document. Available at: https://docs.unity3d.com/ScriptReference/WaitForSeconds.html [Accessed 20 April 2022].

LIST OF FIGURES

Link to GitHub repository

https://github.com/VilleTamminen/Action-role-playing-game-hit-detection