Building a Time Machine in Unity3D

Lukas Kallenbach

Bachelor's thesis
December 2014
Degree Programme in Media

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Media

KALLENBACH, LUKAS:
Building a Time Machine in Unity3D

Bachelor's thesis 22 pages, appendices 2 pages
December 2014

The objective of the thesis was to implement a framework for the game engine Unity3D enabling time travelling in video games, with the purpose of the time travelling framework being to enable freely jumping to any point in time within a certain time line.

A concept of implementing time travelling in a video game is presented. Assuming that a deterministic simulation can be fast-forwarded and thus allows travelling forward in time, travelling backwards in time is possible by resetting the simulation to its beginning and from there forwarding it until the desired point. Restrictions that come with this approach and limitations on the length of the timespans that can be travelled are researched and explained. Additionally, the presented concept is put into context with notable video games that use time travelling as a game mechanic.

The time travelling concept has been successfully implemented and important development restrictions for ensuring a deterministic simulation have been documented. Experiments have shown that a game created with the time travelling framework can be forwarded at a speed of typically 5-15 in-game minutes in ten seconds. Researching games that include time travelling has shown that the option of freely jumping to any point on a time line had not yet been implemented in video games in a notable way.

The time travelling framework created for this thesis is a base for creating games that play with cause and effect by letting the player explore long-term consequences of their actions through time travel. While simulation and puzzle games are predestined genres for implementing time travel as suggested in this thesis, other genres and experimental games are possible, interesting options.

Key words: Unity3D, game development, game design, video games, time travelling.

Contents

1 INTRODUCTION 4
2 Building the Time Machine 5
2.1 Concept 5
2.2 How to make fast forwarding possible 6
2.2.1 Using Time.timeScale 6
2.2.2 Consistency 7
2.2.3 Using FixedUpdate instead of Update 7
2.2.4 Dealing with Player Input 11
2.3 Limitations 14
2.3.1 Fixed starting point 14
2.3.2 Waiting times when forwarding 15
3 Time Travelling as a Game Mechanic 17
3.1 Braid 17
3.2 The Legend of Zelda: Majora's Mask 18
3.3 Achron 19
3.4 Comparing the games to the Unity time machine 20
4 Discussion 23
4.1 Results 23
4.1.1 Implementing the time machine 23
4.1.2 Time travelling as a game mechanic 24
4.2 Development suggestions 24
5 References 25

1 INTRODUCTION

As being able to freely travel in time is still one of humanity's unfulfilled dreams, many videos games play with the idea of time travelling by implementing it in some way.
This thesis documents and researches the creation of a time travelling framework for the game engine Unity3D called the Unity time machine. The goal of the Unity time machine is to enable freely jumping to any point on the time line of a simulation game, so that a player can travel between distant points in time to see long-term consequences of their actions. Additionally, the Unity time machine is put into context with games that notably employ time travelling as a game mechanic.
The Unity time machine and a playable demo using it are implemented as the practical part of the thesis. Part 2 of the thesis focuses on the technical aspect of creating and working with the time machine. The concept used to enable time travelling is presented and evaluated, with the goal of finding and solving problems that occur when time travel is implemented in a simulation game, and researching the limitations of time travelling speed and range. This is done by exploring possible features when creating the playable demo and measuring travel times.
Part 3 puts the Unity time machine into the context of other video games with a focus of time travelling as a game mechanic. The way time travelling game mechanics are used in a selection of relevant games is researched and compared to features available in the Unity time machine.

2 Building the Time Machine

2.1 Concept

The approach to implement time travelling for simulation games in the Unity game engine works as follows:

Firstly, fast forwarding the simulation is made possible, which enables travelling forward in time. Figure 1 shows the trivial case of travelling forward in time from A to B:

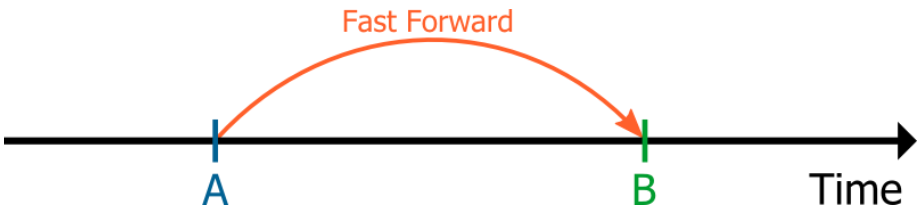

Figure 1: Travelling Forward

Given that fast forwarding the simulation is possible, travelling backwards is done by reloading the level, or scene in Unity terms, and fast forwarding from there until the desired point. Figure 2 illustrates how travelling from A to B is implemented if B is a point in time before A:

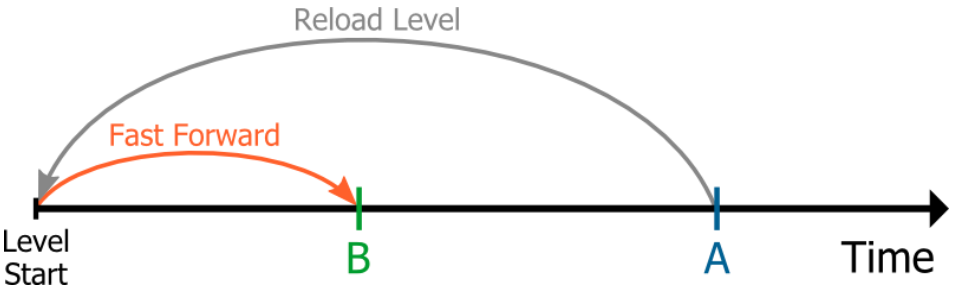

Figure 2: Travelling Backwards

Player actions that influence the simulation are recorded before they are applied to the simulation, so that they can be repeated automatically when a situation that has been influenced by player actions before is played again.

The suggested approach enables travelling to virtually any point on time after a fixed starting point, which is the level start.

An advantage of this compared to the approach of saving all actions in the game and rewinding by reversing them is that it takes much less memory space: Instead of storing

everything that has happened in order to move between present and past game states, any game state can be reproduced by simulating the events that ultimately lead to it. Only user input that changed events on the time line has to be stored and included in the simulation.

2.2 How to make fast forwarding possible

2.2.1 Using Time.timeScale

Forwarding means making in-game time passing faster. The first and obvious Unity functionality to be used for this is Time.timeScale, as it allows changing the speed at which the time passes in the game.

Time.timeScale has the default value of 1, which means that time is passing at normal speed (Unity Documentation: Time.timeScale 2014). The maximum value allowed by Unity is 100, implying that the maximum speed at which a simulation can be forwarded is 100 times faster than real time.

Update and FixedUpdate

While a video game is running, a game loop is used to create the illusion of real-time interaction. Part of the game loop is the update stage, in which game logic, animations and AI actions, practically all ongoing processes within the game world, are updated and applied to the game state. A game's performance is commonly measured by counting how many times per second this happens. (Valente, Conci, & Feijô, 2005)

In Unity, there are two different types of update stages which are relevant for this paper: FixedUpdate and Update. Update is commonly used for most things that happen over time, such as moving non-physics objects, counting down timers and reading player input. FixedUpdate is intended to be used for updating objects in the game that use the physics engine. (Unity Tutorials: Update and FixedUpdate 2014.)

The execution of the Update function is referred to as one Update frame, the execution of the FixedUpdate function is called a physics frame or FixedUpdate frame respectively.

2.2.2 Consistency

Definition Consistency:

A simulation is called consistent if it, when run multiple times, each time results in exactly the same state after the same amount of passed in-game time. This implies the simulation being deterministic.

If a game or simulation is fast forwarded, the time that has passed in-game may be different from the time that has passed in real time, however, this must not compromise consistency.

Why is Consistency required?

If a game or simulation resulted in different outcomes when run multiple times without changing starting parameters or other factors, it would mean potentially undesired randomness. For example, a game might be lost the first time it is run, but won the second time, without anything done differently by the player.

2.2.3 Using FixedUpdate instead of Update

It was found that, to ensure consistency, for most objects updated in the game FixedUpdate has to be used instead of Update. This is explained in the following section.

The Consequences of Changing Timescale

To illustrate the effect of changing timescale on the frequency of update steps, the time between update steps has been measured for different timescales:

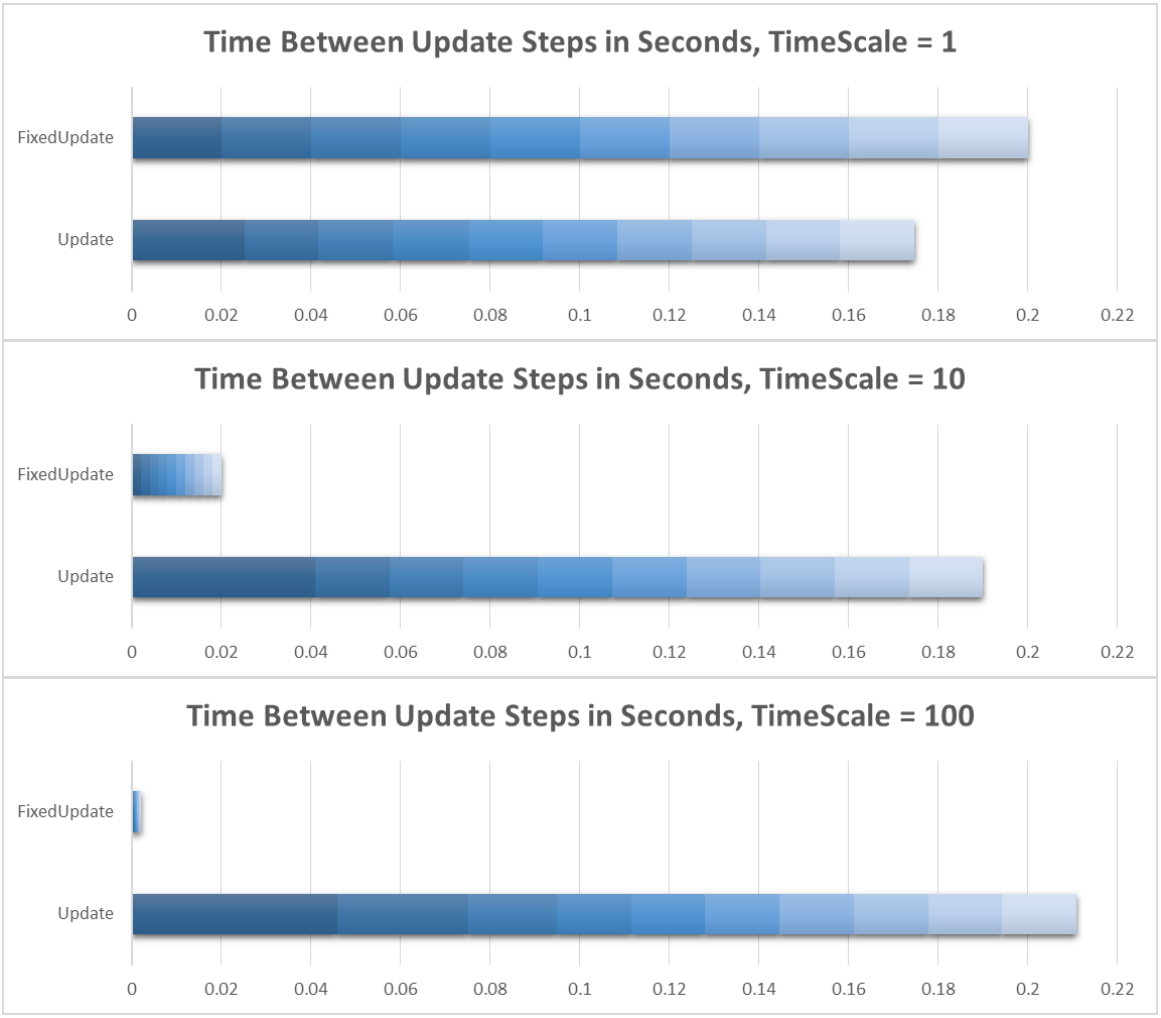

Figure 3

The measured results lead to the following conclusions:

- FixedUpdate always happens at regular intervals. This is also stated in the Unity Documentation (Unity Tutorials: Update and FixedUpdate 2014).
- Update does not happen at regular intervals.
- The frequency of FixedUpdate scales antiproportionally to the Timescale, while the frequency of Update seems to be unaffected by Timescale.

Why to use FixedUpdate instead of Update

In most video games created with Unity, Update is used to update moving objects in the game over time. However, in order to ensure a simulation consistency as defined earlier, it is required to use FixedUpdate for moving objects. This is because the intervals between Update steps are dependent on the current processing capacity of the computer,

and usually vary each time the game is run. Therefore, an event that is triggered by movement (e.g. a character reaching the end of the level) might happen at slightly different times when a simulation is run multiple times if Update is used, leading to inconsistency.

In contrast, as FixedUpdate always happens at the same interval independently from available processing power, using it for moving objects over time will ensure that each time the simulation is run will result in exactly the same outcome.

For fast-forwarding a simulation, this comes with the additional advantage that movement speeds are scaled as they should right away, since FixedUpdate frequency scales according to Timescale as shown in Figure 3.

An example of how using Update for movement can lead to inconsistencies:

Figure 4 shows a simplified illustration of some object in a game hitting a wall when FixedUpdate is used for the movement: Assuming the object moves at a constant speed, it moves an equal distance each frame that passes.

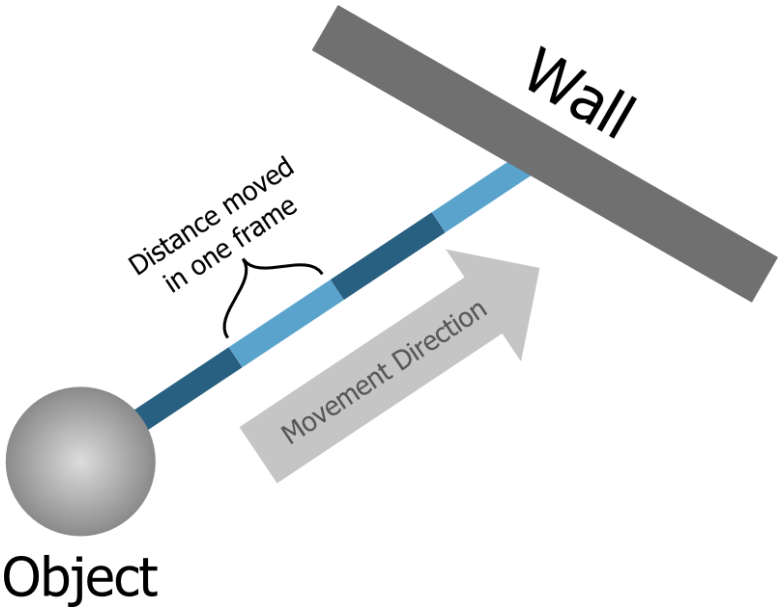

Figure 4: An object moving towards a wall in four FixedUpdate steps

As the FixedUpdate frequency is constant throughout simulations, the object will hit the wall at the same time in the same frame every time the simulation is run again. If Update is used for moving the object in the same scenario, this is not necessarily the case, as Figures 5 and 6 show:

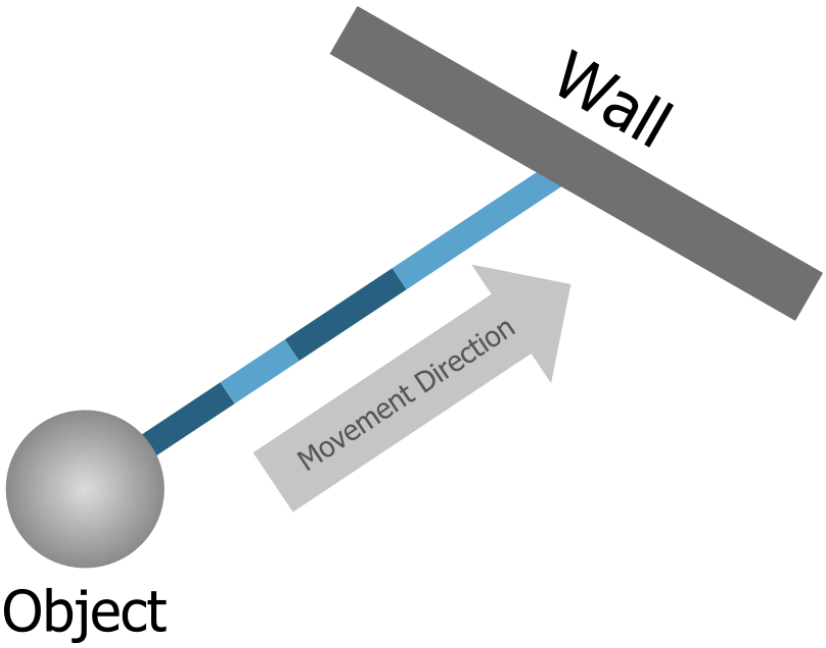

Figure 5: An object hitting a wall in four Update steps

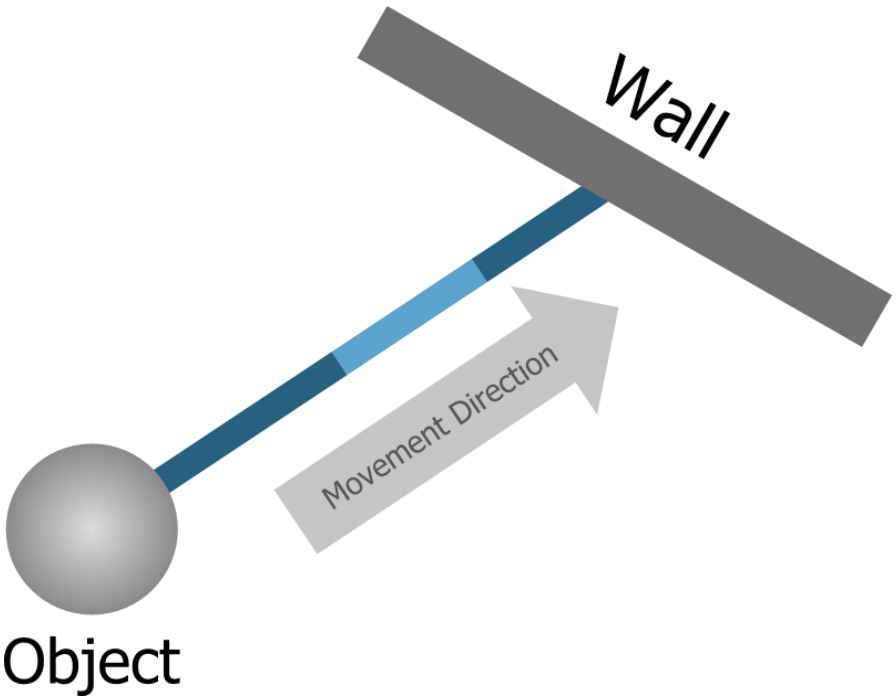

Figure 6: An object hitting a wall in three Update steps

Due to the irregular occurrence of Update steps, it is possible that the object hits the wall during different frames when the simulation is run twice. Comparing Figures 5 and 6, the object would hit the wall one frame earlier in Figure 6 than in Figure 5. This alone would not necessarily lead to an inconsistency, however, due to the irregular step lengths the points in time at which the collision is detected can be different one from

simulation to simulation. As a consequence, the collision might happen at a different times when the simulation is run twice.

2.2.4 Dealing with Player Input

Why player input still has to happen in Update()

Although, for the purpose of consistency, actions changing the game state have to be applied in FixedUpdate, it is still required to use Update for listening to player input for the following reason:

The input buffer is reset by Unity every Update frame (Unity Documentation: Input 2014). As shown in section 2.2.3, Update frames happen at irregular intervals which can be shorter than the intervals at which FixedUpdate frames happen. Was FixedUpdate used for checking for new input, it would be possible that input gets not registered in certain situations when Update frames happen faster than FixedUpdate frames. In this case, it could happen that the input buffer is reset before it is read during FixedUpdate. It is therefore preferable to still check for player input during Update frames, as also recommended by the Unity documentation (Unity Documentation: Input 2014).

As in-game actions triggered by user input still have to happen in FixedUpdate frames to ensure consistency as described in sections 2.2.3, it is required to listen to player input in Update frames, buffer it and apply it to the game in FixedUpdate.

Buffering player input

The player input is gotten during the Update step and there translated into the action that it will trigger. For example, if a key that makes the avatar jump is pressed, the action “Jump” will be added to the buffer.

At the next FixedUpdate step, the buffer is read, emptied and actions triggered by the input, if there are any, are applied to the game. That way, player input can be used without compromising consistency. Figure 7 illustrates this process on a timeline with some example actions.

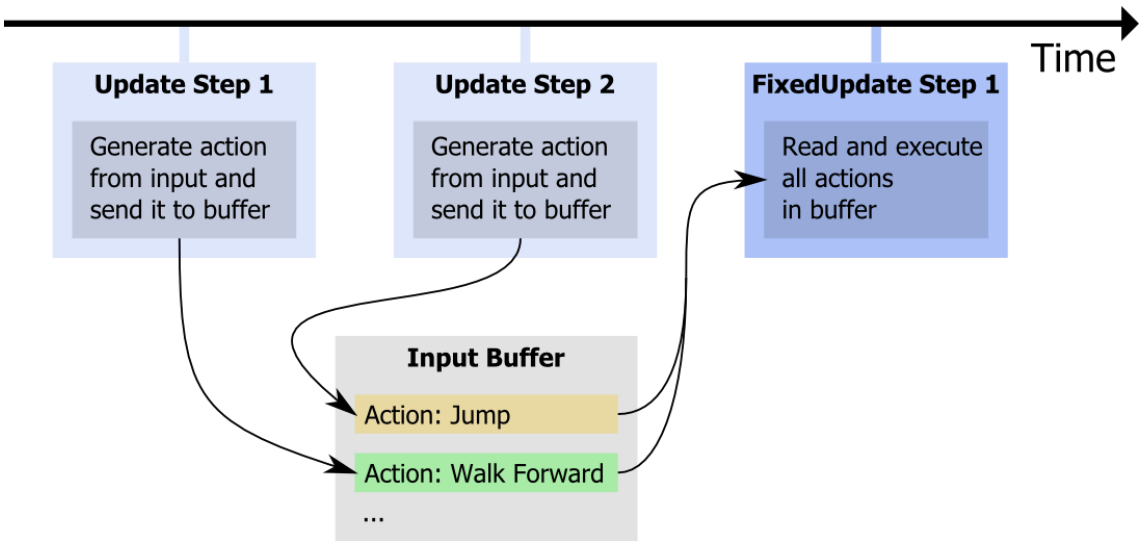

Figure 7: Buffering Input

Waiting with applying the player actions until the next FixedUpdate step will result in a small delay in reaction time to key presses. However, as FixedUpdate usually happens every 20 milliseconds, the delay will not be longer than that and therefore not noticeable for most games.

Recording player input

Travelling back and forth in time can implies that the same situation may be played multiple times, requiring that the player actions can be repeated without the player actively repeating the same actions again. The actions therefore need to be recorded and included in the simulation, so that the actions that have influenced a certain situation will happen the exact same way they have happened the first time when a situation is repeated.

Saving and repeating player input

In order to save and replay user actions, FixedUpdate frames are indexed. Whenever a player action is played for the first time in a FixedUpdate frame, it is saved together with the index number of the frame, so that it can be loaded again and assigned to the right frame for replay.

Figure 8 shows the additional steps that need to be taken in FixedUpdate in order to replay actions that have been done before.

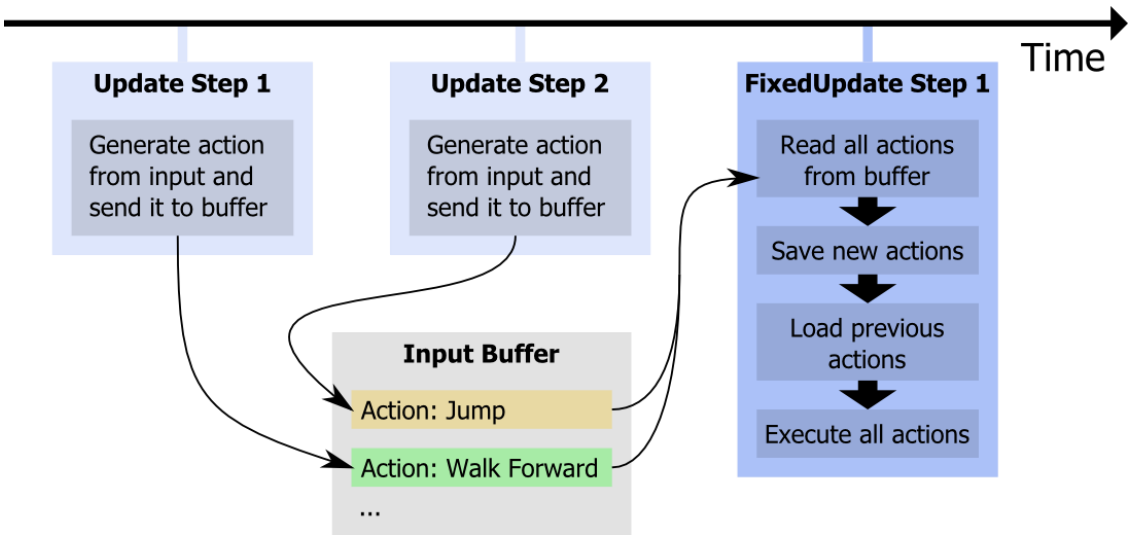

Figure 8: Buffering, saving and loading input

Dealing with rounding errors

Some variable types such as float have a limited precision (Dawson, 2012). During creation of the demo for this thesis, it has been found that subsequently, saving information about a player action and loading it again can imply rounding errors and therefore change information about that player action, compromising consistency. For example, if the player would spawn an object somewhere by clicking on a certain position on the screen, the saved action would have to save the spawning position of the object. When the situation is replayed, and the spawning position of the object is now loaded with slight rounding errors, the simulation is not consistent any more.

The rounding errors can be avoided by using variable types that do not cause rounding errors. In cases where this is not possible, the rounding errors need to be applied to the user action information before the action is applied to the game for the first time. As the rounding error will stay the same after having occurred once, applying the rounding error to the action before executing the action in the game will avoid consistency problems.

In Figure 9, the additional step needed to apply rounding errors when using variable types that cause them is included.

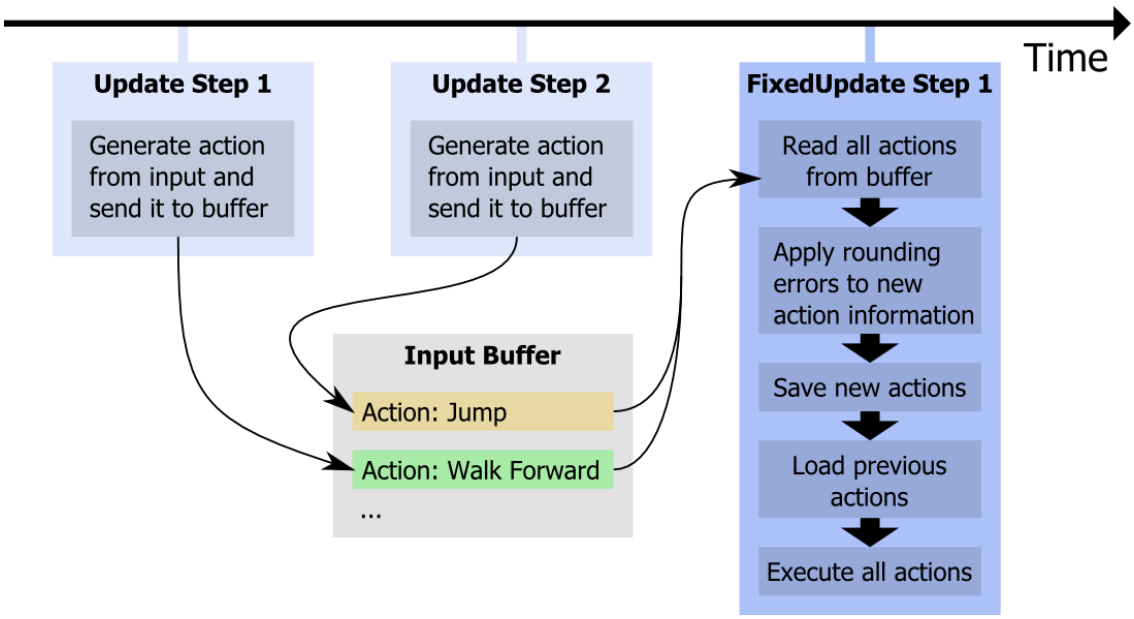

Figure 9: Dealing with rounding errors

2.3 Limitations

The following section discusses limitations of the time machine and how to work with them.

2.3.1 Fixed starting point

As travelling backwards in time is done by reloading the scene and forwarding from there until the desired point (as explained in section 2.1), the point in time at which the scene starts inevitably becomes the earliest point in time that can be travelled to, the "point zero" of the timeline. While it is not possible to travel to a point in time before the scene start point, it is possible to have the player start playing at a point from which they will be able to travel backwards. This can be done by forwarding the game to a predefined starting point before having the player enter it, as shown in Figure 10.

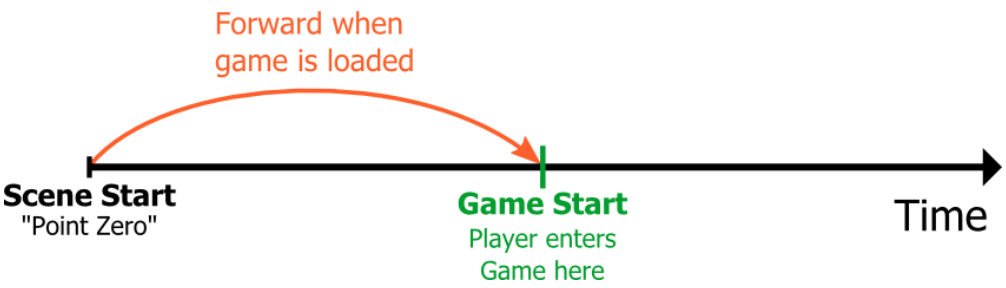

Figure 10

2.3.2 Waiting times when forwarding

While travelling backwards is limited by the scene start point, travelling forward does not have a hard limiting point, but is constrained by various factors which make travelling forward for too long distances impractical by causing too long waiting times for the player.

Usability implications of forwarding speed

While the game is forwarded, the player can not interact with the game and will therefore likely face some kind of loading screen. From a usability perspective, a relevant question is how long the loading screen can be shown before the user loses interest.

A long established guideline is that after 0.1 seconds delay a user will not feel that the application reacts instantaneously any more, after 1 second the user's flow of thought is interrupted, and 10 seconds is about the maximum waiting time for the user to keep their attention focused on an application (Nielsen, 1993).

Whether a player will actually stay focused on the game while waiting for it to forward will also highly depend on the kind of game and whether it is a casual game or not. Aiming for a relevant way of measuring forwarding speed, an interesting value to look at is the amount of in-game time that can be skipped by 10-seconds of forwarding.

Forwarding speed hard limit

The theoretical hard limit for forwarding speed is the maximum value Unity allows for Time.timeScale, which is 100. This means that forwarding at more than 100 times normal speed, or 100 in-game seconds per second, is not possible. Consequently, the maximum amount of in-game time that can be skipped by forwarding for 10 seconds would be 1000 seconds, or almost 17 minutes.

What slows down forwarding speed

Measuring forwarding speed in a real game shows that the actually possible forwarding speed is lower, which is partly due to the time machine code slowing down travelling speed before reaching the destination point in time to ensure a smooth arrival. Fast forwarding a Unity scene that was empty except for the time machine and a script to measure the results, a forwarding speed of around 87 in-game seconds per second has been measured on a 2014-built mediocre gaming laptop.

Using a same hardware, forwarding an early version of the demo project for this thesis including a lot of NPC spawning and AI resulted in a maximum forwarding speed of around 33 seconds per second. This shows that having many objects in the scene will additionally slow down forwarding.

The goal of Figure 11 is to indicate what can be reasonable distances to be travelled in a game:

Scenario	In-game seconds per second	In-game seconds travelled in 10 seconds	In-game minutes travelled in 10 seconds
Theoretical speed limit	100	1000	16.67
Empty unity scene	87.55	875.5	14.59
Scene packed with objects and AI	32.7	327	5.45

Figure 11: Forwarding speed

Assuming that the travelling loading screen should never be shown for longer than 10 seconds, the maximum allowed travel distance could be approximately between 5 and 15 in-game minutes. If the player is allowed to travel freely on the game's timeline, this will consequently also be the maximum length of the timeline. As a player's willingness to wait and actual forwarding speed can both highly depend on the type of game, the actual possible timeline length may vary significantly depending on the game.

Conclusion of forwarding speed

The above measurements indicate that a reasonable timeline length can be roughly 5-15 in-game minutes. However, as the willingness of a player to wait can also highly depend on the type of game they are playing, and hardware and platform a game is run on also influence forwarding speed, it is recommended to do individual benchmarking when developing a game with this thesis' time machine.

3 Time Travelling as a Game Mechanic

This part aims to put the approach to time travelling suggested in this thesis into the context by comparing it to a selection of games that employ time travelling as a game mechanic. While there are many games that use time travelling in some way, a few notable or relevant ones have been chosen.

3.1 Braid

Game description

Braid is a puzzle platform game which is about solving puzzles by manipulating time, with a rewinding game mechanic that enables the player to "learn from mistakes, but undo the consequences" (Number None, Inc.: Braid 2014). The game's plot is also themed around time and fixing mistakes. Braid was created by independent game developer Jonathan Blow and released for Xbox 360 in 2008, later for Windows and Mac in 2009.

Screenshot 1: Braid

How is time travelling employed as a game mechanic?

Braid uses time travelling mechanics in various ways: Firstly, the player can rewind time at any point to undo their actions. Many puzzles in the game are made of moving elements, of which some are unaffected by rewinding. In later levels, the flow of time is also manipulated using other game mechanics, such as the player's walking direction determining the flow direction of time, a "shadow" of the player repeating their past

actions, or the ability to slow down time in a certain radius around the player (Whitehead: Braid Review 2008).

On the technical level, Braid works fundamentally different from the time travel approach in this thesis: Braid saves the game state every frame so that rewinding is possible by resetting the game state to earlier versions. By putting effort into compressing the game state information, among others using serialization and leaving out information about objects that do not change over time, the memory space required by this approach is reduced to a level that keeps the approach feasible (Blow: The Implementation of Rewind in Braid 2010).

3.2 The Legend of Zelda: Majora's Mask

Game description

The Legend of Zelda: Majora's Mask is an action adventure game in which the player has to solve quests and earn masks to gain new abilities. The ultimate goal of the game is to save the game world from the moon crashing into it. It was originally released for Nintendo 64 in 2000 and rereleased for Nintendo GameCube in 2009 and was developed and published by Nintendo.

Screenshot 2: Majora's Mask

How is time travelling employed as a game mechanic?

Majora's Mask is set on a time line of three in-game days, which are about 54 minutes in real time, after which the moon will crash and destroy the game world. The player is given the ability to travel back to the beginning of the first day and also to alter the

speed at which time passes. While most of the game world is actually reset when the player travels back to the first day, progress of completed quests is kept by allowing the player to keep most of their items while time travelling. (Mirabella: Legend of Zelda: Majora's Mask 2000)

3.3 Achron

Game description

Achron is a multiplayer real-time strategy game that allows all players to travel in time and give commands to their units independently from each other within a certain range on the timeline. The game was released in 2011 for Windows, Mac and Linux. It is developed and published by Hazardous Software.

Screenshot 3: Achron

How is time travelling employed as a game mechanic?

While time is moving forward constantly while the game is running, each player can freely move in time within a restricted range around the current present. As events in the game that are in the past can still be changed up to a certain point, they only become permanently “happened” when they too far in the past for the players to be able to reach them. Giving commands at points in time distant from the present uses up an in-game resource called chronoenergy. This allows players to change past decisions that may have been unfavourable for them, given that they have enough chronoenergy. Units can also be send through time by special structures the players can build.

A time line view lets players jump freely using mouse clicks and also gives an overview on some important statistics, such as damage dealt and received over time. (Totilo: A Game Called Achron and the Theory That Games Would be Better With Time Travel 2011)

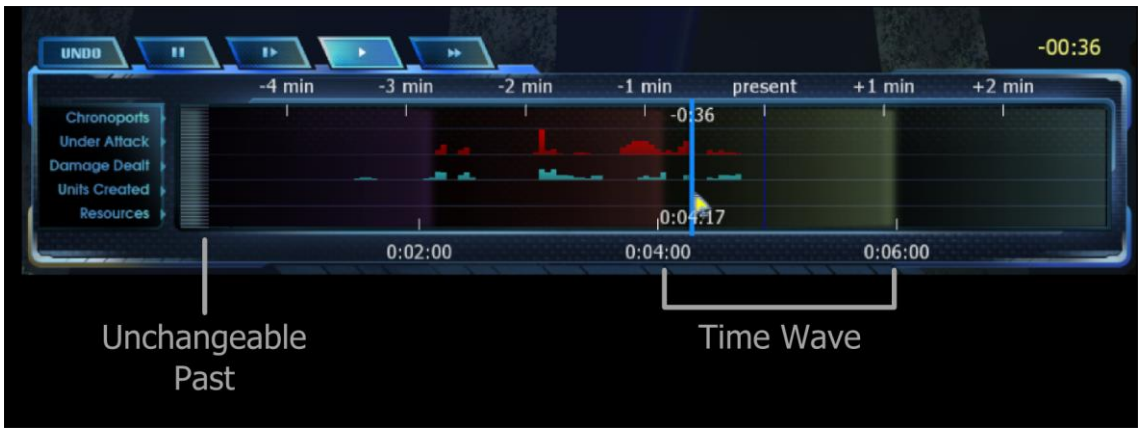

Figure 12: Achron time line

To deal with time travelling paradoxes, Achron has introduced the concept of time waves. A time wave is a section in time that is a closed-off space of causality, so that that the consequences of any action that is added within the time wave will only be considered by the game engine until the end point of that time wave. A time wave is visualized on the time line by a gradient.

If a grandfather paradox is caused by player actions, for example if a unit destroys the factory that has built it, the time waves are used to take into account both possible outcomes of the paradox: While in one time wave the unit will be alive and the factory destroyed, in the next one the factory will still be standing but the unit will not exist, then in the wave after that the unit will be alive again, and so on. This will be the case until the time wave containing the paradox has reached the unchangeable past, at which one of the outcomes will become permanent. (Hazardous Software Inc.: Achron and the Grandfather Paradox 2011)

Achron has been developed with an engine created with time travelling in mind, which includes the concept of time waves (Hazardous Software Inc.: Technology 2011).

3.4 Comparing the games to the Unity time machine

Earliest possible point

All presented games and the Unity time machine have in common that there is one fixed earliest possible point in the game. It is not possible to travel in time anywhere before that point. No game so far extrapolates into the past.

Travelling forward

Travelling to the future by forwarding is possible in Majora's mask and in games made with the Unity time machine. Achron does not allow forwarding, but jumping to points in time in the future within a small range. Unlike Achron, the Unity time machine does

not have a hard limiting point for travelling to the future, but travelling into very distant points in the future may be hindered by practicalities as described in section 2.3.2. Braid does not feature travel to the future other than running in real time.
A game made with the Unity time machine could be created to simulate the future, which could then be explored by travelling there.

Travelling backwards

Travelling to the past by smooth rewinding is only possible in Braid and Majora's mask, unlike in Achron and the Unity time machine. Achron features travel to the past by jumping to chosen points within a certain range, while the Unity time machine allows travelling to any point in the past up to the scene starting point. The option of travelling to the game start, which is rather trivial to implement, is available in Majora's mask and the Unity time machine. Smoothly rewinding is not possible with the Unity time machine.

Comparing time travelling features

Figure 13 gives an overview of time travel features in the presented games compared to what is possible with the Unity time machine:

Feature	Braid	Majora's Mask	Achron	Possible with Unity time machine
Fast forwarding	No	Yes, 2x speed	No	Yes
Rewinding	Yes	Yes	No	No
Travel to random point in future	No	No	Yes, up to 2 min	Yes
Travel to random point in past	No	Only to start point	Yes, up to 4 min	Yes

Figure 13: Time travel features in comparison

When comparing the different time travelling features, it is important to keep in mind that the games and their respective engines were developed to support different game mechanics. It may not be useful or feasible to have all the listed features in one game, but rather should the available features work well with the game mechanics and the overall context of the game. Furthermore, as the framework of the Unity time machine is compared to video games in the above chart, it is possible that the respective engines do support more features than are listed.

Conclusions from the comparison

It can be concluded that enabling smooth rewinding of a game requires a special programming architecture that either allows reverting all actions in the game or saves the game state every frame as described by Braid developer Jonathan Blow (Blow: The Implementation of Rewind in Braid 2010).

In the context of the games presented, the Unity time machine stands out by allowing free travelling on the whole time line of a game without being limited to a certain range around the present. Additionally, it encourages games that extrapolate the future in a meaningful way, as travelling into the future is possible with few limitations. The lack of a rewind feature may pose a disadvantage, as it could be a useful in order to visualize travelling to the past. As rewinding is not possible, using the Unity time machine it may be preferable to create games that aim to allow freely jumping to different points in time rather than smoothly forwarding and rewinding.

4 Discussion

4.1 Results

4.1.1 Implementing the time machine

The Unity time machine has been successfully implemented and notable issues and restrictions have been researched in this thesis. The main feature, enabling freely jumping in time on a given timeline, is functional. Necessary restrictions and limitations have been documented:

Consistency restrictions

The restrictions that apply when developing with the Unity time machine are defined by the requirement of consistency as described in section 2.2.2, which comes down to a requirement of the game or simulation being deterministic when leaving out any user input.

One of the main restrictions is that updating processes that happen over time cannot be done in Unity's Update step, but has to be done in FixedUpdate instead. User input has to be precisely saved and replayed when included in the simulation. When saving user input, possible rounding errors caused by float precision have to be taken into account before executing the actions required by the user input in the game.

Travelling limitations

A game created using the Unity time machine will have one earliest possible point in time, which is the point at which the scene starts. It is not possible to travel to any point in time before that point.

Due to the time taken by jumping long distances in time, the longest travelling distance allowed by the game will likely be around 5-15 in-game minutes. This can however greatly vary depending on the type of game, objects in the scene and the game's target platform. If the game allows free travelling on the whole time line, the longest allowed travelling distance consequently also defines the maximum length of the time line.

Reliability

The restrictions placed on the game development by this time machine may pose a reliability issue, as there are many possible ways of breaking the required consistency which may impair the game development process. While this is important to consider when developing with the Unity time machine, it is an issue that has turned out to be not too difficult to overcome during the development of the playable demo. Ethical concerns are not applicable to this thesis.

Experiments have shown that trying to travel unreasonably long distances in time can cause an application to crash and should therefore not be allowed by the game logic.

4.1.2 Time travelling as a game mechanic

Researching relevant games shows that time travelling related features have been included in a notable amount of video games in various ways. The Unity time machine has some features that are commonly available in time travelling-themed games, and some that aren't.

The option of freely travelling to any point within a certain time line could make a game created with the Unity time machine stand out from other games that employ time travelling game mechanics. Due the way travelling backwards in time is implemented, smoothly rewinding time is a feature that the Unity time machine lacks in comparison to some other games.

Game mechanics need to make sense in the context of the game theme and story they are used in. The characteristics of the Unity time machine encourage games that allow travelling long distances in time and play with causality by letting the player explore long term effects of their past actions.

4.2 Development suggestions

The time travel functionality of the Unity time machine invites to create games that are heavily based on travelling far distances in time. Gameplay elements that involve a lot of cause and effect, such as certain events triggering other events which then again trigger other events could make the option of time travelling really interesting, as the player could play with influencing events and see the effects in distant futures. Visualizing causal relationships between events could greatly enhance this experience by giving the player an overview on the time line, and showing which event happened for what reason.

While simulation and puzzle games are genres predestined to work well with the time machine, it is a tool suited to build time travelling games also for other existing genres and experimental games.

5 References

Blow, J. (2010). *The Implementation of Rewind in Braid.* Read 18.11.2014. http://gdcvault.com/play/1012210/The-Implementation-of-Rewind-in

Dawson, B. (2012). *Don't Store That in a Float.* Read 15.11.2014. https://randomascii.wordpress.com/2012/02/13/dont-store-that-in-a-float/

Hazardous Software Inc. (2011). *Achron and the Grandfather Paradox*. Read 10.11.2014. http://www.achrongame.com/site/achron-and-the-grandfather-paradox.php

Hazardous Software Inc. (2011). *Technology*. Read 10.11.2014. http://www.achrongame.com/site/technology.php

Mirabella, F. (2000). *Legend of Zelda: Majora's Mask.* Read 26.11.2014. http://www.ign.com/articles/2000/10/26/legend-of-zelda-majoras-mask

Nielsen, J. (1993). Usability Engineering. AP Professional.

Number None, Inc. (2014, November 25). *Braid*. Read 25.11.2014. http://braid-game.com/

Totilo, S. (2011, November 7). *A Game Called Achron and the Theory That Games Would be Better With Time Travel*. Read 22.11.2014. http://kotaku.com/5820164/a-game-called-achron-and-the-theory-that-games-would-be-better-with-time-travel

Unity Documentation. (2014, November 21). *Input*. Read 03.11.2014. http://docs.unity3d.com/ScriptReference/Input.html

Unity Documentation. (2014, November 05). *Time.timeScale*. Read 28.10.2014. http://docs.unity3d.com/ScriptReference/Time-timeScale.html

Unity Technologies. (2014). *Create serious games with Unity*. Read 28.10.2014. https://unity3d.com/industries/sim

Unity Tutorials. (2014, November 5). *Update and FixedUpdate*. Read 28.10.2014. http://unity3d.com/learn/tutorials/modules/beginner/scripting/update-and-fixedupdate

Valente, L., Conci, A., & Feijô, B. (2005). *Real Time Game Loop Models for Single-Player Computer Games.*

Whitehead, D. (2008, August 6). *Braid Review*. Read 29.11.2014. http://www.eurogamer.net/articles/braid-review