Nordavind - Report (Tech)

Eirik Gjeruldsen and Dexter André Osiander

May 23, 2018
Inland Norway University of Applied Sciences

Contents

Introduction

Preface

This report is written to explain the development process of our game, *Nordavind*. Our supervisor has permitted us to write with a higher degree of jargon, so we assume the reader is knowledgeable in terms regarding game design, game programming, mathematics, and project management. We also assume some familiarity with the software we have been using, such as Unity, Google Docs and Google Sheets, Slack, GitHub, and other related software.

We also want to clarify that all of the level design and music have been produced by external partners, and our group is by no means claiming this work as our own. The Game Design Document (GDD) has also mainly been written by our remotely working level designer. That is also why we will not discuss those topics in this report unless we feel a need too.

We want to thank the following:

- Jakub Rymsza, for contributing with the level design, much of the game design, and also for writing the game design document (GDD).
- Robert Sørensen, for composing and recording the original soundtrack.
- Our supervisor, Ole E. Flaten, for overseeing the development.
- Catharina Bøhler and Luke Knighton for their expertise and support during the preproduction stage.
- Trond Fasteraune for his technical advice throughout the development.
- Hamar Game Collective, for organizing the TestBonanza for us to participate in.

Seeing as our group consists of five people; two programmers, and three artists, we have decided to deliver both exams, so the examiners have access to the entire context of the project. It should be understood that only the folder named “(Tech) NeonPeon” is directly relevant to this report.

Pilot Assignment

Our assignment was to create a product related to our education, such as a game, a film, a web application, an art installation, or similar. We chose to create a 3rd-person action game. A summary of the specific requirements for a pilot assignment game are as follows:

- Compiled program in a format which has been agreed upon by the supervisor.
- The game will offer a functioning game world that effectively displays the game's gameplay, animations, and effects.
- Assets that have not been integrated in the game, but that are to be examined, has to be displayed in turntables, video files, images, etc.
- Source code and project files for the development platform are to be included (C# scripts, JavaScript, Unity-files, shader files, etc.).
- All work files for assets (Maya, Photoshop, MudBox, Sounds, etc.) are to be included.
- Various development documents are to be included.
- Include a manual that instructs the user on how to play the game.
- A video of a walkthrough of the game is to be included.

We have agreed beforehand with our supervisor that our game will target only PC, and we will not focus on porting it to other operating systems such as Mac, Linux, mobile operating systems, or consoles. Some shaders may not be working on Mac or Linux, because we have not created fallback shaders for some of the advanced materials.

To wrap up this introduction, we present our topic question, which will act as the theme of this report. It is as follows: "How can we create technical solutions that comply with game design choices? And how can we manage a group of five students developing a game?"

The Game

This short section will deliberately not focus so much on the topic question, but instead act as a reference for important game design decisions, as well as to describe our philosophy when deciding on the design of the game.

Description

Nordavind is a 3rd-person action game set in wintery, rural Norway. The game features a child, René, whose gameplay abilities emphasize playing in the snow:

- Throw snowballs to defeat snow monsters.
- Roll around in the snow, and feel the snow respond to your footsteps.
- Sled down hills on your bum slider.

The product will be a vertical slice of the full game, and will be playable for PC. It is recommended to play with a gamepad, but it is also designed to work with a keyboard and mouse.

Game Design

Preproduction

During our previous semester, we finished a course named “Preproduction” in which we designed most of the game, and laid out the production plan for this current semester. Here, we will summarize the most important events and work-related decisions we have made during that semester.

What was important for the team before we started actually making the game concept was that each team member were satisfied with the outlook of the tasks involved in making this game. We decided on some skill-wise constraints that we wanted for the game, one of which was to make a 3D action-adventure with platforming elements. Another one was to include combat, and a third important one was that we wanted to implement advanced shaders. These were decisions made by the team as a whole before we even started to map out the idea of the game, based on our individual desires to explore those fields in game development. Being an aesthetically-oriented game, we also wanted to avoid “gamey” elements, including HUD graphics, text prompts, and generic menus. This would allow the player to be more attentive to its surroundings, instead of, for example, constantly looking at a GUI element. With these constraints in mind, we went forth to develop an immersive game. Adams (2014, pp. 512) describes immersion as: “The feeling of being submerged in a form of entertainment and unaware that you are experiencing an artificial world. Players become immersed in several ways: tactically, strategically, and narratively". We designed our game with the narrative immersion in mind.

Design Approach

Although our level designer from Poland, Jakub, contributed much to the game's design, the entire team participated in designing Nordavind.

We had decided early on that we wanted the game to be a aesthetically-focused game, with emphasis on the wintery feeling of rural Norway. We designed largely in the AMD direction, keeping the design techniques of MDA in mind (Hunicke, LeBlanc, & Zubeck, 2004). To summarize this design approach, it refers to how a developer sees a game versus how a player sees a game. We designed starting from the player's impressions in mind, shaping the mechanics to suit these impressions. This is opposed to designing from the MDA direction, in which you would first design the mechanics, and then let the impressions arise from there. Both are valid approaches, but it is important to keep a consistent approach in mind when designing games.

Project Management

Working with the game as a project (as opposed to a product) requires clarity of the game's vision, the task dependencies, individual progress, and deadlines. In order to execute the project properly, we assigned a person the role of project manager - a person who controls and oversees that every planned asset is either being made or being consciously scrapped, and that the team meets its deadlines.

Development Plan and Scrum

During preproduction, we created a development plan (see Attachment #1 - Development Plan), which is a Gantt diagram showing the milestones we had to reach at specific deadlines. We combined this with the Scrum methodology, as further explained later in this chapter.

Setting our goal as the vertical slice of a game, we strictly followed our development plan in the starting phase of production. But we soon experienced throughout the production that it was difficult to follow a Gantt diagram because they are not meant for frequent changes. We instead adopted the *Agile* methodology, as explained in *Principles behind the Agile Manifesto* (Beck et al., 2001), which can be summarized as embracing the need for change in a project, thus rejecting strict development plans.

According to Luke Ahearn in his article *Budgeting and Scheduling Your Game*, "If a team member fails to complete a task or is late in completing it, the project will hit a standstill..." (2001, pp. 2), showcasing just how unsuited Gantt charts are for something like game development, which is a process that requires constant adaptability. Therefore, Gantt diagrams directly contradict the principles of *Agile*, which is what Scrum instead tries to implement in practice. That is why we embraced the Scrum methodology.

We drifted away from strictly following our Gantt-based development plan and instead set internal deadlines for ourselves for reaching the first playable, alpha, beta, and vertical slice stages. Working directly towards these, and then allowing for more freedom in our work methods, let us keep the focus on quality, rather than some arbitrary demands. For example, if the need arose for the entire team to focus on a single character, then we would do that.

To aid our newly adopted work methodology of focusing on sprints and assets, rather than milestones and predefined workflows, we felt the need to create an asset list (see Attachments 2-5), which would serve as the new "development plan". In this, we split the important assets into three distinct categories: "Characters", "World", and "Systems". In these asset lists, we list each asset as a main category, with the work details listed as subcategories. We mark the priority of every part of the asset, from 1 (very important) to 3 (not important). For example, for the main character asset, the movement mechanics would be priority 1, whereas "idle stance #2" would be of lower priority, such as 3. We ended up making good use of these asset lists, using them as our backlog for the upcoming Scrum sprints, and letting the team have a clean overview of the tasks yet to be finished. We want to make a note that we still did not follow this asset list strictly, as our game constantly changed throughout development. Examples of features that are still in the asset list, but that were not implemented, are checkpoints, the stable, and enemy "hurt" sounds.

One exception to our rejection of Gantt diagrams will be described further below, in the "Production Pipeline" section, where we found great use in applying the strictness of the Gantt diagrams' schedules.

Decision-Making

Creating a game does not always go the way it is planned, and it is important to be able to scale down the project, should the need arise. The robustness of the original game design is put to the test when it is actually being implemented, and the team members' skills can both liberate and limit the possibilities of implementing certain game features. In our case, we experienced a bit of both.

Competence

On one hand, our two programmers had different skill sets and experience in game development. We were also very different in our approach to programming tasks; one being an efficient prototyper (Eirik), the other being a cautious planner (Dexter André). Both of these approaches have been beneficial to the project, and we have assigned the major tasks based on these traits: Eirik programmed combat and AI, and ended up becoming the generalist of the project; Dexter André programmed the avatar mechanics and created some of the advanced shaders.

On the other hand, being overly cautious in planning a feature could end up wasting more time than it had saved, and being overly prototypical could result in a messy base for a system which would later have to be cleaned up or rewritten.

This occasional discrepancy between game design and competence lead to some important decisions regarding the game's features. We mention some of these later during the "Game Mechanics" section, whenever it is relevant.

Decision-Making Structure

During our previous course, Preproduction, we decided that we wanted to avoid a flat structure. A flat type of structure would mean that every team member has equal say in the designs of the game. Coming from many different disciplines, this could be viewed as a positive influence, but it ended up being very tedious and time-consuming during Preproduction, often having to explain basic concepts of game design to those with little experience within in. The meetings in Preproduction would last for hours deciding on things that should have just been based on merit, rather than free voting. So when we started out on the production of Nordavind, we had decided that we wanted to avoid such a democratic structure, and we decided on a hierarchical structure as follows:

1. Project Manager: has the final say on whether to keep or scrap a feature. Bases decisions on the production's deadlines.
2. Project Owner: decides the game's vision. Corrects inconsistencies in the game design, visual direction, and so on. If a certain feature requires too much time, then the project manager still has final say.
3. Art Director: ensures that all the graphics follow a consistent style as described in the *Style Guide*. Makes sure that the concept art, 3D, animations, and visual effects adhere to the style of the game, which in turn is decided by the project owner.
4. Individuals: each team member micromanages only themself, and makes sure to meet the deadlines. They can suggest features, but it is up to the project owner, and ultimately, the project manager, to decide whether it makes it into the game.

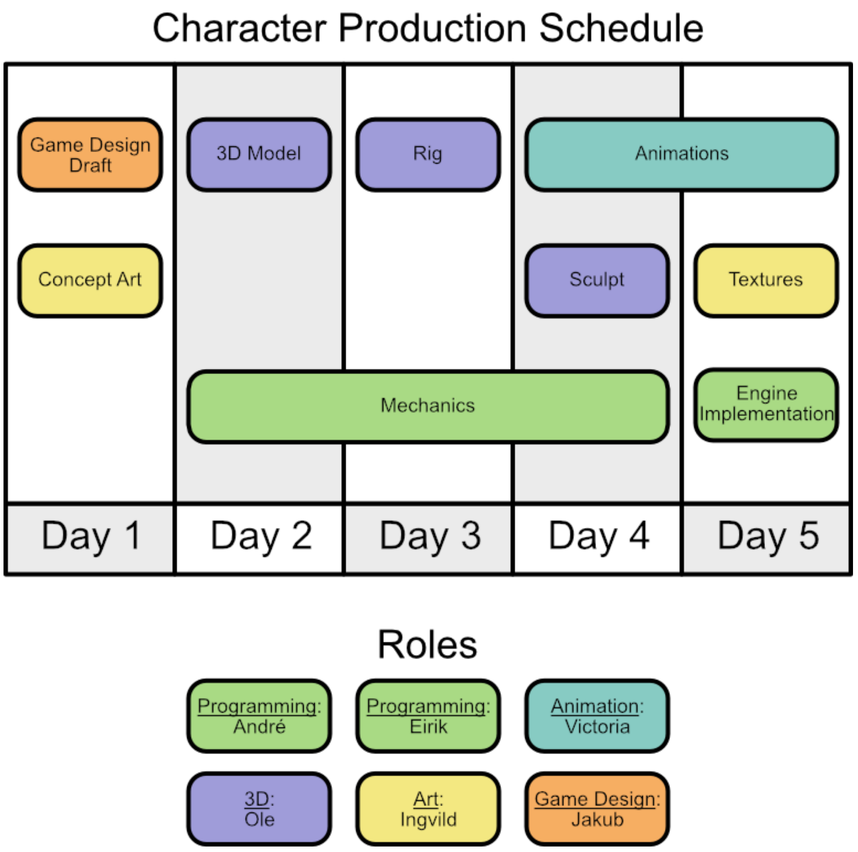
Production Pipeline

Certain tasks - such as animating a character - are dependent on other tasks to be finished first. Throughout this chapter, we will refer to any task upon which another task is dependent as a “dependency”. For example, the task of rigging is a dependency, because animation would not be able to start without a rig; texturing is not a dependency, because there are no tasks awaiting the texturing to finish. Thinking in terms of dependencies has helped us streamline our production pipeline so that every team member always had a task to do.

The game’s characters required the most interdisciplinary cooperation. Our game focuses heavily on the characters involved, and we have created a total of five core characters for the game, producing more efficiently with every iteration.

Generally, the production pipeline for a character could be split up as follows:

1. We describe the character on paper, following game design requirements. Meanwhile, the artist starts with the character design and comes up with a visual concept.
2. The programming starts, and runs in parallel with the other tasks. 3D modelling also starts.
3. After completing the model, rigging will immediately start.
4. After rigging is finished, the animations can start. The high-resolution sculpt also starts at this point.
5. After the sculpt is finished, the texturing can start.
6. After the core animations are finished, we start implementing the animation system, sound- and particle effects, and other minor tasks.



Production pipeline for a character. Note that every color-coded role is only busy with one task at a time for maximum efficiency. We have omitted miscellaneous tasks such as sound design, animation handling, and visual effects because they do not cause any dependencies.

In the diagram above, we have simplified the character production pipeline for a *core* asset, meaning that we can make minor changes to it after it has been implemented. For example, implementing the asset before the sound effects have been finished, would be fine, because they aren't dependencies. In reality, we would continue to add extra animations, textures, mechanics, etc., after the asset had been implemented, and this is how we spent our time when we were not assigned to a specific task in this schedule.

If need be, we could have altered the pipeline to allow for the texturing before the animations, but seeing as animations are more work-intensive, we prioritized them before the texturing. Having tested this pipeline during the *Aesthetic Prototype* sprint (see Attachment #1 - Development Plan) helped us in making such prioritizations.

Task Queue

The production pipeline for a character in Nordavind was largely queue-based, with each task (e.g. rigging, concept art, etc.) being one element of that queue. Once the early tasks were finished, that person would then go on to start doing their work on the next character. Working with the established pipeline as described above, we managed to fit the production of several characters in overlap so as to maximize efficiency.

In the pseudo-Gantt diagram below, we show how we fit several characters with a production time of 5 days into a shorter time period. Having 5 characters total in the game, producing them linearly after each other would have taken $5 \cdot 5 = 25$ days to complete. Instead, we managed to implement all character assets (with their core features) within a period of 17 days. Specifically, starting the production of a new asset at day 4 of the current asset production, let us sustain a predictable and effective character production pipeline.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
Game Design	🟧			🟧			🟧			🟧			🟧		
Art		🟨			🟨			🟨			🟨			🟨	
3D		🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	
Animation				🟦	🟦		🟦	🟦		🟦	🟦		🟦	🟦	
Programming		🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	

This chart shows the core character production pipeline diagram from earlier, but with overlapping assets. The different asset periods have been outlined in the upper timeline, where for example the first blue line marks the start of the second asset, and the second blue line marks the end of the second asset.

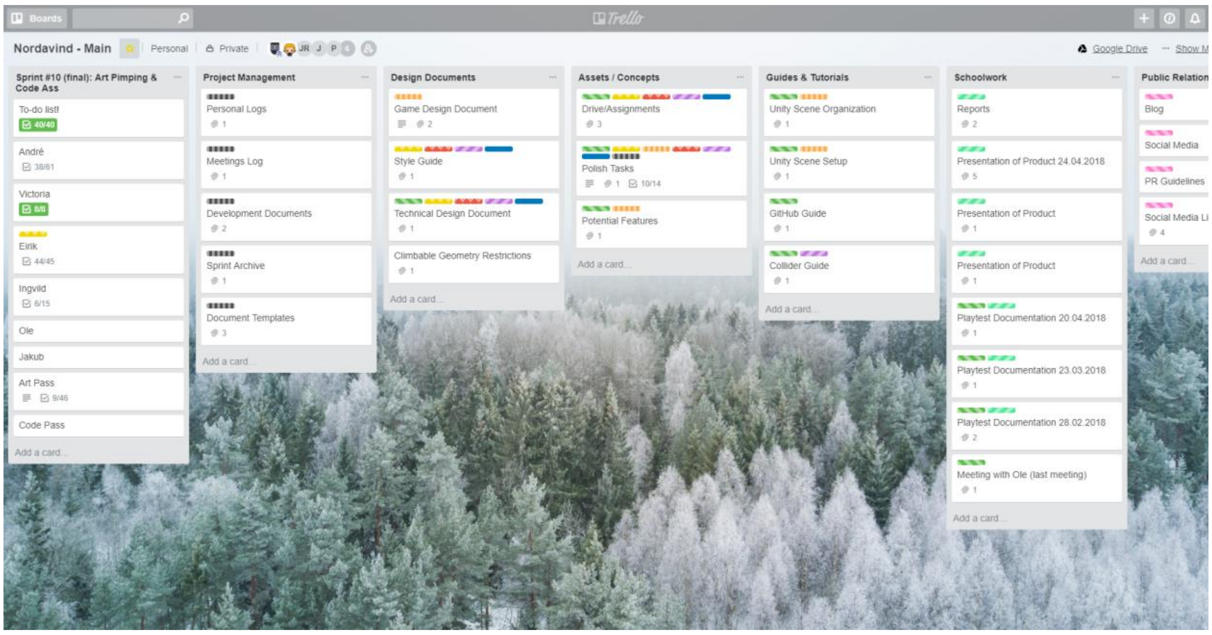
Project Management Tools and Techniques

This section will summarize some of the project management tools and techniques we have used as part of the production and preproduction.

Software

Trello

Trello is a great way of implementing the Kanban-style of visualizing the project canvas. As mentioned in their book, *Prosjekthåndboka 3.0* (Aakre & Scharning, 2016, pp. 189), it is important that everyone sees the project canvas. We invited everyone one our team to this board with full edit access so that everyone could personalize their digital workspace, giving more customization options to each individual team member. The flexibility of Trello also synchronizes with *Agile*. This suits game development perfectly, as the production is prone to change throughout the development.



Screenshot of our Trello board for Nordavind.

Slack

Taking advice from our guest lecturers during the Preproduction course, we set up our own Slack workspace in order to communicate without distraction (Bøhler & Knighton, 2017, pp. 7). Slack has worked very well for our team in communicating while working remotely, and also while sitting at the same table while working. We used Slack as part of our morning meetings to communicate with our remote level designer on voice chat, and could quickly upload a video file to show team members a feature or a problem, all without having to get up from our seat, and without wasting other team members' precious work time.

Google Drive

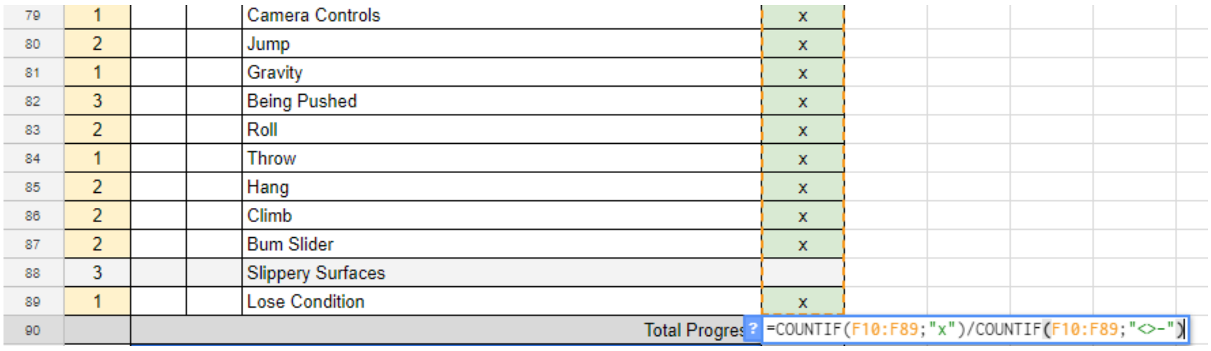
We used Google Drive for our larger uploads, and it acts as a storage hub for all of our important project management-related documentation, such as the development plan, the asset list, sprint archives, and school-related documents. We created a Gmail specifically for this project, and integrated Google Drive in both Trello and Slack to streamline our workflow.

Google Docs

For large text documents, we used Google Docs to collectively edit documents like the game design document, the style guide, the exam reports, and many other.

Google Sheets

We created both our now-obsolete development plan and also our current asset list in Google Sheets. We learned a few tricks along the way in order to make the most of it - especially how to implement a dynamic checklist; one that counts only what has been ticked off as "complete". For more details, see Attachments 2-5.



79	1			Camera Controls	x
80	2			Jump	x
81	1			Gravity	x
82	3			Being Pushed	x
83	2			Roll	x
84	1			Throw	x
85	2			Hang	x
86	2			Climb	x
87	2			Bum Slider	x
88	3			Slippery Surfaces	
89	1			Lose Condition	x
90				Total Progres	=COUNTIF(F10:F89;"x")/COUNTIF(F10:F89;"<>-")

Excerpt of the "Characters" section of the Asset List. The formula for the total progress of the player avatar is highlighted in the image. In here, we tell Sheets to divide the amount of x-es on the total amount of fields, and we also tell it to ignore fields that have a hyphen in it. This allows us to dynamically connect the sheets together and automatically update whenever a change has been made.

GitHub

While GitHub itself is not necessarily needed for project management, it certainly helps with organizing the versions of the game. We were four active users on GitHub at all times, only two of which were programmers. Both our remote level designer and our in-house 3D artist used GItHub in collaboration with the programmers, and this required us to train them in using the tool. We were unfamiliar in using GitHub with Unity, and ran into several problems, such as .meta-files cluttering up the commits, merge conflicts, reversion failures, and file-naming problems. Occasionally, we lost a few hours of work, but using GitHub became a regular habit after a few months, yielding more benefits than pains.

Methodology and Work

Agile and Scrum

As mentioned above, we use the *Agile* methodology for our project. This means we focus on adaptability rather than a strict development plan. One way of implementing this is through *Scrum*. Working with Scrum means that we set up sprints - 1-2-week long periods of work - for a fixed goal. Working in smaller iterations like these lets the team comfortably scale down our tasks and divide the entire project into several digestible pieces. Although the lengths of the sprints varied from each sprint, they usually lasted for two weeks during our production, with meaningful goals like "Aesthetic Prototype" and "Boss Fight", etc.

Meetings

Although much of the game was decided during preproduction, we needed to make changes throughout the project. The latest important change we had to make in the project was to implement an anti wall-jumping mechanism. This happened approximately one month before the final hand-in, which required much testing and balancing. This goes to show that our project was under constant change throughout production. In such events, we would normally bring in the relevant team members and discuss the issue, using the decision-making structure seen earlier.

As part of the Scrum work method, we would also attend morning meetings. These would last approximately 15 minutes, in which each of us mention:

1. What we have been working on yesterday.
2. What problems we have been facing.
3. What we will be doing today.

Work

We had decided during preproduction that we would commit to sitting together as much as possible, based on Catharina Bøhler's and Luke Knighton's 6 tips for better communication in their split lecture, *Project Management - Getting a good start* (Bøhler & Knighton, 2017, pp. 7). After attending the morning meetings, we would work together at the same table from 9am to 4pm every school day, with exceptions being hardware requirements that were not present at school, or our part-time jobs. For example, our concept artist, Ingvild, would work from home at her stationary drawing tablet whenever fine-detail work needed to be done.

Programming Patterns and Architecture

Finite State Machines (FSM)

Finite state machines are very useful in games when you need to switch between several states. Unity's built-in animation system is a FSM, and we have implemented our own for the player avatar's abilities, to ensure smooth transitions between them. Robert Nystrom explains the "State" pattern in his book, *Game Programming Patterns*, as allowing an object to alter its behavior when its internal state changes (Nystrom, 2014, pp. 87). This is further expanded upon to create a system that controls its own transitions between states.

Gameplay Modularity

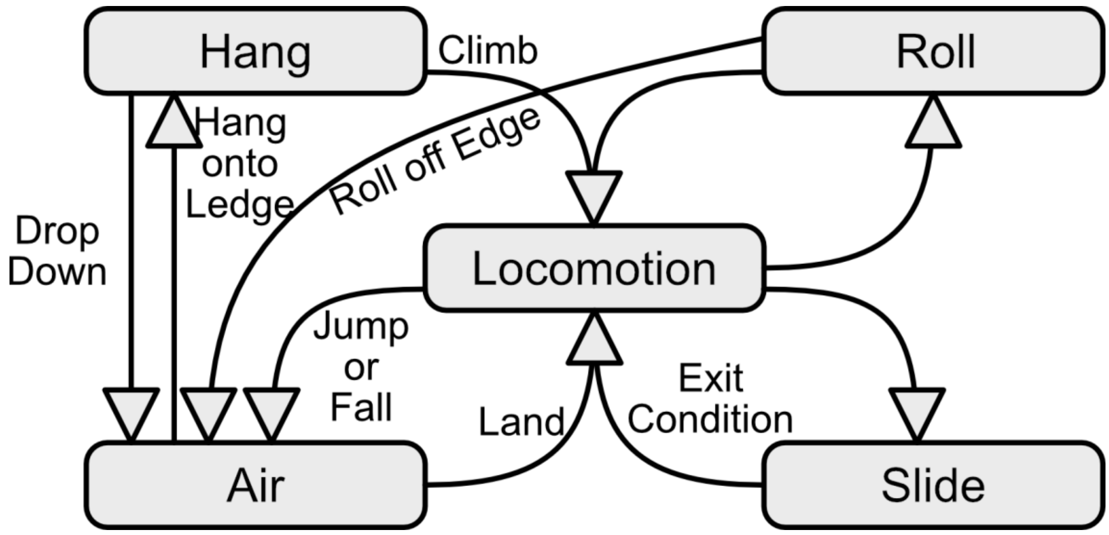
Avatar Abilities FSM

René (the protagonist) can perform many different abilities, from standard locomotion and jumping, to bum sliding and throwing snowballs. In order to support the finite state machine that is the avatar mechanics, we needed the different gameplay abilities to be as decoupled as possible. This was important to avoid bugs in situations where we had to disable an ability. For example, in the start of the game, the player has not yet found the bum slider. We have programmatically disabled that script without any problems. This was one of the requirements we set for all the different gameplay scripts.

We wanted to implement the "Component" pattern following Bob Nystrom in his book *Game Programming Patterns* (Nystrom, 2014, pp. 213). Unity also uses the same decoupling pattern for its GameObjects, so it should have been easy for us to comply with the same architecture for our different avatar mechanics. We followed a simple set of rules:

- The finite state machine has enums which are named after each of the different gameplay states, such as "Locomotion", "Air", "Roll", etc. This script is attached to the player avatar. In Nordavind, the avatar ability FSM is called "PlayerMovement.cs", which also handles the movement based on which state is currently active.
- Each gameplay ability script can set the states of the ability FSM, while the ability FSM does not set its own states, with the exception of detecting automatic transitions to the "Air" state (which should happen regardless of having to disable a script, anyway).

In the image below, we have outlined a simplified version of the FSM we have used for the avatar mechanics. Most of these transitions are controlled by the respective scripts, such as "PlayerRoll.cs", communicating to the "PlayerMovement.cs" script whenever a signal is triggered. We have omitted the text explaining the obvious transitions, such as when pressing "B" on the gamepad to roll.



Simplified version of our avatar mechanics FSM.

Flaws

Although the avatar ability FSM fulfilled its main goals, it was somewhat difficult to maintain whenever we wanted to add or change abilities. For example, making a change to the actual mechanics of the bum slider would require us to change the "Movement()" function located in the ability FSM, while also having to change the enter and exit conditions located in the bum slider script. As such, we did not truly implement the *Component* pattern.

In hindsight, it would have been better to implement Delegates and Events, by sending movement signals to the respective ability scripts, rather than handling all the different movement from the ability FSM itself. This method would also have adhered more closely to the true nature of the Component programming pattern because it would have been more decoupled. For example, we could avoid having to change the ability FSM whenever we only wanted to change the bum slider ability mechanics.

Game Mechanics

Avatar Mechanics

Physics and Movement

Locomotion

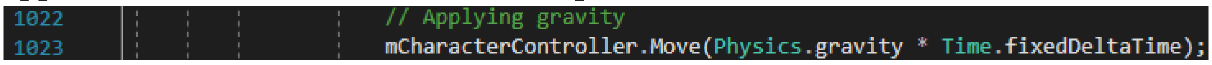
We have implemented a standard 3rd-person locomotion mechanic, which takes the camera's view into account by multiplying the camera's quaternion with the player left stick input, presented as a Vector3. We then project this onto the current ground plane.

Detecting the current ground plane was fairly straightforward, raycasting downwards from the player avatar's position to retrieve that triangle's normal. This technique is used throughout the game to correct movement onto surfaces, e.g. in the bum slider, which relies heavily on reading the current ground triangle.

CharacterController

We decided early on that we wanted to use Unity's built-in CharacterController component - a script that enables easy implementation of simple movement. The main reason for using a CharacterController (as opposed to using a RigidBody) was that it can automatically climb small steps as well as having easy control over slopes, which we wanted to avoid having to program ourselves.

We realized later in the production that this was not at all a good choice, because we had to handle all the physics interaction ourselves. But we did not have time to re-implement the avatar mechanics using a RigidBody instead, so we had to stick with the CharacterController. We apply a constant gravity to the CharacterController's *Move()* function as shown below. This happens after all the other movement is applied so as not to interfere with the important movement calculations.



```
// Applying gravity
mCharacterController.Move(Physics.gravity * Time.fixedDeltaTime);
```

Applying our own gravity to the CharacterController as seen in the "PlayerMovement.cs" script.

Jumping Style

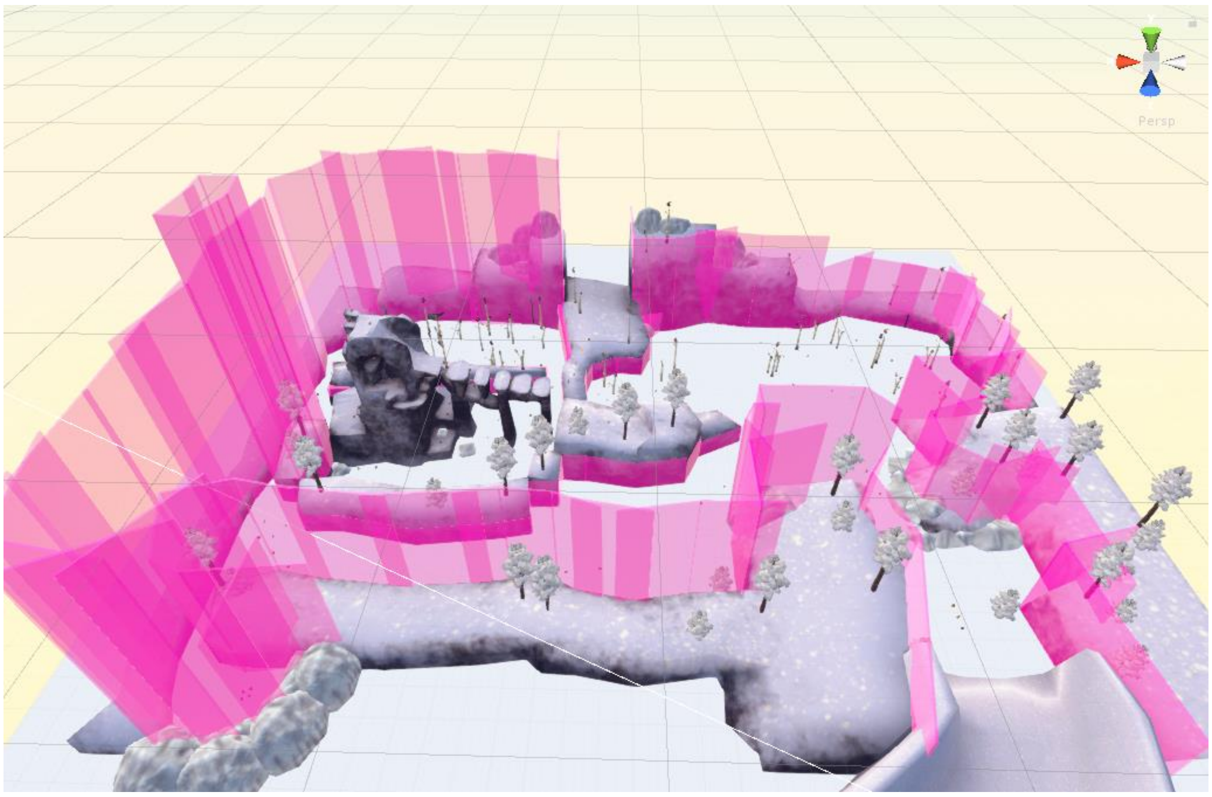
Children often behave clumsy as their fine motor skills have not been refined yet. Thinking in terms of this, we first wanted to add a delay before launching a jump, much like SSX 3 characters (EA Canada, 2003) have to wind up their jump before they actually launch into the air. By itself, this feature worked out well enough, but when actually trying to complete platforming challenges, it was just frustrating having to wait that half-second before every jump. We discarded this feature in favor of a standard instant type of jumping, like many other games have.

Anti-Wall-Jumping Mechanism

In order to prevent the player from glitching through the levels onto places where they are not supposed to be, we had to implement some mechanism. We first tried regulating how the player can behave with regards to steep walls, such as forcing them to enter the "Air" state upon landing on a very steep wall, which we defined as 45 degrees or more.

After trying to implement these restrictions into the player's movement mechanics, we realized that this was much more difficult than it sounded. Something so seemingly simple as not letting the player wall-jump requires some very advanced "cheat" detection algorithms, and making sure that the player is not allowed to jump further onto steep walls, and walk up steep walls, but are then allowed to walk down steep walls, is a battle of compromise. We had too many moments when the anti-wall-jump mechanism triggered a false positive, and many moments where it did not trigger where it should have.

Because of these struggles, we figured that it was not worth the effort to implement and balance this mechanism, so we set up invisible walls throughout the levels instead. We spent much time on building these, and they are not at all optimized, but at this point of the production, we did not have the time to implement a robust system for preventing the player from abusing geometry glitches.



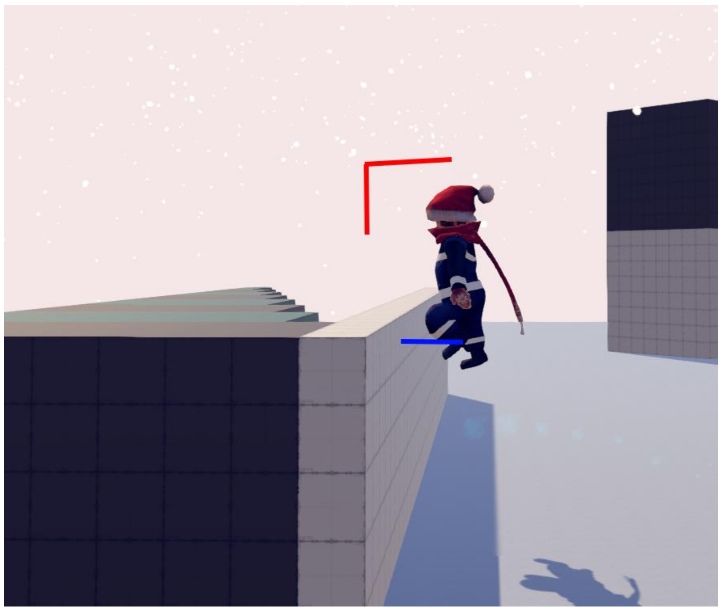
Visualization of invisible walls in level 2.

Hanging and Climbing

Our game design demanded childlike play as part of the player mechanics. Children often climb onto things, and we wanted to emulate this feature. This section will describe the algorithms used for detecting ledges and obstructive geometry when climbing.

For reference, all of the relevant code is located in the “PlayerHang.cs” script, lines 343-496 and 498-580 for the hang mechanic and the climb mechanic, respectively.

Ledge Detection Algorithm



Approximated debug lines for the hang detection algorithm (edited in Photoshop for visualization).

See the image above. In order to detect actual hangable geometry, we used several raycasts paired with the logic of our algorithm to determine whether to hang from a ledge or not. The algorithm (also found in the script called "PlayerHang.cs") can be summarized as follows:

1. Check if the player is allowed to hang.
2. Check if the head bumps into something.
3. Check if the upper forward raycast hits something (red horizontal line). If it does, that means something is blocking your possibility of grabbing a hold of the ledge.
4. Check if the upper downward raycast hits something (red vertical line). If it does, that means you have hit climbable geometry.
5. Check what the angle of the wall is, and use this to rotate the avatar accordingly.

In order to allow for the player to drop down from the ledge, we had to make sure we flag the player as not able to hang again for a brief time. We introduce a hang cooldown to circumvent this problem, so that the player can drop down by the press of a button.

The calculations involved in finding the exact hang point requires some vector projection mathematics, and we will outline the techniques we have used here. See the image for a visualization of the calculations.

1. First we retrieve the ledge's surface normal, which is gotten from step 4 in the hang algorithm above. (The red line.)

2. Then we retrieve the ledge’s wall surface normal (the vertical part), which is gotten from step 5 in the hang algorithm. (The blue line.)
3. Using these two points, we create a vector between them. (The dashed line.)
4. We project this vector onto the surface normal of the upper ledge surface (which we got in step 1 of this current algorithm). (The black solid line.)
5. The actual point the character will hang from can then be calculated to be the point where the downwards raycast hit plus the vector we just projected. (The arrow points towards the hang point).

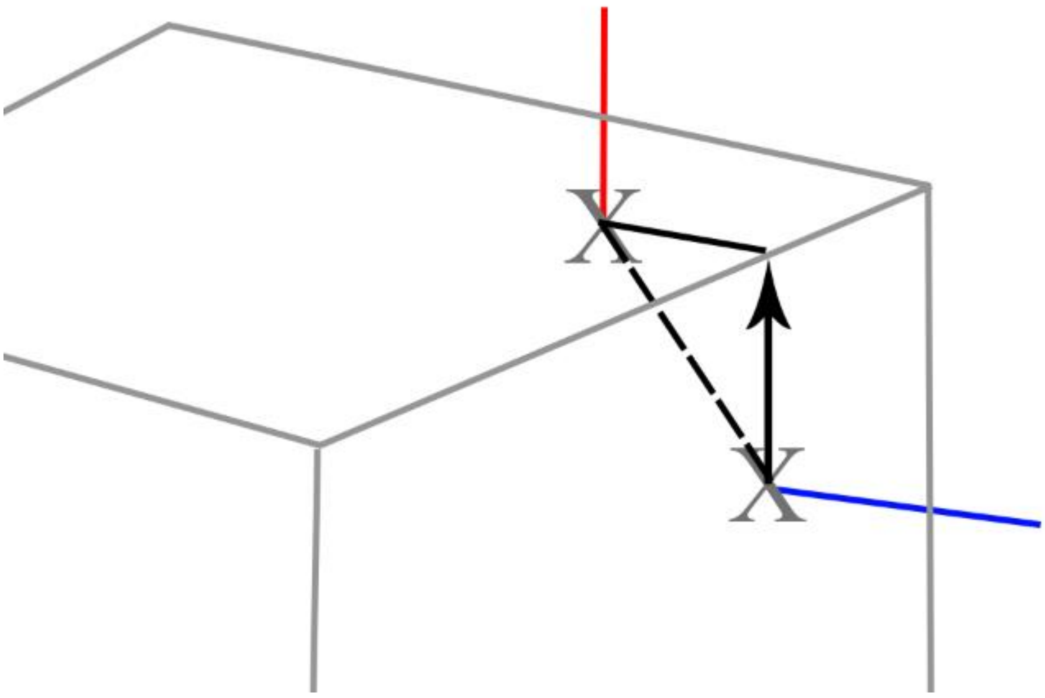
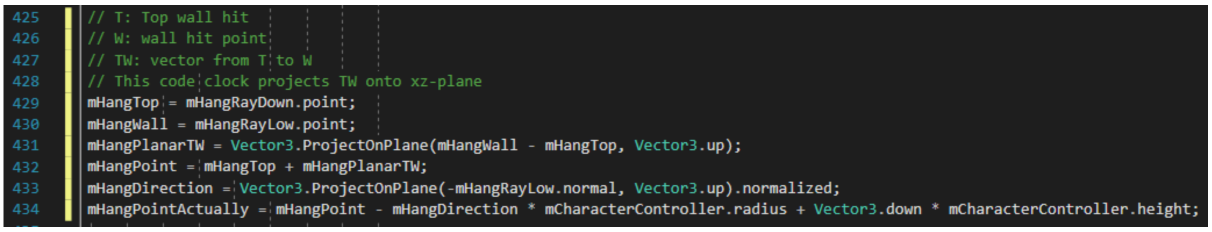


Diagram showing the calculation of the point on the edge of the mesh.

Here’s the actual code for calculating the point on the edge of the ledge:

- mHangTop is the point where the red raycast hits.
- mHangWall is the point where the blue raycast hits.
- mHangPlanarTW is the vector from mHangTop towards the edge of the mesh.
- mHangPosition is the point on the edge of the mesh.
- mHangDirection is the normal of the surface that mHangWall lies on.
- mHangPositionActually is the mHangPosition adjusted to the avatar’s capsule collider.



```
// T: Top wall hit
// W: wall hit point
// TW: vector from T to W
// This code clock projects TW onto xz-plane
mHangTop = mHangRayDown.point;
mHangWall = mHangRayLow.point;
mHangPlanarTW = Vector3.ProjectOnPlane(mHangWall - mHangTop, Vector3.up);
mHangPoint = mHangTop + mHangPlanarTW;
mHangDirection = Vector3.ProjectOnPlane(-mHangRayLow.normal, Vector3.up).normalized;
mHangPointActually = mHangPoint - mHangDirection * mCharacterController.radius + Vector3.down * mCharacterController.height;
```

We want to make a note that this hang-checking algorithm has its flaws. First, it does not take the actual hand positions into account, and can therefore cause geometry clipping. Second, it assumes that any hangable geometry has some wall connected to is that is almost completely vertical. Third, it also assumes that the point hit in the downwards raycast represents the rest of the geometry. These are some restraints we had to give our environment artist. We created a guide for our 3D modeller on how the hangable geometry can and cannot look like in Attachment #6, pages 2-6.

Obstacle Detection Prior to Climbing

Once the avatar is hanging from a ledge, we can check if it wants to actually climb up that ledge. A similar set of logic is used for this, also seen in the "PlayerHang.cs" script. This time, we take the following things into account:

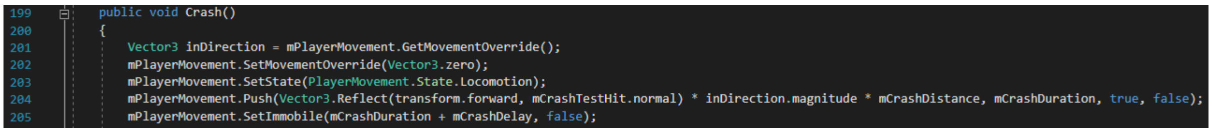
1. Is the avatar still hanging in an acceptable way?
2. Is the ledge blocked by some geometry?
3. Is the ledge actually thick enough for the player to climb onto, or would they immediately fall down after climbing?
4. Is there geometry blocking above the player's head, causing the animation to clip through it?

For a more detailed visualization of the climb checking requirements, see Attachment #6, pages 6-8.

Rolling

As part of the childlike gameplay, we figured that being able to roll would enhance this feeling of playfulness. It also serves as a means for escaping enemy attacks. We implemented a crash function, which would send the avatar back on its butt if it should roll into geometry, such as a wall or a tree. This is similar to how the roll functions in *The Legend of Zelda: Ocarina of Time* (Nintendo EAD, 1998), and it worked well until the player reaches the boss level. The boss level is a tight corridor in which the player has to dodge the Snowman's attacks. Dodging frantically in this tight corridor frequently led to the playtesters crashing into the walls, causing them to lose the boss fight. We disabled the crash function for the boss fight, and in order to maintain consistency, we also disabled it throughout the game.

Calculating the crash vector is a matter of reflecting a vector along a normal. Unity has a built-in Vector3 function called "Reflect()", which takes the incoming direction and the surface normal as inputs, and outputs the reflected vector.



```
public void Crash()
{
    Vector3 inDirection = mPlayerMovement.GetMovementOverride();
    mPlayerMovement.SetMovementOverride(Vector3.zero);
    mPlayerMovement.SetState(PlayerMovement.State.Locomotion);
    mPlayerMovement.Push(Vector3.Reflect(transform.forward, mCrashTestHit.normal) * inDirection.magnitude * mCrashDistance, mCrashDuration, true, false);
    mPlayerMovement.SetImmobile(mCrashDuration + mCrashDelay, false);
```

This code snippet shows how we reflect the incoming direction along the normal of the surface that the player crashes against, resulting in a vector that is symmetric along the surface normal, but reversed in its direction.

Throwing Snowballs

Modes of Throwing

Initially, we wanted the player to be able to throw using two different techniques:

- Free-Throw: quick and easy throw from the default camera state.
- Aimed Throw: over-the-shoulder aiming mode for precision throwing.

The level designer wanted to create some puzzles based on throwing snowballs to hit key objects. This required the player to look around to aim. As mentioned earlier, we have been designing the game from an ADM-direction, focusing on making the aesthetic experience first, and then making the mechanics to support this. We figured that having to look around to aim would enhance the graphical impressions for the player, which further justified the need for an “aimed” throwing mode.

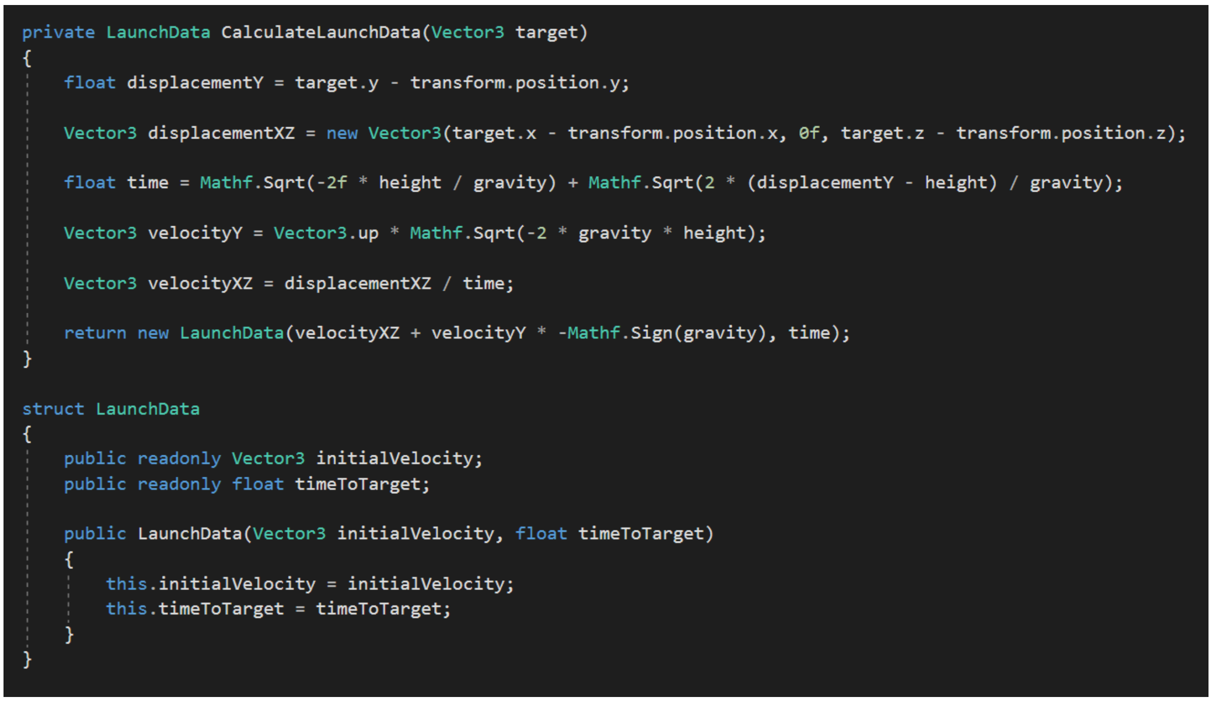
However, after working for one month on getting this feature up and running, we felt that it was contradicting the combat design of the enemies, as they were designed for the player to quickly engage in dodge-and-shoot style combat. Aiming mid-combat meant that you had to point the camera towards the moving enemy (who was often sprinting towards you), shoot, and then exit so you could dodge. Assuming you hit your target, you would have to do this over again until all the enemies have been defeated. Changing the camera angle from the default 3rd person overview to an over-the-shoulder view felt visually disturbing, as we had to make the transition quick because of the rush of combat. On top of all this, we weren’t able to comply well enough with Cinemachine’s behavior (a Unity plugin), and couldn’t program a camera that behaved predictably. We describe our relationship with Cinemachine in more detail in the “Game Mechanics” section further below.

All of these complications lead to us scrapping the aimed throwing mode in favor of the free throw, which was rather simple to implement. As a consequence, this lead to changes in level design, going from a Zelda-like puzzle setup, to a more free camera experience.

Snowball Curves

The player relies heavily on precision, to allow the player to hit the target and still allow the snowball to be thrown in a curve.
The first iteration of this calculation was simple, lerping between points to create a curve that the object could follow. Even though it served its purpose, it did not look good, and was often moving in straight lines and doing pointy turns.

In the second iteration we introduced new type of math, using kinematic equations to calculate the starting velocity of the object to reach the end point by the given gravity scale of the object.



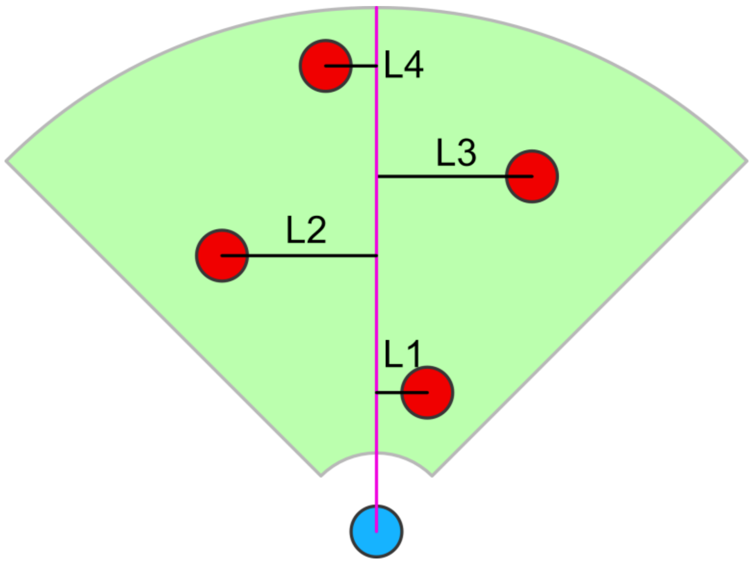
```
private LaunchData CalculateLaunchData(Vector3 target)
{
    float displacementY = target.y - transform.position.y;

    Vector3 displacementXZ = new Vector3(target.x - transform.position.x, 0f, target.z - transform.position.z);

    float time = Mathf.Sqrt(-2f * height / gravity) + Mathf.Sqrt(2 * (displacementY - height) / gravity);

    Vector3 velocityY = Vector3.up * Mathf.Sqrt(-2 * gravity * height);

    Vector3 velocityXZ = displacementXZ / time;

    return new LaunchData(velocityXZ + velocityY * -Mathf.Sign(gravity), time);
}

struct LaunchData
{
    public readonly Vector3 initialVelocity;
    public readonly float timeToTarget;

    public LaunchData(Vector3 initialVelocity, float timeToTarget)
    {
        this.initialVelocity = initialVelocity;
        this.timeToTarget = timeToTarget;
    }
}
```

Using the calculation above, we allowed the player to always hit the world position of the target at the time the snowball was released.

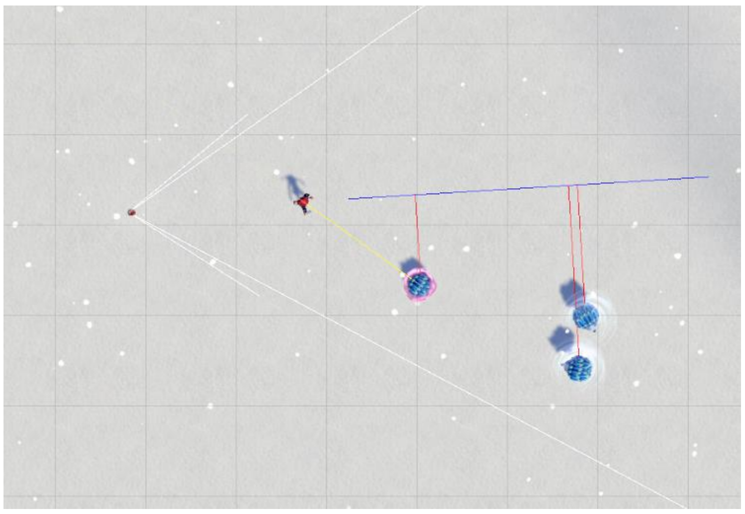
Target Selection Cone

Lobbing snowballs around based on the rotation of the player avatar was also not a success. Taking feedback from playtests, we decided to implement an auto-aim feature, which would cleverly select the target the player is aiming the most at. There is no "cone collider" in Unity, and using a static mesh would mean we could not easily tweak its values, such as range, angle, etc. In order to implement a target selection cone that is also susceptible to design changes, we decided to manually program one. This would yet again require some extensive knowledge about vector projection and vector rejection, which Eric Lengyel describes and implements in his book, *Foundations of Game Engine Development, Volume 1* (Lengyel, 2016, pp. 31-36). Our implementation will be explained in detail further below. Firstly, we present a high-level summary of the target selection algorithm:

1. Check for nearby enemies within the range of the furthest cone distance.
2. Exclude nearby enemies who are between the player and the nearest cone distance.
3. Exclude nearby enemies which are located behind the player.
4. Set up a dictionary which holds the transform of each target candidate, along with its vector pointing directly onto the player's view direction.
5. Compare these vectors, and select the target that has the least magnitude.

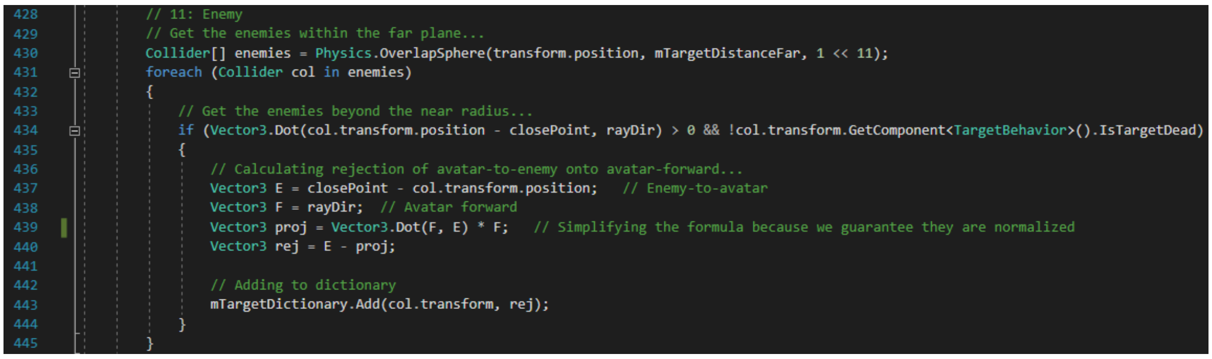


Target selection cone, showing several potential targets within the cone. This algorithm generates a rejected vector (in black) pointing from the enemy's position onto the view direction (magenta). It compares the lengths of these, and the shortest one of these will be selected.



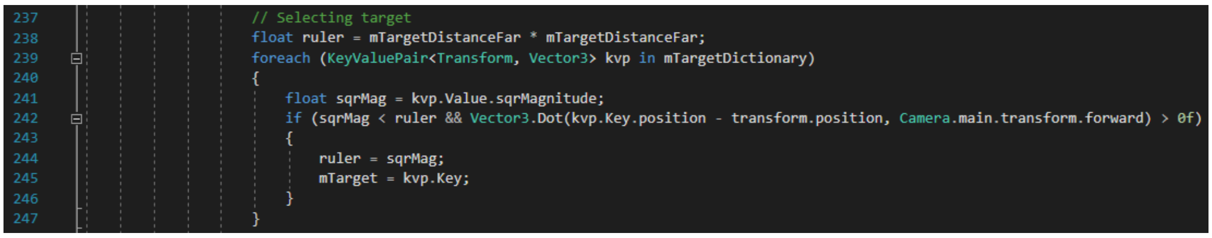
This image shows a real-time calculation of the rejected vectors in red. The blue line shows the view direction, and the pink outline on the nearest enemy indicates that it is the selected target.

Below, we show code snippets of the candidate selection algorithm, and the target selection algorithms, respectively. This code can be found in the script "PlayerThrow2.cs", lines 237-247 and 428-445.



```
// 11: Enemy
// Get the enemies within the far plane...
Collider[] enemies = Physics.OverlapSphere(transform.position, mTargetDistanceFar, 1 << 11);
foreach (Collider col in enemies)
{
    // Get the enemies beyond the near radius...
    if (Vector3.Dot(col.transform.position - closePoint, rayDir) > 0 && !col.transform.GetComponent<TargetBehavior>().IsTargetDead)
    {
        // Calculating rejection of avatar-to-enemy onto avatar-forward...
        Vector3 E = closePoint - col.transform.position;   // Enemy-to-avatar
        Vector3 F = rayDir;  // Avatar forward
        Vector3 proj = Vector3.Dot(F, E) * F;   // Simplifying the formula because we guarantee they are normalized
        Vector3 rej = E - proj;

        // Adding to dictionary
        mTargetDictionary.Add(col.transform, rej);
    }
}
```

This algorithm shows how we find the nearby target candidates. Every enemy that is found here will be assigned a rejection vector, which in the next step will be used to find the most suitable selection.



```
// Selecting target
float ruler = mTargetDistanceFar * mTargetDistanceFar;
foreach (KeyValuePair<Transform, Vector3> kvp in mTargetDictionary)
{
    float sqrMag = kvp.Value.sqrMagnitude;
    if (sqrMag < ruler && Vector3.Dot(kvp.Key.position - transform.position, Camera.main.transform.forward) > 0f)
    {
        ruler = sqrMag;
        mTarget = kvp.Key;
    }
}
```

This algorithm shows how we select the nearest enemy based on their attached rejection vectors.

Bum Slider

One of our key selling points is the ability to sled on your bum slider. We wanted the bum slider to serve mainly as a fun addition to the game, without too much emphasis on training the player to become a good bum slider, because of time constraints in the level design. As such, we implemented its use mainly at three separate points in the levels, although the player can choose to use it whenever they would like.

The bum slider was possibly the most difficult player mechanic to implement, much because we were using a CharacterController instead of a RigidBody, meaning we had to keep track of every force acting upon the player. These forces include gravity, surface-based velocity, input velocity, and any potential overriding velocity, such as when being pushed. We started implementing the bum slider mechanic quite late during production, and we could not implement a RigidBody solution for this mechanic, so we had to build upon what we already had. For reference, see the script called "PlayerMovement.cs", lines 1080-1278 for the actual movement implementation, and the script "PlayerAutoSlide.cs" for the state transition logic.

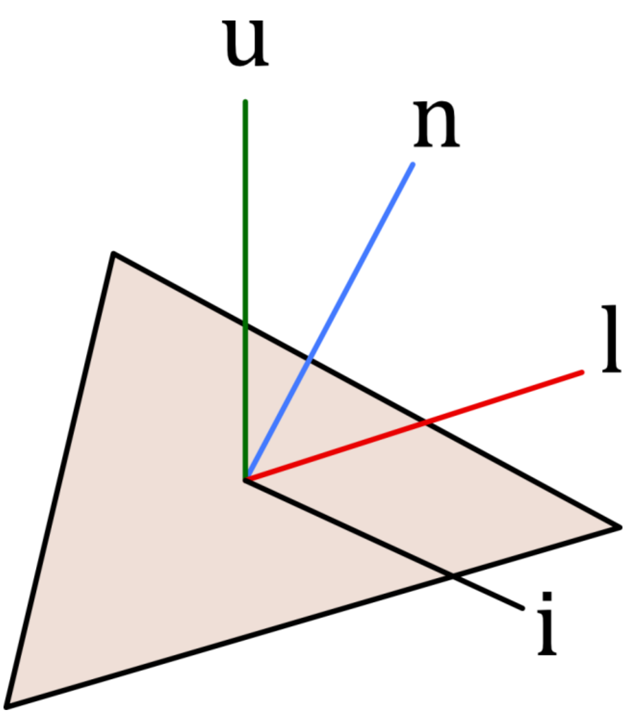
Surface Movement

In order to slide along the current surface, we read the currently inhabited triangle's normal from a basic downwards raycast, and use this as a basis for calculating the current planar velocity. Using this method, we have to keep track of the velocity vector from the previous frame, store its magnitude, and project it onto the current triangle. This projection will likely cause an unnatural loss of magnitude, as often happens with vector projection. That is why we stored the previous magnitude, and we then normalize the newly calculated vector, and then multiply it with the previous magnitude vector.

Player Input Restrictions

Using only the method described above, we will never see a change in magnitude. That is why we let the player's input affect how much the speed should increase or decrease. For this, we impose restrictions such that the player cannot slide up slopes, unless they have built up velocity that will take them in an upwards direction. Whether the player is going forwards or backwards or sideways, is visually obvious to the player. But in order to calculate it in the script, we need to take the surface normal, the stick input, and the camera's orientation into account as follows:

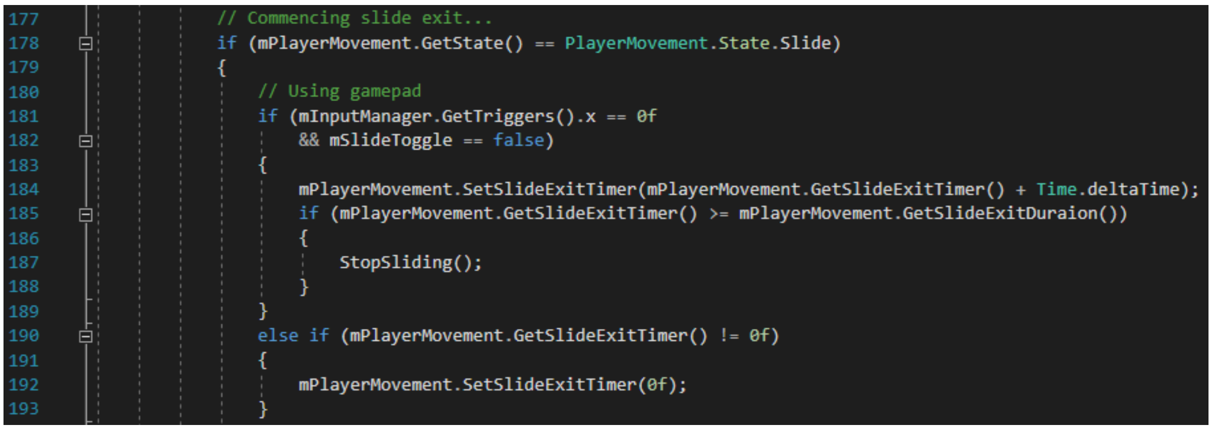
1. Store the current surface normal by raycasting downwards.
2. Calculate the left stick input based on the camera direction by multiplying the camera's quaternion with the left stick input (represented as a Vector3).
3. Calculate a vector that lies along the current surface by taking the cross product of the surface normal and the world's upward vector. This vector will always point to the left when looking down the triangle as if about to walk down it. Because Unity uses a left-handed coordinate system, and because the cross product is non-commutative, the first factor in this operation is the surface normal, followed by the world up-axis. See the illustration below. (We would like to note that because we are using this leftwards vector as the base for our vector projection, it does not matter whether it points to the right or left, because that information will be lost in the projected vector.)
4. Take the dot product between the input vector and the world up-axis to determine if the player is steering the control stick in the upwards direction. If they do, project their input onto the "l" vector so they can maximally reach a 90 -degree sideways movement. This disallows for sliding backwards when there is no apparent force letting you do so.



Calculating the leftwards vector, “l”, using the surface normal, “n”, and world up-axis, “u”. One example of a resulting input vector is shown as “i”. This vector is then used to correctly set the velocity through the CharacterController’s Move() function.

Exit Conditions

In a previous build, we implemented an automatic exit condition, letting players exit the bum slider once they reach low enough speeds at a low enough surface angle. We would then expect the player to manually brake with the bum slider in order to trigger this behavior. This turned out to feel inconsistent, as the player experienced different speeds throughout their bum sliding gameplay. In order to counter this, we introduced the ability to exit the bum slider in a fixed amount of time, slowing down linearly (based on mSlideExitTimer) while preparing for the exit condition. This is implemented in the “PlayerAutoSlide.cs” script in lines 177-193, as seen below:



```
177  // Commencing slide exit...
178  if (mPlayerMovement.GetState() == PlayerMovement.State.Slide)
179  {
180      // Using gamepad
181      if (mInputManager.GetTriggers().x == 0f
182          && mSlideToggle == false)
183      {
184          mPlayerMovement.SetSlideExitTimer(mPlayerMovement.GetSlideExitTimer() + Time.deltaTime);
185          if (mPlayerMovement.GetSlideExitTimer() >= mPlayerMovement.GetSlideExitDuraion())
186          {
187              StopSliding();
188          }
189      }
190      else if (mPlayerMovement.GetSlideExitTimer() != 0f)
191      {
192          mPlayerMovement.SetSlideExitTimer(0f);
193      }
```

Camera

For our camera gameplay, we decided early on to use Cinemachine, an official Unity plugin. It has a built-in feature for 3rd-person free look-style camera controls, which is exactly what we needed for our game. Using this plugin, we massively cut down the time needed to program camera controls from scratch, but it also introduced some problems that arose from our inexperience with it. For example, it was one of the main reasons we had to cut the over-the-shoulder style of throwing, because we could not get Cinemachine to work the way we needed it to. Seemingly simple tasks such as directly setting the yaw rotation around its target, were not an option while using Cinemachine's FreeLook component.

Enemy Mechanics

Taking a lot of inspiration from older 3D action-adventure games like *The Legend of Zelda: Ocarina of Time* (Nintendo EAD, 1998) we wanted to have similar difficulty in our combat system. Making it easy to understand, and give enough challenge for it to be fun. As mentioned earlier under avatar mechanics we have two major abilities for combat use: the first one being the roll, allowing the player to roll out of danger when presented to that type of challenge; the second one being throwing snowballs, which is how the player defeats the enemies. Due to us having limited time, and short amount of combat abilities, we had to make sure our design was easy to create, and would be consistent with the player mechanics.

Enemies

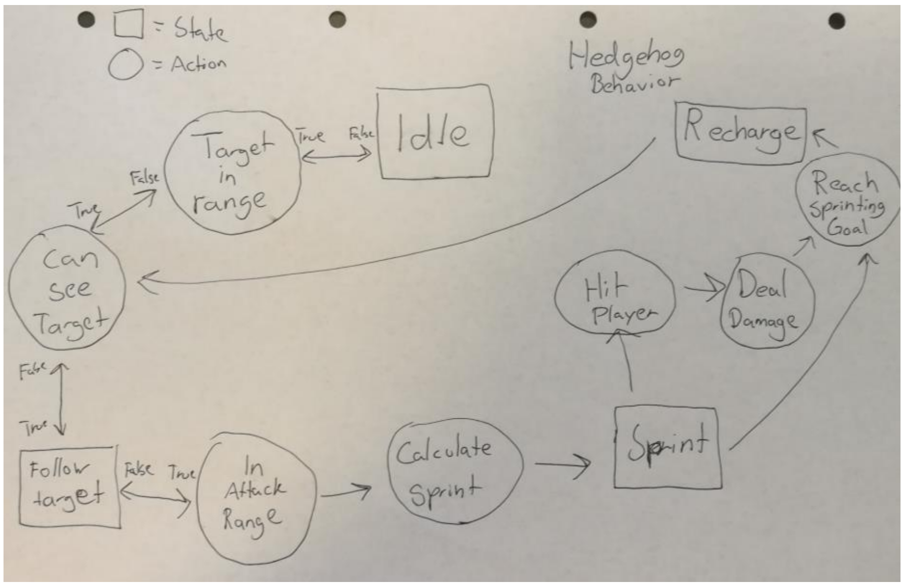
Creating the enemies with different stances, allowed them to be easily adjusted in how they would respond to the player. The important part of our designs was to create enemies that felt unique. Whenever you encountered a new type of enemy, it would feel similar in ways of gameplay, while the aesthetic design features made them look and feel special.

In early iterations of the gameflow we had mismatch between a slow aiming system from the player's perspective, with a fast paced rhythm to the enemies. We initially decided to keep the faster paced gameplay, and started reworking how the player would react to the enemies. See section Avatar mechanics (Throwing mechanic), for further information.

Hedgehog

They have much resemblance to a regular hedgehog, but with icicles spikes on their back, and they charge aggressively towards their foes. Their general design was to be easy to tackle in normal situations, but would soon become troublesome if there were multiple hedgehogs, or if surprising you from an unexpected position. They consist of straightforward design patterns, allowing the player to quickly learn the behavior, and find ways to deal with them.

In the original design of the hedgehog would run towards the player in a straight line, and exploding when close enough. If it missed the player, it would continue to sprint forward, and exploding on impact with the environment. Due to difficulties in registering the unpredictable world geometry, and having a hard time giving audio visual cues when the camera was facing away, we had to redesign the hedgehog to better fit the scope of the project. The new design was to give it a simple and wider visual cue while sprinting, and dealing damage on impact. Then continue the sprint, but instead of exploding it would recharge its strength and try again once fully charged.



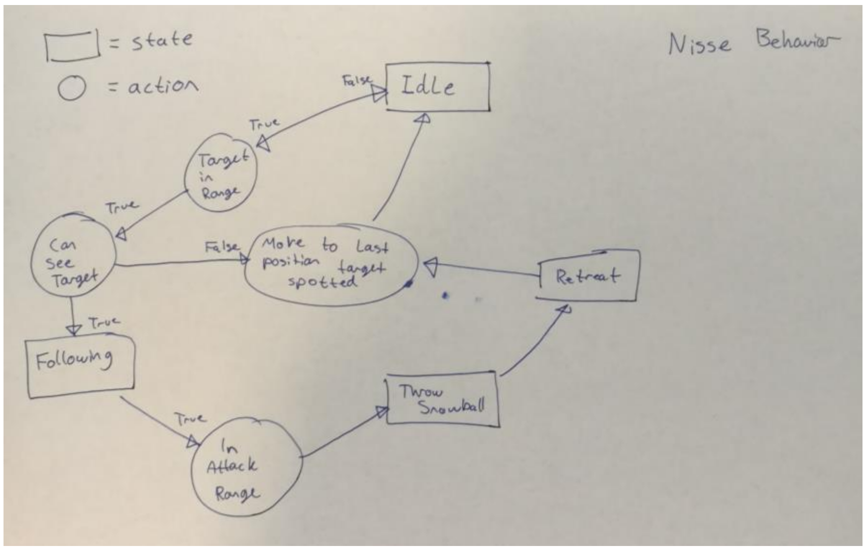
As displayed in the picture above, the linear approach is easy to understand once it has been experienced, there is never more than two possible outcomes. Despite of it being simple, the built in reaction time to avoid the sprint, and the window of opportunity to deal damage to the hedgehog during the recharge.
This is allowing the player to find challenge in their approach. Focusing on simplified difficulty, rather than big over complicated systems.

Nisse

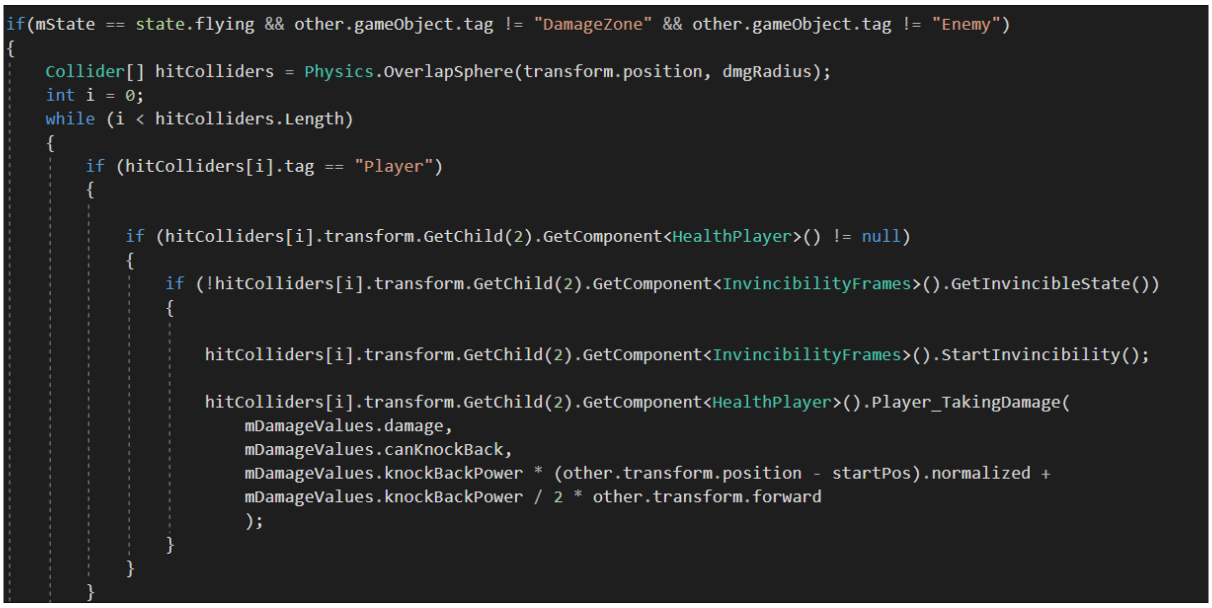
After the hedgehog came into play, we soon felt the need for adding enemies with more contrast. This led to the nisse, a vile rural creature running around, and annoying anyone in its path. The main concept was to create a ranged enemy which would force the player to reposition itself to better handle this new threat. This is where their ability to throw objects at the player came into play, but we found constraints with the character design. Throwing objects was difficult with small arms, and felt unnatural for such a small creature. That is when the new design of the throw came into play, they would now lift objects with magic, and could easily pick up snowballs with the same size as themselves. This also juiced up the nisse design, while solving the issue we had with not giving the player enough visual cues when the nisse was going to attack.

To show of the personality of the nisse even further, we added a retreat after attacking the player. The nisse is considered to be playful, and often spend time running back and forward between attacks to mimic the feeling of playing with the player.

In order to to make the throw feel realistic we had to implement a system that would make it feel both smooth and trustworthy. Lucky for us we had already created this calculation in the avatar mechanics, and could easily reuse code to save time.



During TestBonanza #16 (see Attachment #12 - Playtest Documentation 20.04.2018), we discovered that it was to easy to dodge the snowballs for the player. This lead us to creating an explosion upon impact, allowing the snowballs to be more predictable, while also making them more difficult to avoid.



```
if(mState == state.flying && other.gameObject.tag != "DamageZone" && other.gameObject.tag != "Enemy")
{
    Collider[] hitColliders = Physics.OverlapSphere(transform.position, dmgRadius);
    int i = 0;
    while (i < hitColliders.Length)
    {
        if (hitColliders[i].tag == "Player")
        {

            if (hitColliders[i].transform.GetChild(2).GetComponent<HealthPlayer>() != null)
            {
                if (!hitColliders[i].transform.GetChild(2).GetComponent<InvincibilityFrames>().GetInvincibleState())
                {

                    hitColliders[i].transform.GetChild(2).GetComponent<InvincibilityFrames>().StartInvincibility();

                    hitColliders[i].transform.GetChild(2).GetComponent<HealthPlayer>().Player_TakingDamage(
                        mDamageValues.damage,
                        mDamageValues.canKnockBack,
                        mDamageValues.knockBackPower * (other.transform.position - startPos).normalized +
                        mDamageValues.knockBackPower / 2 * other.transform.forward
                        );
                }
            }
        }
```

As shown above, we used a simple Unity function called Physics.OverlapSphere, which spawns a invisible collision sphere collecting all the GameObjects hit within its radius. Adding this to an array, and then running through the array in the same frame to check for a player allowed it to be quick and functional. In retrospect it would be fast and better to add a layer check for storing the GameObjects, but we did not prioritize performance in these calculations much due to not noticing big changes in the profiler (Unity Technologies, 2018), and because we have experienced troubles with layer-masking (Unity Technologies, 2018) in builds.

Snehetta

To create some contrast to the puzzles we wanted to add a last obstacle preparing you for the snowman encounter, this was the fight with Snehetta. Snehetta is a pile of snow, brought to life by the snowman. She was also the mother of hedgehogs, spawning them and issuing commands of their master.

During the design process of Snehetta, we came up with different approaches on what type of abilities she would have. The first ability, that was consistent throughout her design, was spawning hedgehogs during her encounter, but after removing the explosions mechanic from the hedgehogs the fight started to become dull. Therefore, we start discussing what we could add to improve the fight, in the end we decided on a breath covering the room. The breath could only be dodged by hiding behind nearby rocks.

The first iteration spawning got implemented, and went by without to many issues. Snehetta's breath on the other hand became troublesome, early versions use a lot of collision boxes, and did not utilize the particle system (Unity Technologies, 2018) correctly. Even though it held a steady frame rate on our development laptops, we could spot problems on weaker graphic cards, therefore we had to go back and rethink our approach.

By using utilizing the particle system correctly, it could not only reduce the GPU usage, but also handle collision detection. This gave us the opportunity to create a both consistent and visually pleasing result, while averting 20-30 frame drop on a high end computer.

A little later into the development we had a meeting with our external mentor Trond Fasteraune regarding the current state of the game. Here we discussed the direction of our game, and how our current scope would fit into the remaining development process. He came with a lot of good feedback, the most valuable feedback he gave us was to scale down our current plans. He wanted us to focus on creating a smaller level, and deleting all that was unnecessary to show the flow of the game.

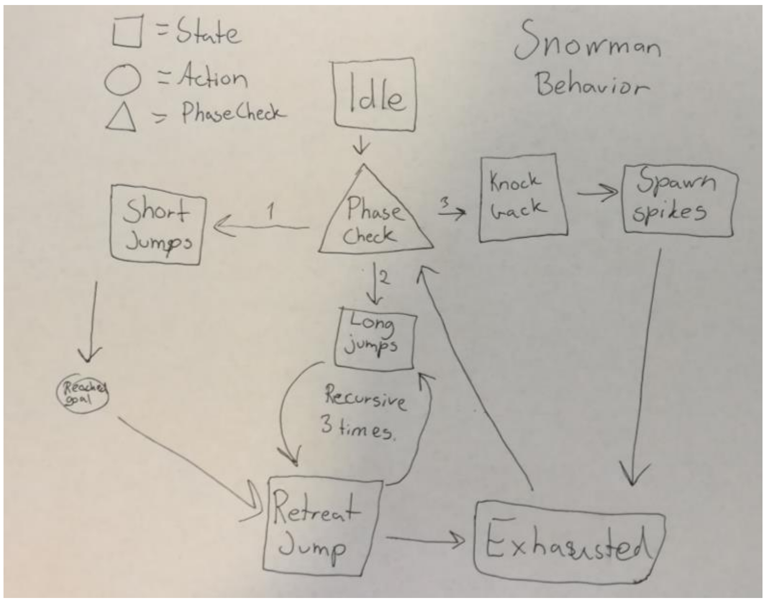
After receiving the feedback we had a meeting with the entire team, and start to discuss what to cut. In this discussion everyone was in favor for keeping the snowman, and scratching Snehetta from the game. This resulted in Snehetta being cut, and is no longer part of the game, much of this due was because of the simplicity of the current version, and Snehetta had no actual narrative interaction with the player. To see the different assets used for Snehetta go to (Tech)NeonPeon/NordavindGame/Assets/Enemies/snehetta.

Boss Fight

Snowman

The design of the snowman was meant to challenge while keeping what the player have learned throughout the game in mind. All the attacks were designed around dodging, and dealing damage by finding the perfect opportunity to attack. Having these design decisions in mind we created a three phase boss fight.

Before we have a look at each phase of the boss, we need to look at the general design.



As seen above we focus on making the different phases recognizable. The main thoughts around this was to have the player understand how to deal with the boss in each phase after mastering the first one, even though the abilities and challenges differ. The difficulty of the boss was meant to challenge the player until they find a way to defeat each phase, but not punish the player mindlessly. Having these predetermined design choices in mind we created three different abilities that worked in some way similar:

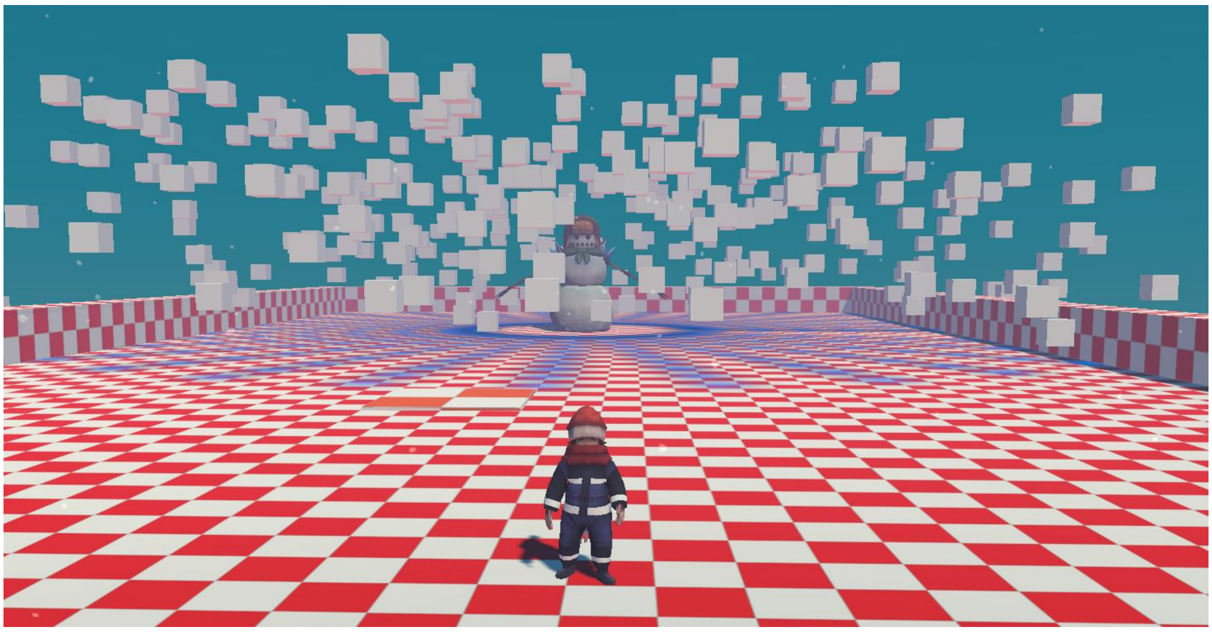
The first one being a series of jump attacks towards a set goal. We called this attack thwomping. Thwomping is meant to be an easy to learn ability to learn the player how to understand the boss, and how the boss interacts with the player. Being the first ability, we prefered having an ability relying more on understanding the fight, instead of depending on the players reaction time.

Second we have the jump attack. The jump attack is quite similar to thwomping, but instead of having a jump towards a world position. It jumps towards the player position instead, producing a scenario where the player both have to react and respond. Having already learned how to tackle thwomping, reacting to the jump attack should be fairly straightforward.

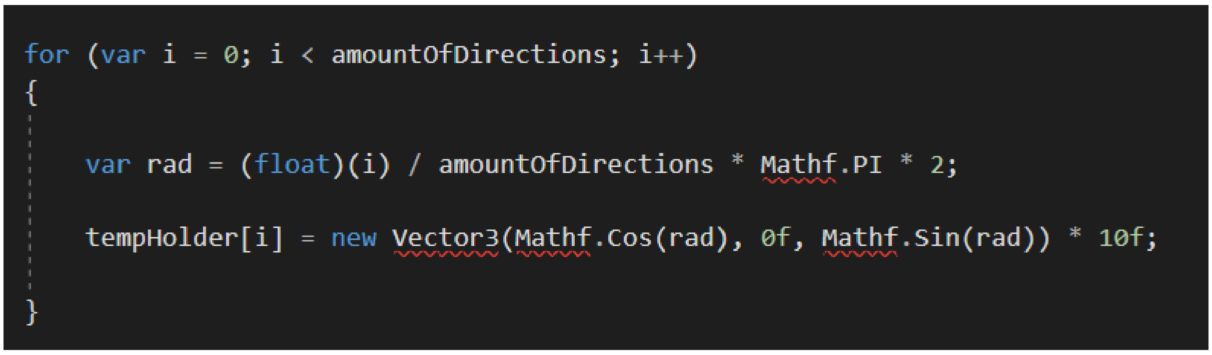
For the two first abilities we rely on physics, and precision. Therefore we had to figure out a way to calculate the jump-curve, while having both height and speed of the jump easily adjustable. This was a perfect opportunity to reuse code. Having already implemented an kinematic formula to calculate starting velocity to reach a target in the player throw mechanic.

The third ability is the summoning of spikes. They are the one ability that have undergone the most changes throughout the production.

At the start, the snowman was designed to have a barrage of snow coming at the player. It would function as a blast coming from the snowman releasing particles landing around him dealing damage on impact. Which can be seen on the picture below:

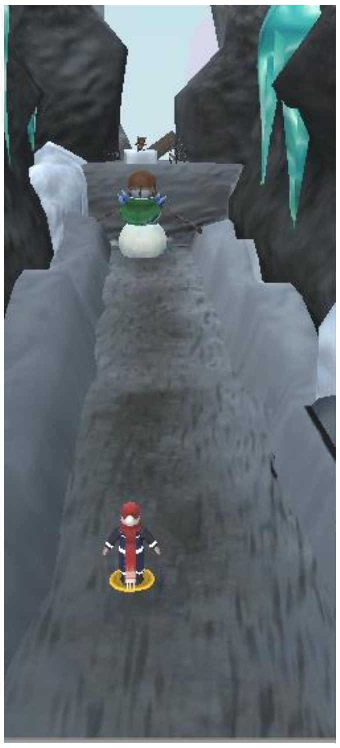


By using the calculations used in phase one and two, and by using trigonometry to calculate the directions we managed to get the result above. The functionality was there, but it was yet to be tested in the actual boss environment.



```
for (var i = 0; i < amountOfDirections; i++)
{

    var rad = (float)(i) / amountOfDirections * Mathf.PI * 2;

    tempHolder[i] = new Vector3(Mathf.Cos(rad), 0f, Mathf.Sin(rad)) * 10f;

}
```

Trigonometric calculation of particles' positions.



If we look at the picture to the left we can see that in room you are fighting the boss is a lot more narrow than the level used to previously test the barrage.
This lead to difficulties in two ways;

- The barrage was depending on gaining a high altitude before falling down, which left little to no visual cues of where the snow chunks were at all time.
- We would have to rewrite the pattern in which the snow chunks would land, to make the dodging more difficult.

Due to these difficulties, and trying save as much time as we could, instead of trying to hold on to an ability which do not fit the environment design, we decided to recreate the ability and give it another way to interact with the player.

Previously we had discussed the ability for the snowman to roll snowballs towards the player, having to dodge left and right to avoid getting hit. We went back on this idea, but ended up having spikes spawning in a line moving in a forward direction. The design was simple, and the implementation did not require a lot of work. Due to it relying on having collision detection, all we had to do was to use a rigidbody and set the standard velocity towards the direction we want the spikes to go.

After a couple tests, we understood that the general concept of the design was good, but the difficulty had to improve. Most players stood idly by the boss while waiting for an opportunity to attack, or waiting for hazards to spawn. Therefore we added two new features to the ability:

1. Every time the boss goes out of tiered state the player would be knocked back to the back of the room. This forces the player to run all the way back to the boss, and give them another thing to do while waiting for the vulnerable opportunity.
2. The second thing we added was a wall of snow forcing the player to break habit and not just go left and right. This gave the fight some contrast, and was meant to make the ability more engaging.

How is the snowman now?

In retrospect, we lost track of time, and while the first two abilities have been in the same state for quite some time the third ability went through many iterations to late in the production. This have lead to the third phase being buggy, and giving some wrong visual cues. While it is playable, the knockback is a bit off with the rest of the design. Much of this is due to the lack of modularity in the bridge between code, animations, and cutscenes. Although code and animations were originally well matched, the addition with cutscenes broke the structure and made the design process difficult.

Audiovisuals

Early in the production it was decided to minimize the amount of GUI and visual guiding systems that normally would make it easier for the player to understand what is happening on the screen. This was thought of to bring more attention to the world and the character, instead of a piece of GUI. This led to a lot of different approaches to how we would help the players understand their surroundings, while still holding on to our core idea. Therefor we had to use the tools at our disposal, which included shaders, animations, SFX, and cutscenes.

GUI

Even though we decided early on to minimize the amount of GUI, we still had to use it to certain parts of our game. Some examples include tutorial prompts, collectibles count, and the menu system.

In our tutorial level, we need to teach the player to use the rolling function, as well as the bum slider, and the throwing mechanic. As these mechanics' control mapping are not intuitive, we were compelled to give tutorial prompts in the form of signposts explaining the game, much like in *The Legend of Zelda: Ocarina of Time* (Nintendo EAD, 1998). This also makes the tutorial optional for those who already know the controls.

Shaders

Plugin: Shader Forge

During our education so far, we have touched on the topic of shader programming, but never really gotten very deep. One of Dexter André's skill-wise requirements for the project was to be able to implement some advanced shader techniques. Being familiar only with GLSL from our graphics programming courses, we had to make some adjustments to Unity's ShaderLab language (an extension of NVIDIA's Cg shader language). In order to save time on specializing in this new language, we used the Shader Forge plugin for Unity, which is a node-based shader workflow. Note that our project was made in Unity's 2017.3 version, which does not feature its newly released Shader Graph package.

Using Shader Forge let us focus more on the visual and mathematical understanding, rather than the esoteric ShaderLab language.

Show Deformation

We wanted the player to feel immersed into the wintery world we have created, and one way of doing this is to let the player avatar deform the snow as it walks on it. We thought the best way to do this was through a shader, since no physics or raycasting needs to be directly involved with the shape of the snow. Everything related to the ground is still calculated based on how the mesh looks before deforming it in this shader. For reference, see the script "SnowDeform.cs" for the texture painting part of this feature, and "SnowDeformCopy.shader" for the actual shader code. Please refer to Attachment #8: Showcase of Snow Deformation Shader for a video showing this shader at work.

Displacement Map

In order to communicate to the world that the player has stepped at a certain position in world space, we use a raycast to retrieve the current ground plane's UV map position. We then use a dedicated script to access a special displacement map which we call the Splat Map. In this script, we draw onto a RenderTexture using a custom black brush, depressing the snow wherever there is black on that splat map. This script is attached to each object that also has a material called "SnowDeform". In order to allow for several actors to paint on the snow simultaneously without overwriting each other, we collect all relevant Transforms and run a loop every frame to collect their drawing data.

All the code related to texture painting was inspired by a tutorial by Patreon user MinionsArt (2017).

Texture Blending

By default, the snow looks just like a standard snow shader. But when depressed, there should be a visual difference in the snow, or else it would look like nothing really had happened when walking in the snow. We blend between the two modes, using one set of textures for the "upper" appearance, and one for the "lower" appearance. This highlights the effect of stepping in the snow.

Tessellation

In order to yield a convincing result, we had to increase the resolution of the mesh by at least 10 subdivisions from what it actually is. Fortunately, Unity has built-in functionality for dynamically subdividing the mesh inside the vertex shader program. This process is called tessellation, and a guide can be found at Unity's manual (Unity Technologies, 2018).

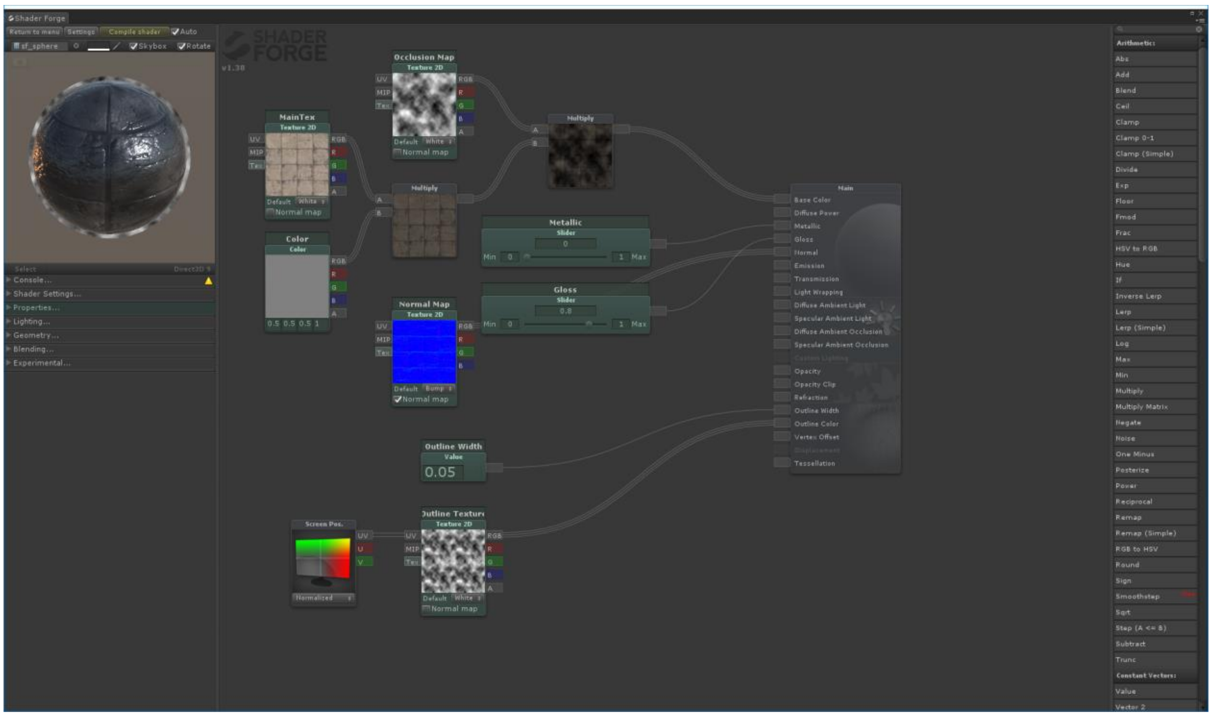
Wooly Outline

To add to the credibility of the wool materials, we decided to add a fibery effect, as commonly seen in wool fabric (see image below). This effect is most prominent when looking at the silhouette of such materials, so we decided to make an outline shader for this. For reference, this shader is called “WoolOutline.shader”, and was written mostly in Shader Forge, so it is quite messy, but functional.

Wooly outline.

Simple Outline Prototype

Implementing simple outlines in Shader Forge is straightforward - there is an “Outline Width” and “Outline Color” handle, and for simple toon-style outlines, that would be good enough.



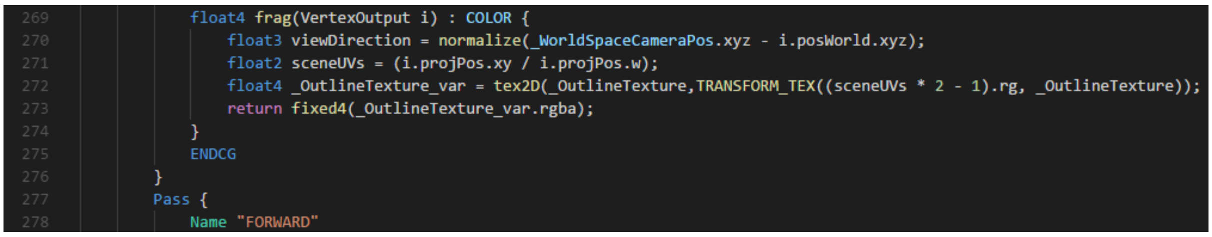
Outline shader made in Shader Forge. Note that the preview image in the top-left corner does not portray any issues that would occur in a scene with other geometry.

Additionally, we wanted the wool outline to have a texture that always faced the screen, so as to keep the same quality even if the triangles were not directly facing the camera. The red, green, and yellow node in the bottom-left of the image above lets us map the texture using screen-space UV coordinates, instead of the model's UV coordinates.

Transparency Issues

But our goal was to add a wooly outline that appears around any wool material, which has thin strands of fabric sometimes poking out of the surface. These are not displayed in solid color, but rather as something that is transparent at times, and other times solid color.

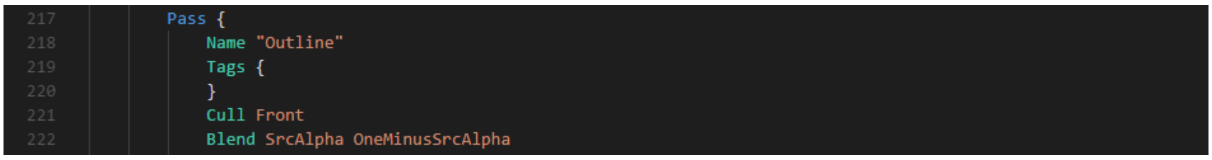
In order to add transparency to the wool material already created by Shader Forge, we had to tweak the fragment shader of the outline pass to utilize the alpha channel.



```
269             float4 frag(VertexOutput i) : COLOR {
270                 float3 viewDirection = normalize(_WorldSpaceCameraPos.xyz - i.posWorld.xyz);
271                 float2 sceneUVs = (i.projPos.xy / i.projPos.w);
272                 float4 _OutlineTexture_var = tex2D(_OutlineTexture,TRANSFORM_TEX((sceneUVs * 2 - 1).rg, _OutlineTexture));
273                 return fixed4(_OutlineTexture_var.rgba);
274             }
275             ENDCG
276         }
277         Pass {
278             Name "FORWARD"
```

Line 273: using alpha for the outline.

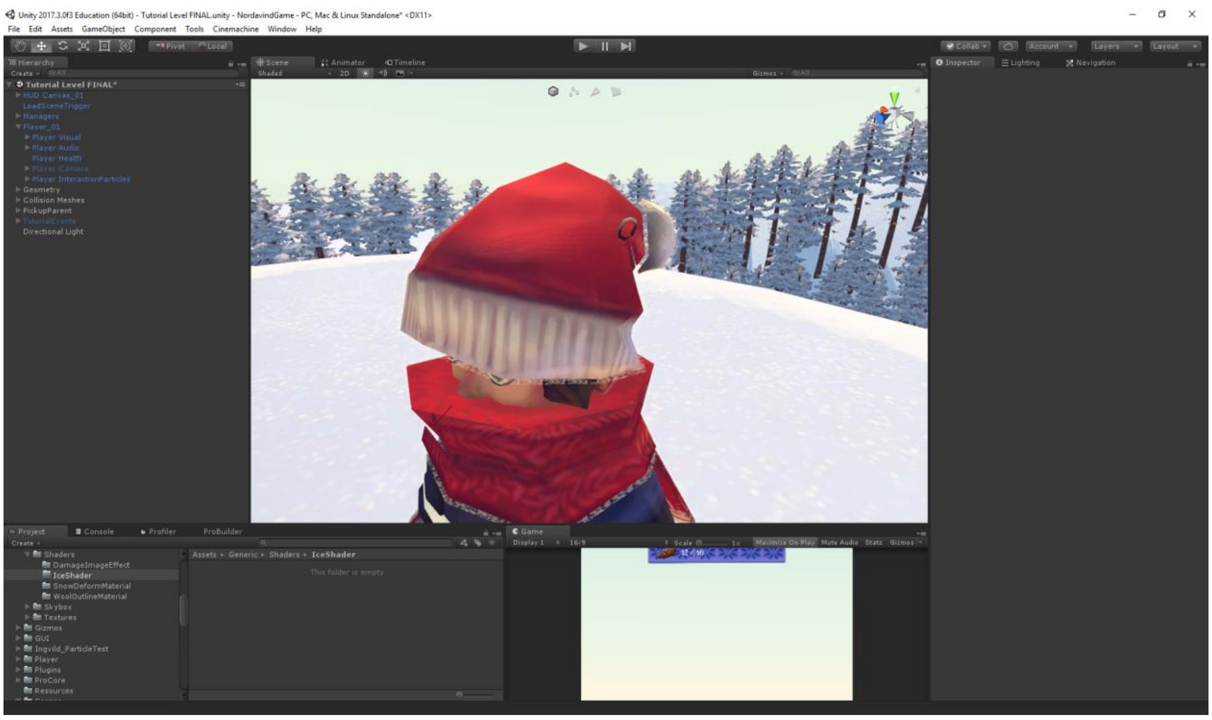
We also had to make sure that the blend mode was set to "SrcAlpha", "OneMinusSrcAlpha".



```
217         Pass {
218             Name "Outline"
219             Tags {
220             }
221             Cull Front
222             Blend SrcAlpha OneMinusSrcAlpha
```

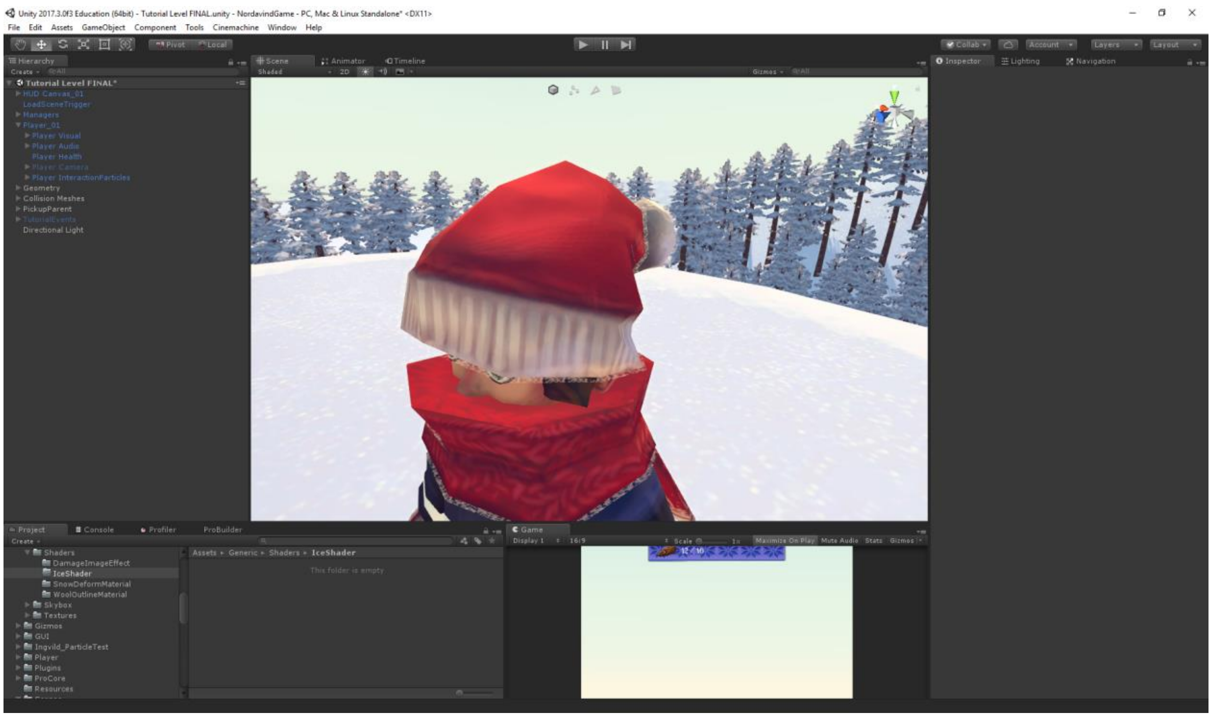
Line 222: enabling alpha blending.

Another problem arose because of this: the outline would render transparently through its own geometry, and render parts of the background environment in its place.


Renders background through the outline. Most apparent on the transition between the hat and the hat tassel, and also on the folds on the front of the scarf.

The Shader Forge documentation mentions that the outline is created by displacing each vertex along its own normal, and then flipping the normal direction so that the faces won't display from the back due to Unity's backface culling. This will render an outline which appears behind the actual model, and not in front of it (unless there are protrusions that would also cause an outline, like the folds of the scarf, which is intentional).

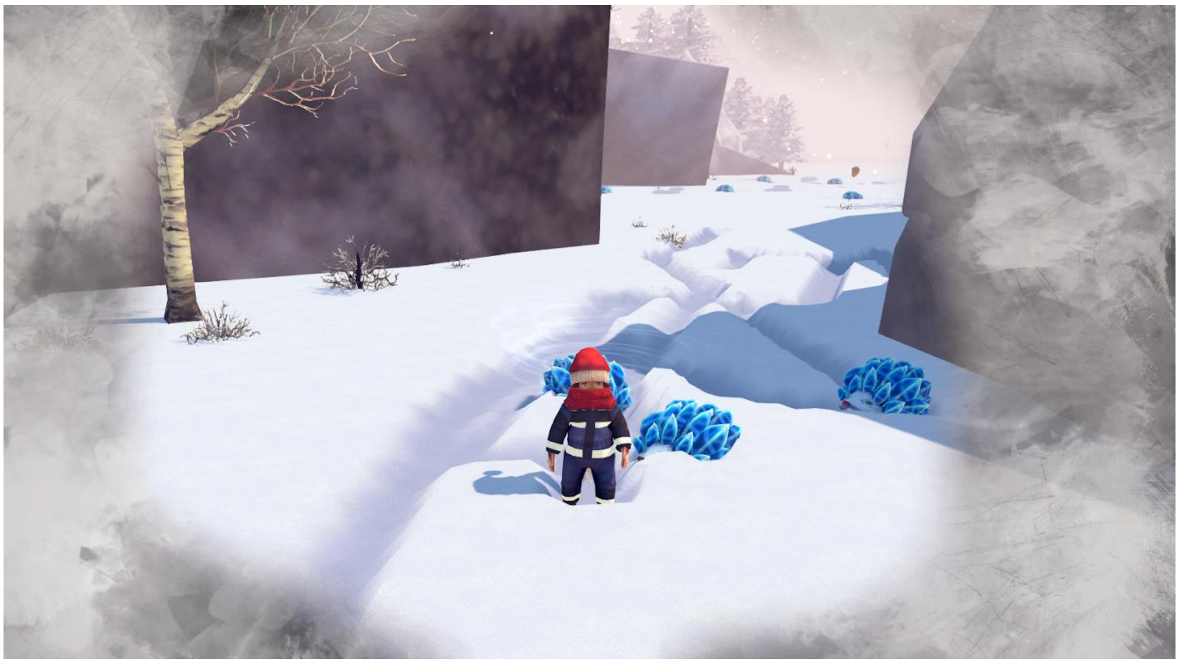
Knowing this, we deducted that we could make one more render pass before the outline appears, so that the area behind the transparent parts of the outline would be rendered as itself, instead of as the surrounding geometry. This entire shader requires three passes, which is not ideal, but we thought it to be adequate for our game, seeing as it does not feature too detailed meshes.


Finished wool outline.

Damage Image Effect

Because of our “as little GUI as possible”-rule, we needed to convey the loss of health in a subtle manner. We thought to make a frost-like effect appear on the camera whenever the player takes damage. We did this by sending a material to the camera which would then render parts of the original image, but with the frost effect overlaid. Our player avatar has four stages of health: full health, medium health, low health, and dead. We only use the effect for the three upper health classes, seeing as we do not really need the effect when the player has died.

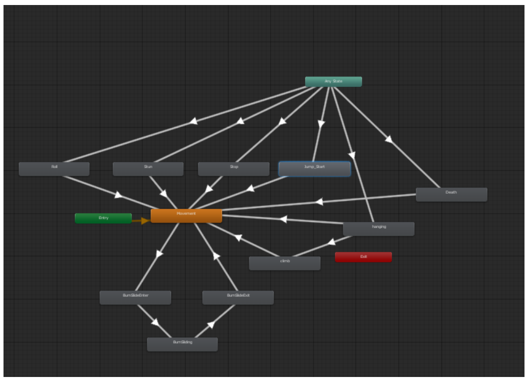
Our script, “DamageImageEffect.cs”, determines the health of the player, and then sends this information into our screen effect shader, which is called “DamageImageEffect.shader”. This shader functionality for increasing and decreasing health, transitioning between them, and masking the frost image effect according to the amount of health left.


Damage image effect at work.

Animations

In any game, animations contribute to the immersion of the player. Adams discuss VandenBerghe's five domains of play (Adams, 2014, pp. 82). The one corresponding to our game setting is novelty, which Adams refer to as the openness to experience (Adams, 2014, pp.82). Will our game feel familiar for our player base, and is there a lot of unexpected occurrences?

Our game is based on a real life setting, rural Norway. The setting will feel familiar in the western world, due to similar nature and the snowy weather. While people from around ecuador might experience snow, the actions you do in the snow are different. Early in the production we decided to enhance the feel of being a part of the world for the player, and to do this we relied heavily on animations to bridge the connection between the player and the screen.



Animations would help us giving the player visual feedback and make the world more believable. We will talk about the different technologies used within Unity (Unity Technologies, 2018), and how we created code structures to work well with both this technology and the other systems it will be communicating with.

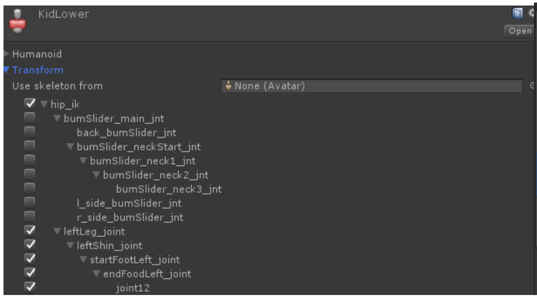
When working with animations in Unity, the majority of the work you do is with

animation trees within Unity's animation system (Unity Technologies, 2018), formerly known as mecanim.
This allowed us to work with different technology that would simplify the pipeline for creating animations. Mecanim allow the user to create blend trees, make animation events, and create blending transitions between animations while you are playing your animations.

Avatar masking

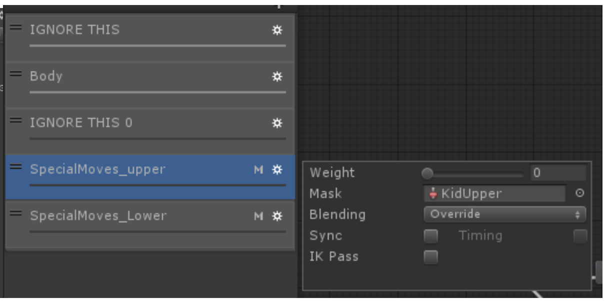
In earlier version of the game we had planned using the scarf around the main characters neck as a living item. It would throw snowballs, and work on its own. Our animator had a lot of troubles, having to animate them as two different models, and we decided to find a better solution.

We came across something called the Avatar Mask (Unity technologies, 2018). The avatar mask allowed us to divide different part of a rig into different animation layers. Shown in the pictures below:


KidLower

Humanoid
Transform
Use skeleton from: None (Avatar)

- [✓] hip_ik
  - [ ] bumSlider_main_jnt
    - [ ] back_bumSlider_jnt
    - [ ] bumSlider_neckStart_jnt
      - [ ] bumSlider_neck1_jnt
        - [ ] bumSlider_neck2_jnt
          - [ ] bumSlider_neck3_jnt
    - [ ] l_side_bumSlider_jnt
    - [ ] r_side_bumSlider_jnt
  - [✓] leftLeg_joint
    - [✓] leftShin_joint
      - [✓] startFootLeft_joint
        - [✓] endFoodLeft_joint
          - [✓] joint12


IGNORE THIS
Body
IGNORE THIS 0
SpecialMoves_upper
SpecialMoves_Lower

Weight: 0
Mask: KidUpper
Blending: Override
Sync: [ ] Timing: [ ]
IK Pass: [ ]

Having the option to divide the rig into different animations, allowed the scarf to work on its own while the animator animated everything in one animation.

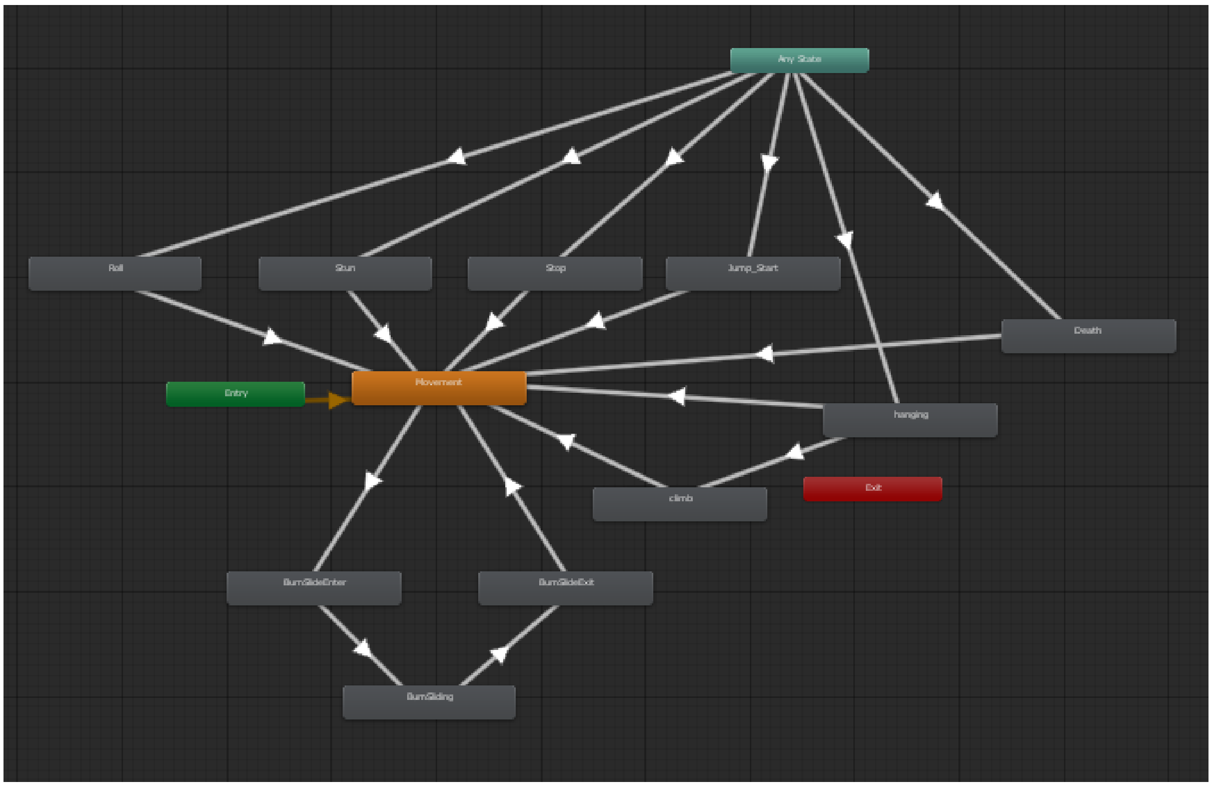
We later decided to cut the scarf idea, mainly because of animation constraints and the visual aspects of it. Despite of the removal of the scarf we kept our avatar masks for throwing, having the upper and lower body of the avatar work separately while aiming. This saved our animator a lot of work not having to hand animate every possible scenario.

Our animation trees

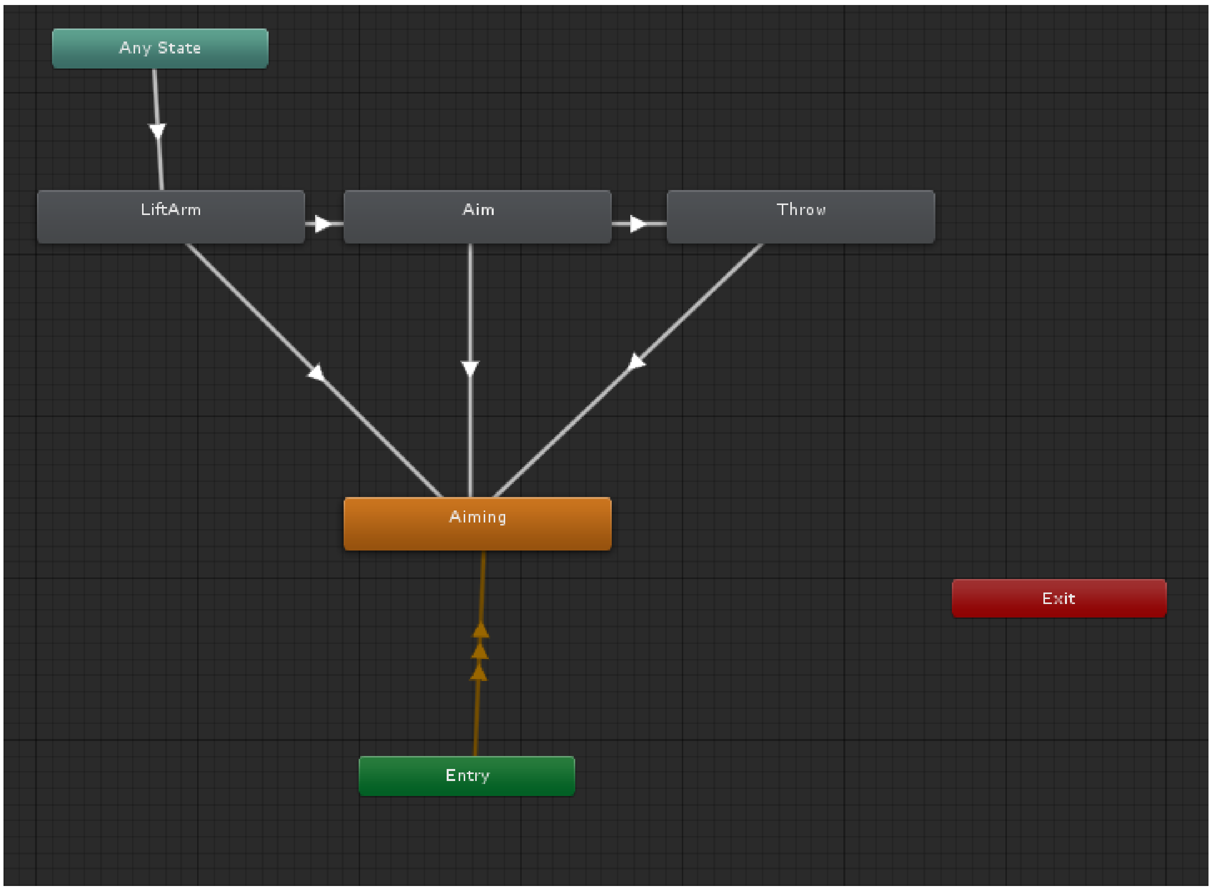
Player

As mentioned in the section above, the player animation tree consist of avatar masks and layers, which have lead to many different challenges, many of them having to do with timing, and transition between layers. Our player animation tree is divided into three different layers:

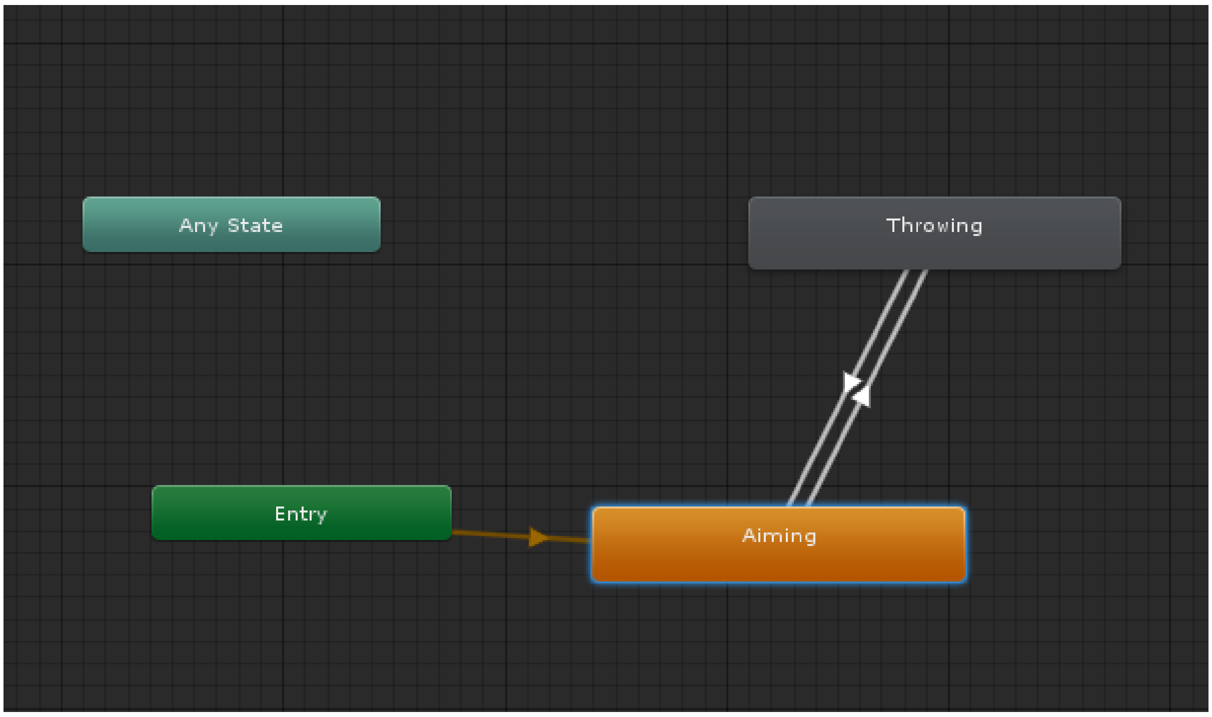
Body (also known as the main animation tree used for normal movement)



SpecialMoves_upper (Used to control the avatars upper body during throw)

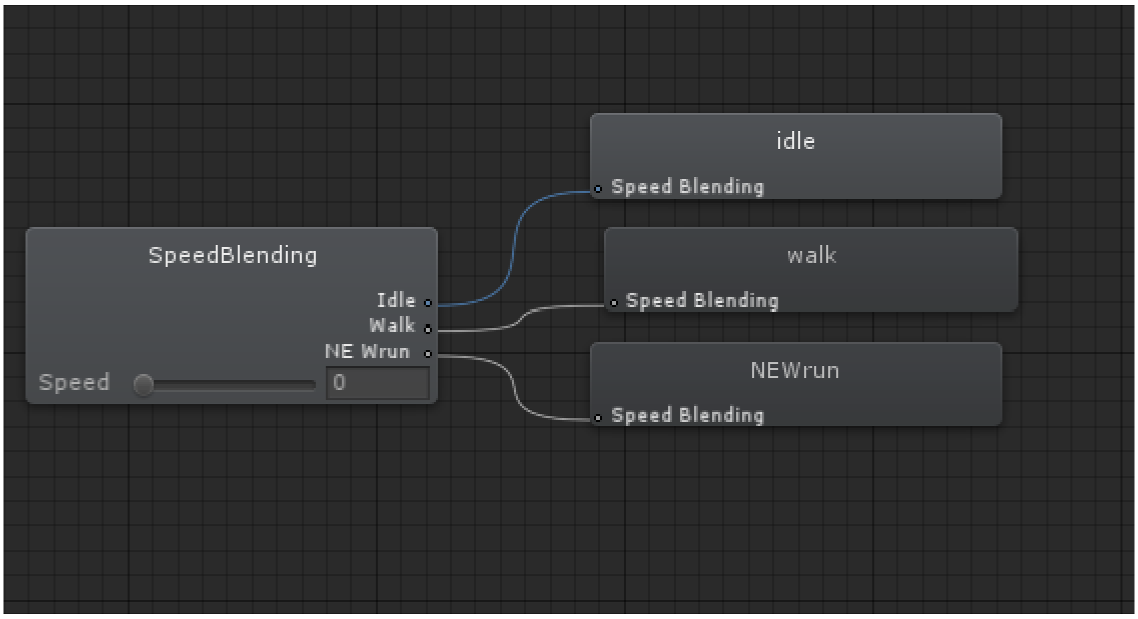


SpecialMoves_lower (Used to control the avatars lower body during throw)



Body

Using blend-trees allowed us to get a smooth transition between idle, walking, and running. This made the avatar's movement more realistic, and could be accurately timed with the movement speed by dividing current speed by max speed, returning a float between 0 and 1. This value was then sent into the blend-tree shown below:

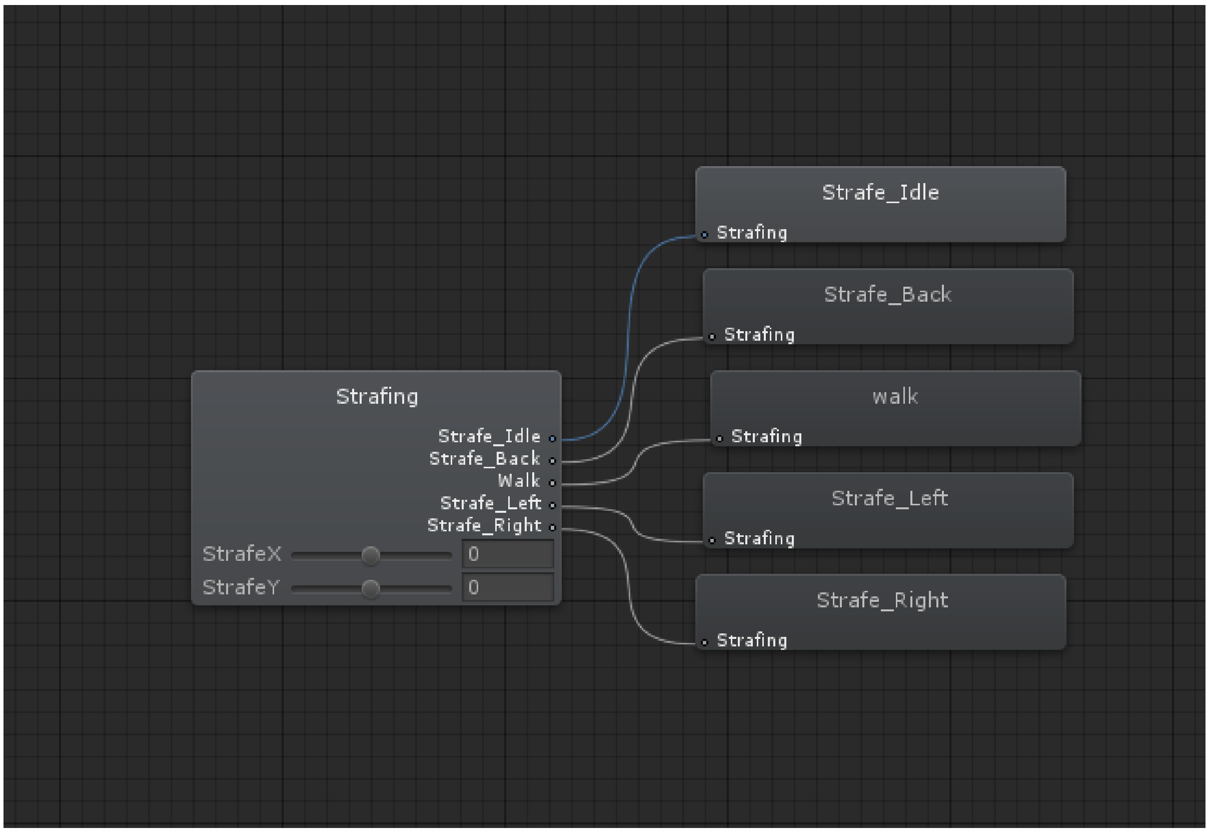


The main difficulties with constructing the body was the challenge of predicting player behavior, we noticed throughout the production that players tends to play the game different than one another. This added a lot of difficulties when having to use the "Any state" option in mecanim. Having different animations writing over each other, created frustration for both the developers and the players. Despite of this, we later redesigned the player mechanics which lead to a much cleaner way for the mechanics to communicate with the animations. With the player mechanics redesigned it allowed us to use less animation events baked into the animations, and could now add proper triggers for the animations inside the code.

Special moves

Both the upper and lower special moves layers are similar, the major difference is that the upper contains the holding of a snowball before the avatar throws. The logic around it is to have every possible situation covered. Therefore if the player stand still and throw the lower body will shift over to shooting, but if the player is moving, it will have to lower part of the body move the legs as if you were strafing during aim.

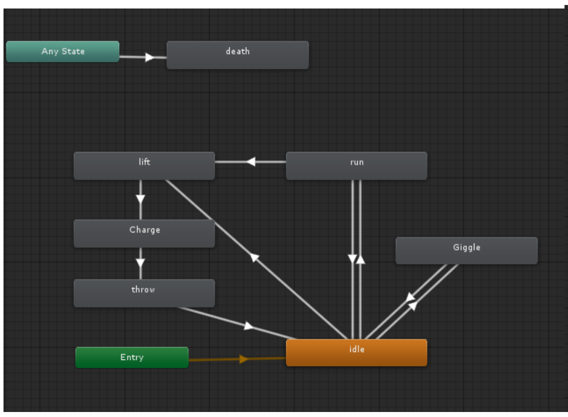
On top of this we blend between the different strafing animations to give a smooth transition when using a joystick, shown in the picture below:

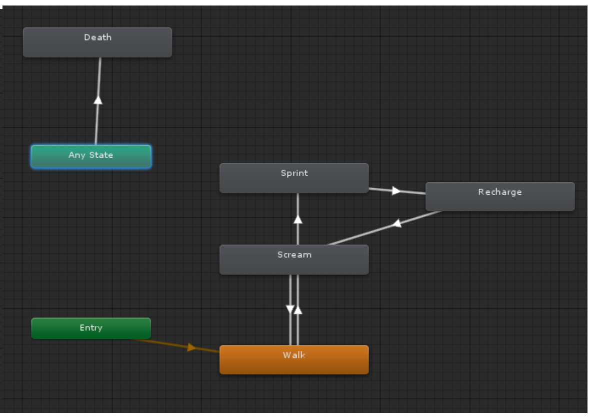


Enemies

Creating animations for predictable units was a lot easier than creating the avatar animation-tree, due to them being replicated from the behavior design talked about under the enemy mechanics. Having straight forward behaviors lead to little overlaps, because there were no need for using the "any state"-functionality.

Nisse animation-tree: *Hedgehog animation-tree:*



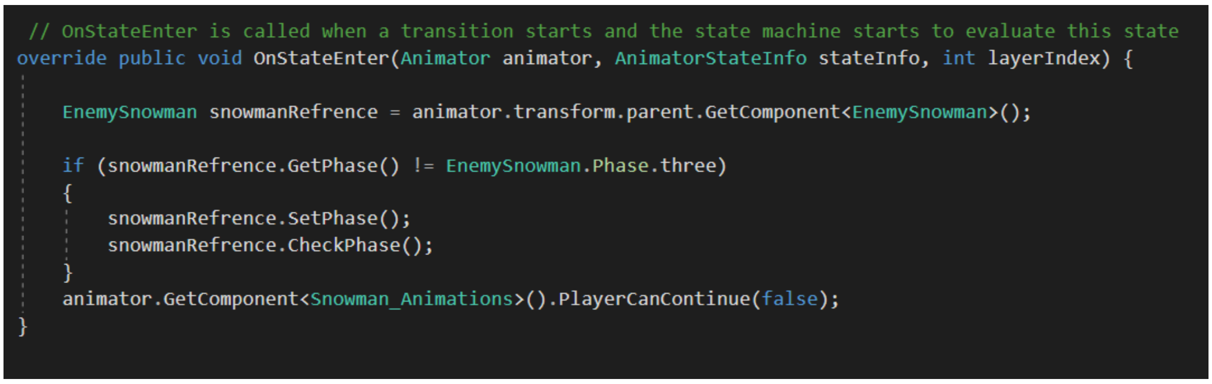


Boss

The boss animation-tree is a little more complex than the other enemies, much due to it having different phases that we do not want overlapping. Here we had to use what we had learned with layers from making the player animations. Dividing each layer into the different phases mentioned under "Boss Fight" earlier in this report.

To make the boss fight animations as consistent as possible we refrained from using "any-state". The reasoning behind this was to make the animations-trees work well with the code, and use the code to determine at what stage the boss is in.

Although the code controls everything, we use OnStateEnter() when entering the idle to check and set current phase.

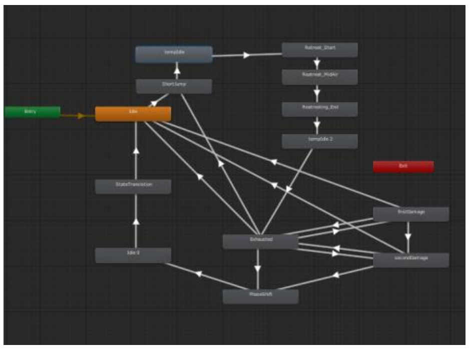


```
 // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {

    EnemySnowman snowmanRefrence = animator.transform.parent.GetComponent<EnemySnowman>();

    if (snowmanRefrence.GetPhase() != EnemySnowman.Phase.three)
    {
        snowmanRefrence.SetPhase();
        snowmanRefrence.CheckPhase();
    }
    animator.GetComponent<Snowman_Animations>().PlayerCanContinue(false);
}
```

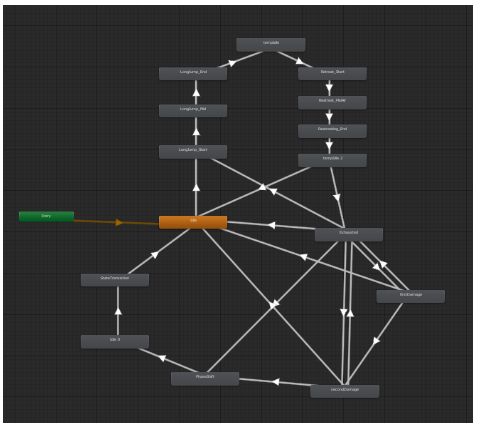
As seen above we had to refrain from this in the third phase due to the use of timeline to push the player back during the encounter.

In the pictures of the different phases below you can see some similar logic. Both health and idle was implemented to make it easy to add more phases.

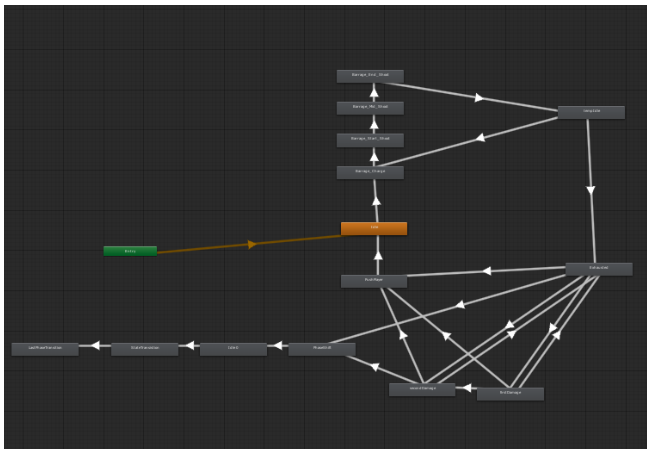
Phase 1:



Phase 2:



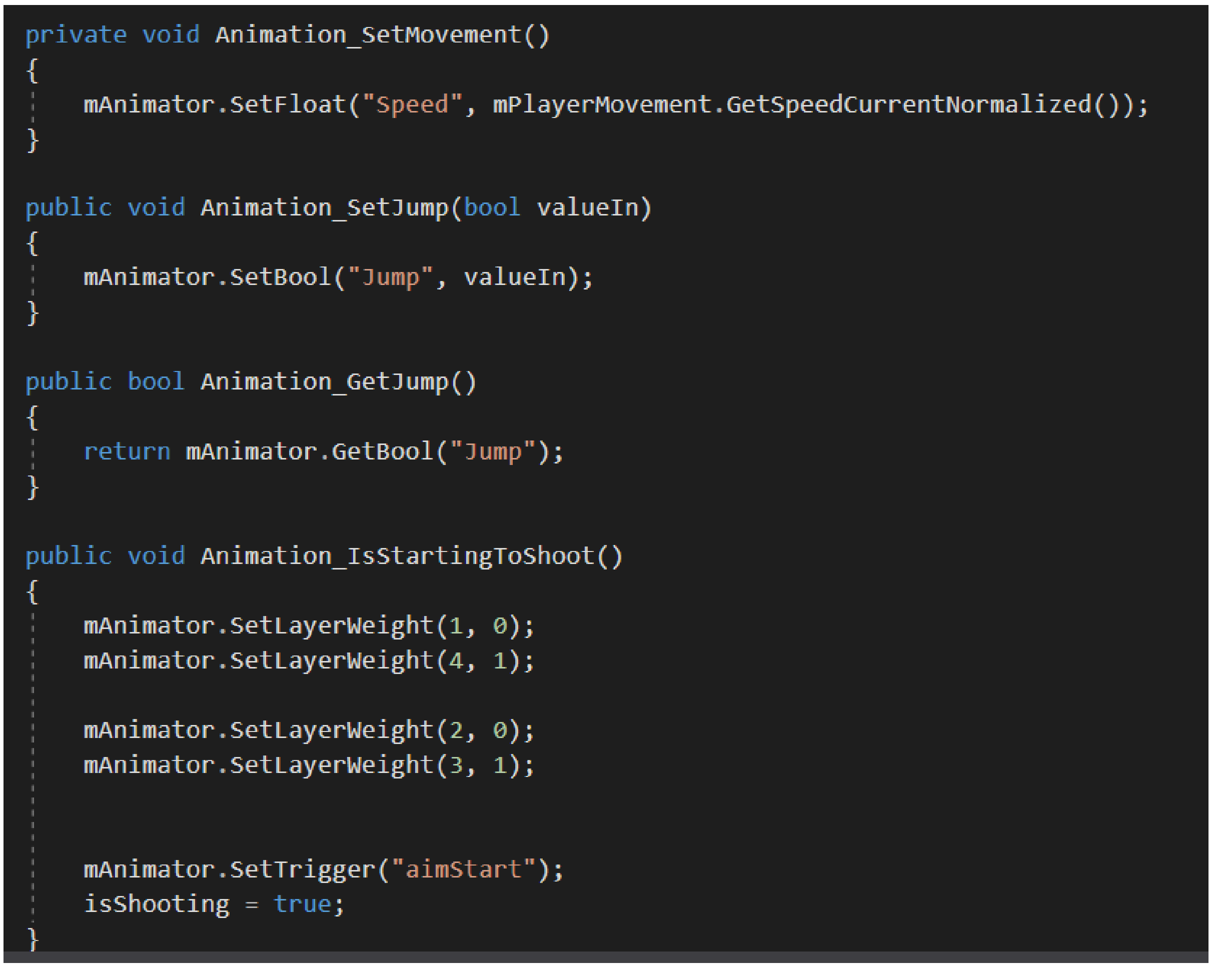
Phase 3:



Coding animation behavior

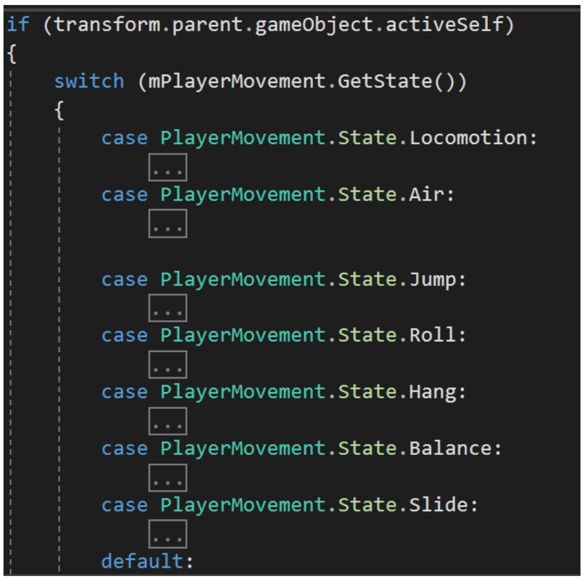
The most important thing we had to keep in mind when creating the animation behavior was to keep it modular. It was going to be implemented and called upon by many different scripts and behaviors, and had to be able to contact the animation behavior.

What we ended up doing was making a lot of public functions to allow different scripts, and objects interact with them. A typical animation behavior can be seen here:



```
private void Animation_SetMovement()
{
    mAnimator.SetFloat("Speed", mPlayerMovement.GetSpeedCurrentNormalized());
}

public void Animation_SetJump(bool valueIn)
{
    mAnimator.SetBool("Jump", valueIn);
}

public bool Animation_GetJump()
{
    return mAnimator.GetBool("Jump");
}

public void Animation_IsStartingToShoot()
{
    mAnimator.SetLayerWeight(1, 0);
    mAnimator.SetLayerWeight(4, 1);

    mAnimator.SetLayerWeight(2, 0);
    mAnimator.SetLayerWeight(3, 1);


    mAnimator.SetTrigger("aimStart");
    isShooting = true;
}
```

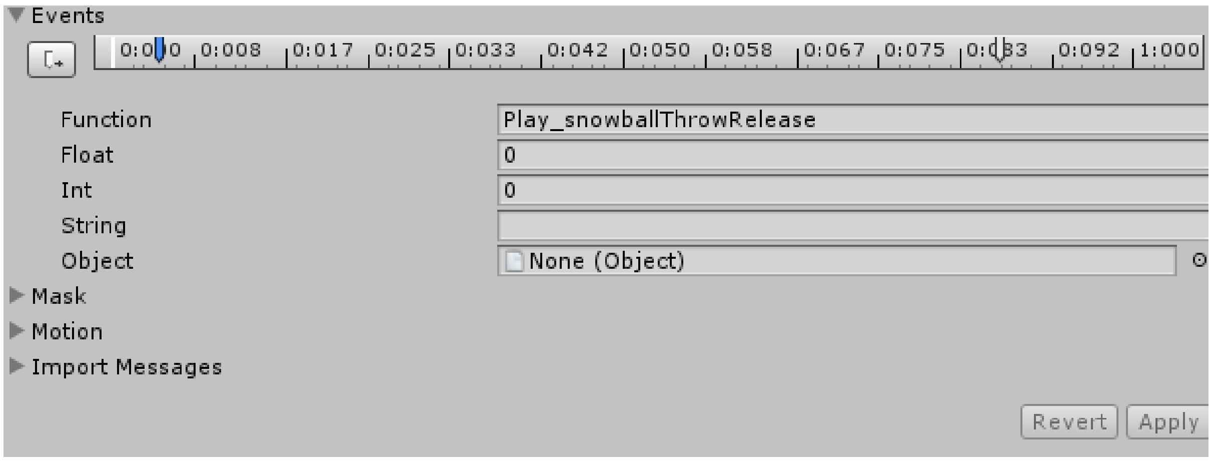
The player animations in particular, which we previously mentioned was relying on using “any-state”, forced us to implement the FSM previously used in the avatar mechanics inside of the player animation behavior, as seen in the picture below:



```
if (transform.parent.gameObject.activeSelf)
{
    switch (mPlayerMovement.GetState())
    {
        case PlayerMovement.State.Locomotion:
            ...
        case PlayerMovement.State.Air:
            ...

        case PlayerMovement.State.Jump:
            ...
        case PlayerMovement.State.Roll:
            ...
        case PlayerMovement.State.Hang:
            ...
        case PlayerMovement.State.Balance:
            ...
        case PlayerMovement.State.Slide:
            ...
        default:
```

Combining Animations and Sound

Allowing us to easily create sounds at the correct time for our animations we decided it would be best to bake events into the animations.


Events

0:000 0:008 0:017 0:025 0:033 0:042 0:050 0:058 0:067 0:075 0:083 0:092 1:000

Function	Play_snowballThrowRelease
Float	0
Int	0
String	
Object	None (Object)

Mask

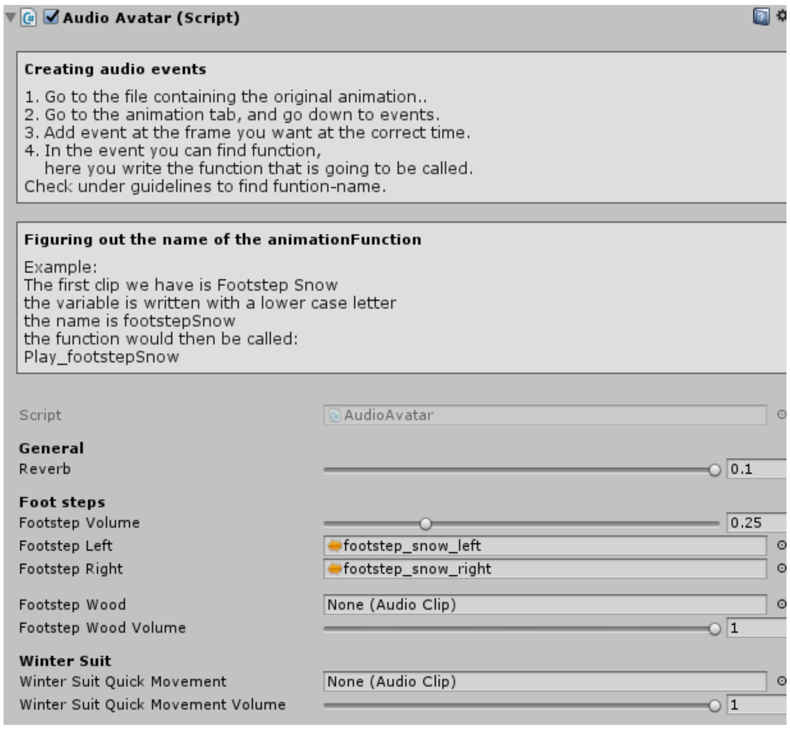
Motion

Import Messages

Revert Apply

This allowed us to get the correct timing on sounds. Another reason to be using the animation events, was to allow the art team to add the sounds, removing a lot of the bottle necking from a programmers stand point.

To better allow the artist working with sound on how to implemented we used editor scripts to all our audio sources. This editor script contained a How-To-Section within the inspector to avoid back and forth communication to figure out how to implement the sounds.


Audio Avatar (Script)

Creating audio events

1. Go to the file containing the original animation..
2. Go to the animation tab, and go down to events.
3. Add event at the frame you want at the correct time.
4. In the event you can find function,
   here you write the function that is going to be called.

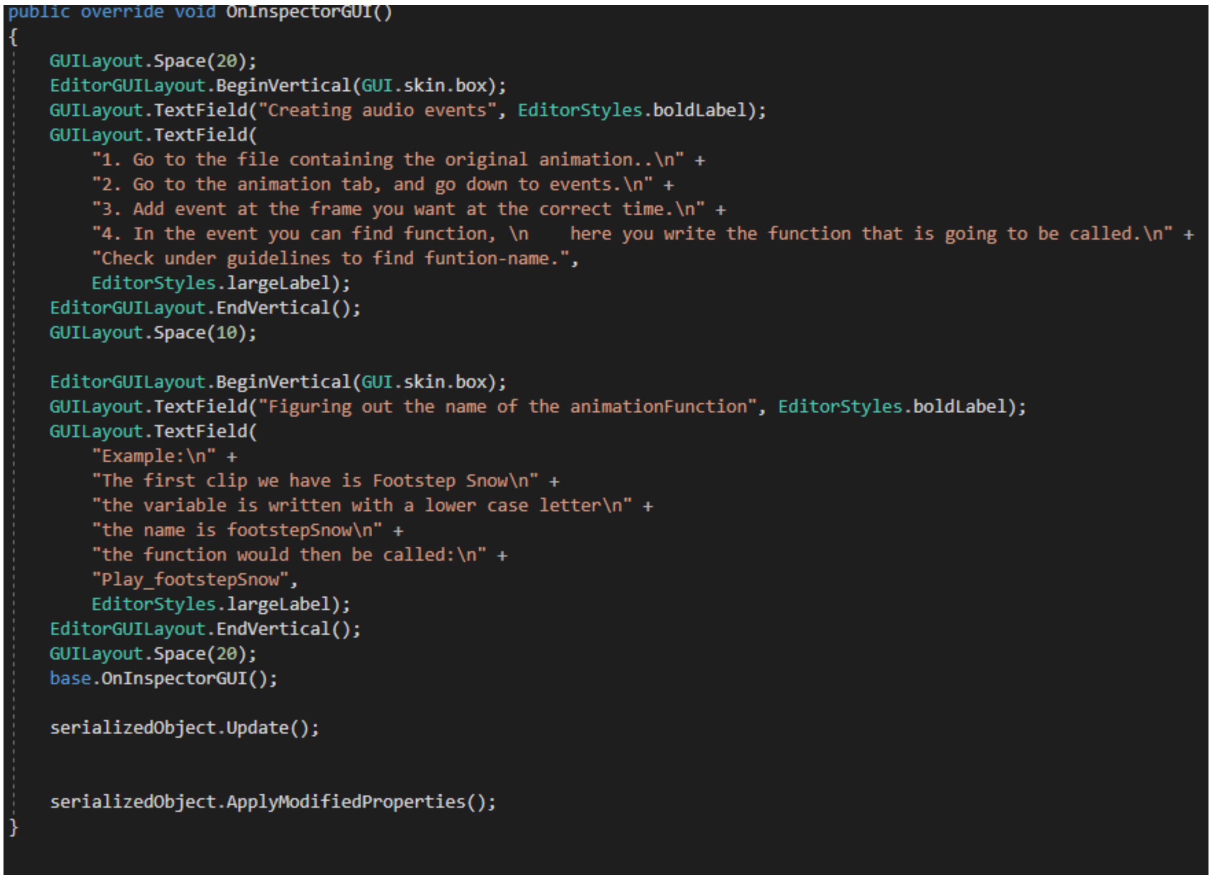
Check under guidelines to find funtion-name.

Figuring out the name of the animationFunction

Example:
The first clip we have is Footstep Snow
the variable is written with a lower case letter
the name is footstepSnow
the function would then be called:
Play_footstepSnow

Script	AudioAvatar
General	
Reverb	0.1
Foot steps	
Footstep Volume	0.25
Footstep Left	footstep_snow_left
Footstep Right	footstep_snow_right
Footstep Wood	None (Audio Clip)
Footstep Wood Volume	1
Winter Suit	
Winter Suit Quick Movement	None (Audio Clip)
Winter Suit Quick Movement Volume	1

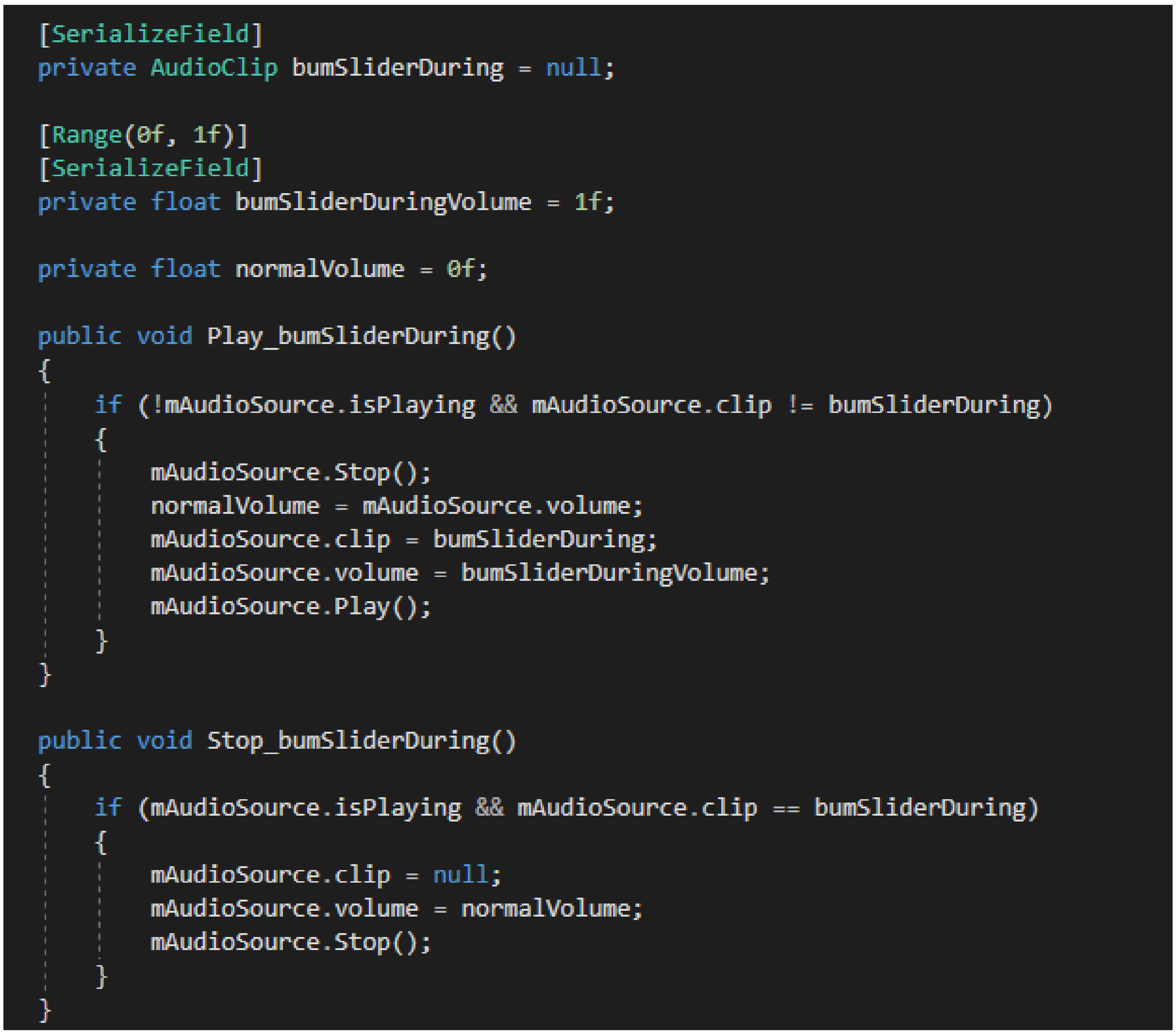
Tutorial text for working with the audio system.



```
public override void OnInspectorGUI()
{
    GUILayout.Space(20);
    EditorGUILayout.BeginVertical(GUI.skin.box);
    GUILayout.TextField("Creating audio events", EditorStyles.boldLabel);
    GUILayout.TextField(
        "1. Go to the file containing the original animation..\n" +
        "2. Go to the animation tab, and go down to events.\n" +
        "3. Add event at the frame you want at the correct time.\n" +
        "4. In the event you can find function, \n    here you write the function that is going to be called.\n" +
        "Check under guidelines to find funtion-name.",
        EditorStyles.largeLabel);
    EditorGUILayout.EndVertical();
    GUILayout.Space(10);

    EditorGUILayout.BeginVertical(GUI.skin.box);
    GUILayout.TextField("Figuring out the name of the animationFunction", EditorStyles.boldLabel);
    GUILayout.TextField(
        "Example:\n" +
        "The first clip we have is Footstep Snow\n" +
        "the variable is written with a lower case letter\n" +
        "the name is footstepSnow\n" +
        "the function would then be called:\n" +
        "Play_footstepSnow",
        EditorStyles.largeLabel);
    EditorGUILayout.EndVertical();
    GUILayout.Space(20);
    base.OnInspectorGUI();

    serializedObject.Update();


    serializedObject.ApplyModifiedProperties();
}
```

The AudioAvatarEditor script showing how we created tutorial text shown in the picture above.

This allowed the team to save a lot of time when implementing sounds into our game. Having an artist which experienced a decrease in work-tasks, could now fix the audio without having a programmering consulting. Keeping a consistent naming of functions made this a simple process, and relied more on testing the timing than the implementation itself. Even though implementing sound went smoothly, we did experience some trouble with looping sounds, having them being activated by keyframes. They usually started playing multiple sounds due to the animations looping. This was fixed by creating coroutines checking if the audio-source was playing the same clip:



```
[SerializeField]
private AudioClip bumSliderDuring = null;

[Range(0f, 1f)]
[SerializeField]
private float bumSliderDuringVolume = 1f;

private float normalVolume = 0f;

public void Play_bumSliderDuring()
{
    if (!mAudioSource.isPlaying && mAudioSource.clip != bumSliderDuring)
    {
        mAudioSource.Stop();
        normalVolume = mAudioSource.volume;
        mAudioSource.clip = bumSliderDuring;
        mAudioSource.volume = bumSliderDuringVolume;
        mAudioSource.Play();
    }
}

public void Stop_bumSliderDuring()
{
    if (mAudioSource.isPlaying && mAudioSource.clip == bumSliderDuring)
    {
        mAudioSource.clip = null;
        mAudioSource.volume = normalVolume;
        mAudioSource.Stop();
    }
}
```

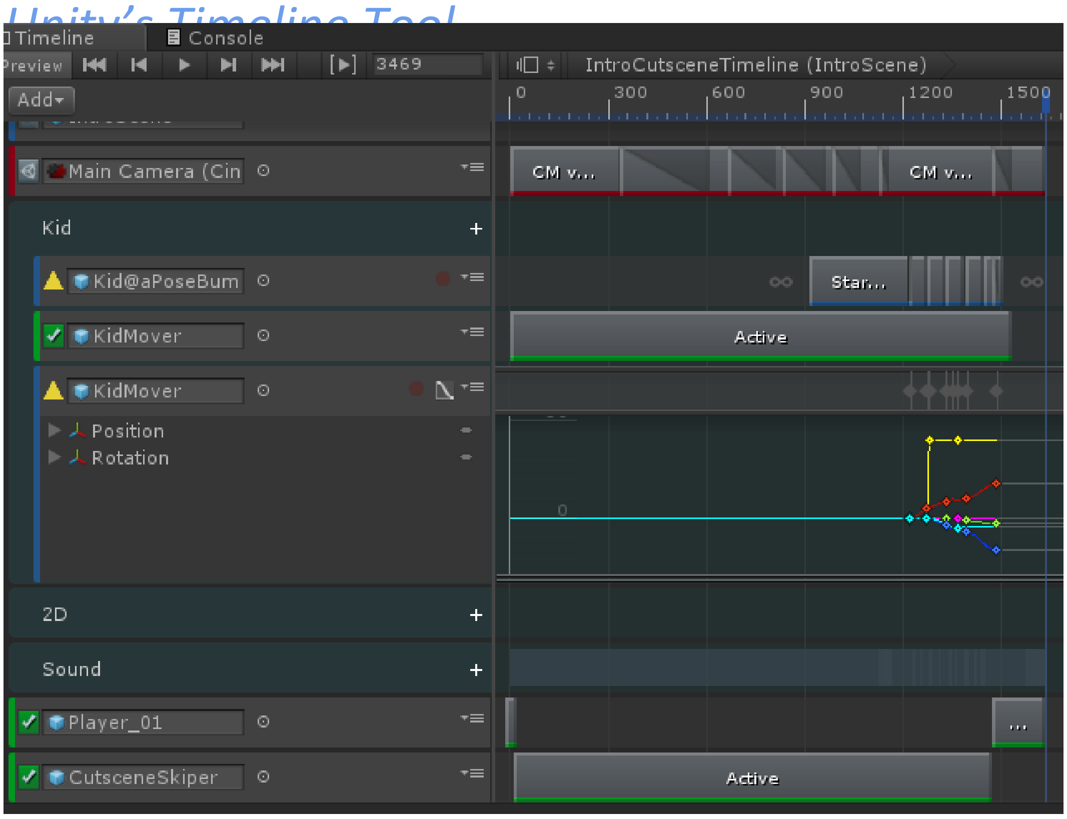
This forced a programmer to create some extra systems, but took little time and went rather smoothly.

Cutscenes

Why We Added Cutscenes

In the first couple of months the implementation of cutscenes were non-existing. After having problems with scene-transitioning being inconsistent, we decided cutscenes could be a valuable feature to our game. We immediately started researching the timeline (Unity Technologies, 2018) feature built into Unity. Alexander Freed (2017) explains how the use of cutscenes can break the immersion of the player. In one of his examples of advantages of viewpoint change, he points out the use of introductory cutscenes, these are cutscenes used before the player's point of view has been established. Having this in mind, many of our cutscenes are there to introduce the player to something new, and rarely stops the player mid action, with the exception of the final boss.

Unity's Timeline Tool



To better understand how we work with cutscenes in our game, we have to look at the tool used in Unity. The timeline tool allows you to work with animating scene in real time, creating events and cutscenes throughout your game without having to go deep into coding. The tool works much like a timeline editor used in a movie editing software, having keyframes, animations, activation, audio, and camera options. This made creating cutscenes less time consuming, and the actual code that had to be written was more about when to play, and when to skipping a scene. Instead of having to create movement, and camera angles by code, we could simply drag and drop, and use the keyframes to make it happen at the correct time. Adding OnDisable() and OnEnable() functions to objects in the scene while a cutscene were playing, allowed us to use custom code like pushing the player, or stopping them entirely from moving.

The First Implementation

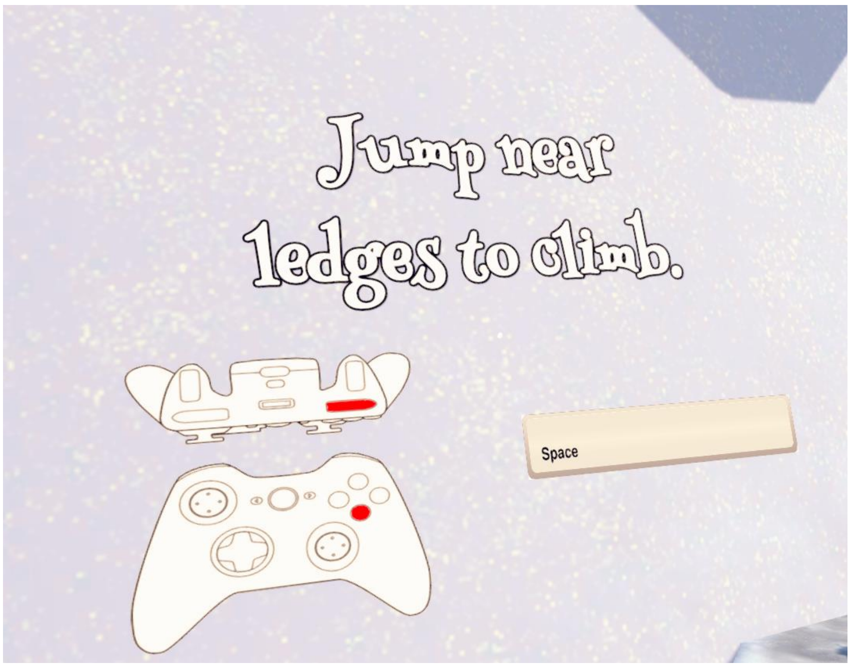
First test with cutscenes implemented was at TestBonanza #16 (see Attachment #12 - Playtest Documentation 20.04.2018). In this build we had a 15 second cutscene before the snowman-fight to give the player an introduction to the boss, small clips during the fight with the snowman to represent phase-shifting, and a small introduction when you start the game. The feedback was great and many told us that the cutscenes were helping the game in a big way to give them a connection to the world and the characters. If we consider Freed's (2017) statements about losing control of the avatar once the player's point of view had been settled, and how it would break the feeling of immersion for the player. We had to take a look at the feedback from our testers, we found to possible outcomes for why they liked it:

1. We managed to keep the player immersed with the cutscenes.
2. The initial player-avatar relationship was poor to begin with.

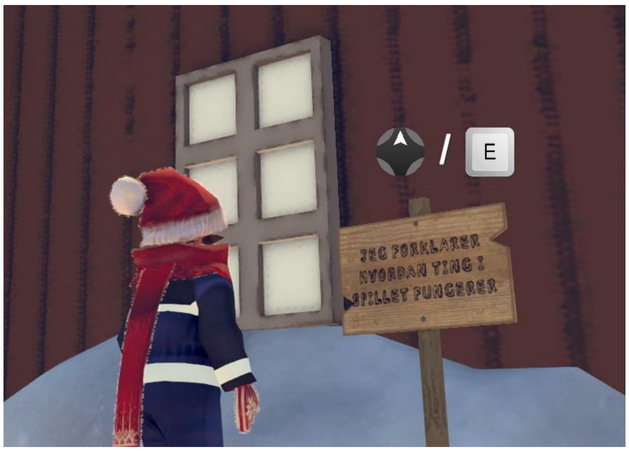
As this was late in the development phase, we had to make a choice that requires less time to implement. Rebuilding the game to make the player feel more immersed would be hard and time consuming, therefore we decided on building the player-avatar relationship through small cutscenes. To better hold the the player's point of view we decided to have all camera movement with lerping, and trying to stay away from teleporting the camera angle during these scenes.

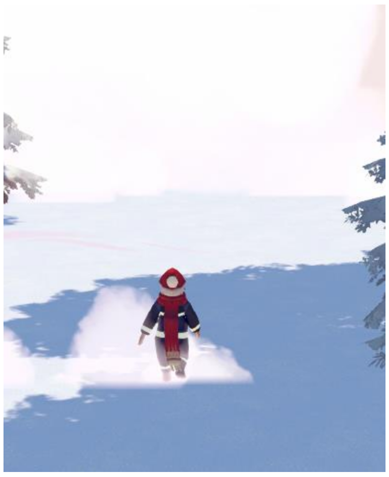
How it Looks Now

Emphasising on the feedback, we decided to look for more ways to use cutscenes to bridge gameplay and narrative further. First thing we did was to add cutscenes to signs in the tutorial level, using images of the inputs to control the character instead of having a bland text at the bottom of the screen. As this new way of conveying information progressed, we constantly had fellow developers testing, and found out that this approach was consistent and easy to understand.



Having the signs optional allowed experienced players to skip the tutorial cutscenes without giving it a second thought, while allowing new players approaching the signs to see the information multiple times.





As mentioned earlier we struggled with scene transitioning, were the biggest difficulty was getting matching geometry across the levels. This was later scraped, and instead replaced by a snowstorm particle effect, and a cutscene having the avatar walking into the storm. This way of doing scene transitions was inspired by *The Legend of Zelda: Ocarina of Time* (Nintendo EAD, 1998), where they used black caves to make scene transitions consistent.

Working with Storyboards

The last thing we had to do was fixing the cutscenes used for the intro scene and the boss fighting. Adding a bigger emphasis on telling a simple story of how the snowman stole your teddy bear. Therefor our animator started working on storyboards to tell the story better.

Working with storyboard allowed the programmer implementing the cutscene to use better camera angles, and create more consistency in the clips. The biggest change was in the first scene. This scene went from having the kid jumping out of bed, to having “Teddy is missing, go find him!” text at the start, fading over to the shelf were there are pictures of the main character and the teddy bear. The pacing of the cutscenes were slowed down, and was meant to give a more expressive feeling for the player.

Thinking Back

Using a powerful tool already implemented in Unity has helped us save a lot of time instead of having to reinvent the wheel. We could hide some of our more questionable design choices like inconsistent geometry in the scene-shifting. The narrative in the game was easier to bring in late in the development, and resetting camera angles could be done during cutscenes.

In retrospect, we should have designed the world and the game mechanics around having different camera angles using timeline and cinemachine. Due to us finishing the gameplay without having this in mind, the limitation it brought made cutscenes only work while the player had no control over the avatar. A good example is the boss fight arena, we could have made a unique camera behavior for the whole fight. Allowing the player to be in complete control throughout the fight, instead of losing control if they move the camera around to much. This was perhaps one of the biggest challenges for us using cutscenes, and could easily have been avoided.

Conclusion

Developing Nordavind was both a technical and management challenge. From the start, we knew that we had overscoped the project, but we were always prepared to scale down. This saved our project several times from falling apart, and we executed the product on time.

We set out to learn new concepts during production, like combat systems and shaders, and we managed to implement these well into the game. Both the programmers are satisfied with the learning experience, and have built up some skills that are transferable to other projects.

Creating the technical solutions to the game’s design required much discussion and insight, and although we struggled with finding the best implementations for us, we found solutions that were adequate. For example, we would have liked to implement a well-functioning over-the-shoulder style of throwing, but we had to make cuts for the production to survive. Overall, we are happy with the result.

Managing a team of five in-house students plus one remote student, was not as much of a challenge as we had expected. Implementing the Scrum methodology means that the project manager lets go of their micromanaging and puts trust in their team members. People managed themselves generally well, and when missing a deadline, we simply downscaled. Being ready to downscale from the start really paid off.

References

Adams, E. (2014). *Fundamentals of Game Design*. San Francisco, CA, United States: New Riders.

Ahearn, L. (2001). *Budgeting and Scheduling Your Game*. Retrieved from https://www.gamasutra.com/view/feature/131492/budgeting_and_scheduling_your_game.php

Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., ..., Thomas, D. (2001). *Principles behind the Agile Manifesto*. Retrieved from http://agilemanifesto.org/iso/en/principles.html

Bøhler, C., Knighton, L. (2017). *Project Management - Getting a good start* [Powerpoint Slides]. See Attachment #13.

EA Canada. (2003). *SSX 3* [GameCube, PlayStation 2, Xbox]. United States: EA Sports BIG.

Freed, A. (2017). *On Cutscenes and Viewpoint Changes*. Retrieved from http://www.gamasutra.com/blogs/AlexanderFreed/20170221/291946/On_Cutscenes_and_Viewpoint_Changes.php

Hunicke, R., LeBlanc, M., Zubek, R. (2004). *MDA: A Formal Approach to Game Design and Game Research*. Retrieved from http://www.cs.northwestern.edu/~hunicke/MDA.pdf

Lengyel, E. (2016). *Foundations of Game Engine Development, Volume 1*. Lincoln, CA, United States: Terathon Software LLC.

MinionsArt. (2017). *RenderTexture Painting Examples* [Patreon Tutorial]. Retrieved from https://www.patreon.com/posts/rendertexture-15961186

Nintendo EAD. (1998). *The Legend of Zelda: Ocarina of Time* [Nintendo 64]. Kyoto, Japan: Nintendo.

Nystrom, R. (2014). *Game Programming Patterns*. Seattle, WA, United States: Author.

Unity Technologies. (2018). Unity game engine. Retrieved from https://unity3d.com/

Unity Technologies. (2018). Animation system overview. Retrieved from https://docs.unity3d.com/Manual/AnimationOverview.html

Unity Technologies. (2018). Avatar masking. Retrieved from https://docs.unity3d.com/Manual/AnimationMaskOnImportedClips.html

Unity Technologies. (2018). Particle system. Retrieved from https://docs.unity3d.com/Manual/class-ParticleSystem.html

Unity Technologies. (2018). Profiler. Retrieved from https://docs.unity3d.com/Manual/ProfilerWindow.html

Unity Technologies. (2018). *Surface Shaders with DX11 / OpenGL Core Tessellation*. Retrieved from https://docs.unity3d.com/Manual/SL-SurfaceShaderTessellation.html

Unity Technologies. (2018). *Timeline*. Retrieved from https://docs.unity3d.com/Manual/TimelineSection.html

Attachments Overview

- Attachment #1: Development Plan
- Attachment #2: Asset List, Overview
- Attachment #3: Asset List, Characters
- Attachment #4: Asset List, World
- Attachment #5: Asset List, Systems
- Attachment #6: Climbable Geometry Restrictions
- Attachment #7: Walkthrough of the Game (see video in the Attachments folder)
- Attachment #8: Showcase of Snow Deformation Shader (see video in the Attachments folder)
- Attachment #9: Game Design Document
- Attachment #10: Playtest Documentation 28.02.2018
- Attachment #11: Playtest Documentation 23.03.2018
- Attachment #12: Playtest Documentation 20.04.2018
- Attachment #13: Project Management Presentation2
- Attachment #14: Game Manual