Denis R [redacted]

DEVELOPMENT OF A CUSTOM CHARACTER CONTROLLER FOR UBISOFT GAME LAB

COMP 396 REPORT

Introduction

This paper provides a summary of the development of a custom character controller for the game *Besunder* developed for the 2021 Ubisoft Game Lab competition. This is a game development competition in which 22 teams from different universities in Quebec have 10 weeks to make a 10-minute playable game prototype that fits a particular theme. This year it was “apart//together”. This paper’s author was a producer and programmer for one of the McGill teams sent this year. While their responsibilities included many management tasks on top of programming, this paper focuses exclusively on the latter, specifically on the development of a custom character controller whose development took nearly the entirety of the competition.

Character Controllers and *Besunder*

The player-controlled character is arguably the central piece of any game. Its functionality enables the player in the game world. It should therefore not get in the way or hinder the player’s experience. Bugs, bad button mapping, inconsistent behaviors are a few things to watch out for when making a character controller. With few exceptions, controls should be enjoyable, providing a rewarding learning curve. Achieving this is a lengthy process that requires constant testing and feedback consideration. These are all general guidelines that apply to almost any game, regardless of its genre. A game’s genre will come into play to define more specific requirements and design guidelines, a game like *Mario Kart* does not have the same specification as *Call of Duty*.

Besunder is an online 2 player co-op only 3-D platformer with a 3rd person camera view. A platformer is a video game sub-genre of action game where the primary goal is to move from point A to B by solving movement and acrobatic puzzles; in our case these puzzles require player cooperation. Games like *Super Mario 3D World*, *Mario Odyssey* and *Furi* were direct sources of inspirations for the design. In *Besunder*, 2 players must navigate a strange world of floating islands by solving jumping puzzles and fight off the monsters that have been unleashed upon the land. To do so the players have at their disposal a highly acrobatic and fast paced movement set, that will be detailed later.

General Design

The movement system design is directly inspired from a modular design by game development youtuber Dapper Dino (2020). The design consists of 2 parts:

(i) Movement handler

(ii) Movement modifier interface.

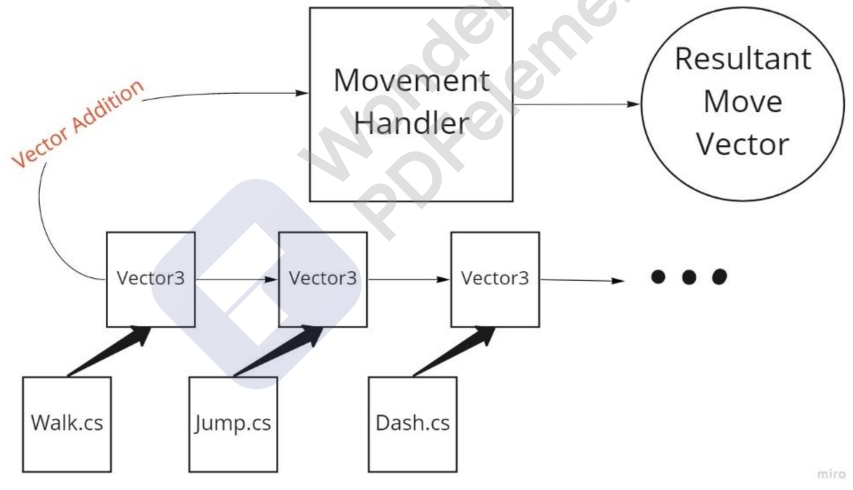

Figure 1: Movement System Design

The movement handler is a centralized system that holds a list of movement modifiers. A movement modifier is an interface which represents an action. It holds a movement vector for the frame and the movement type it is associated with. The handler's list is dynamic and can be updated at run time as different movement modifiers can remove themselves from the list when a particular behavior is explicitly disabled and ready themselves when reenabled. Note that here disabled means stopping a script from running, not when its action or power is not being used as the constant list manipulation would be unnecessary overhead. Instead, when a modifier's action is not being used, the vector sent to the handler is a zero vector.

The movement handler iterates over the list and adds all the vectors to produce a resultant move vector in world space. Furthermore, it provides a straightforward way to filter out specific modifiers during a particular state as they can be identified through the interface's type Enum and discarded from the computation for as long as we want. This was especially useful for establishing priority moves.

This system was developed to create a modular and scalable system that spans multiple semi-independent scripts that manage their own actions with minimal interdependency (no interdependency is not possible as some actions are not allowed to happen under some states). The result is a complete system that spans 9 files instead of 1. It may seem complicated from the outside, but the benefits are scalability, ability to test compartmentalized behaviors and not having a flow control that runs through thousands of lines in one single file.

Finally, this was developed with the main purpose of controlling player action movements, but an unanticipated side effect was that every movement imposed over on the player by the environment and enemies had to be done through the system and be considered a movement modifier too. Not doing so would cause inconsistent behavior and visual bugs during gameplay. For instance, during testing, any elevator contraptions pushed and pulled the player. A movement modifier was added or removed dynamically upon the player stepping on or off the platform. Even though this particular use case did not make the cut for the final game, it did provide a solution for the knockback problem detailed later.

API Choice

After the design of the handler was set it was time to get the player to move. Before even coding the simplest moving functions, one more decision had to be made. There are 2 main ways to implement a character's movement in Unity; the *Character Controller* and *Rigidbody* APIs. The former allows movement without making use of Unity's internal physics engine all while keeping collision constraints meaning all movement must be code driven. A *Rigidbody* implementation, on the other, hand gives control to both the internal physics engine but also to the game's code; for instance, using this implementation one does not need to code gravity. This might seem like the superior option but keep in mind we may not always want to have a default downwards gravity applied to the player. Due to this reason the decision was made to move forward with a *Character Controller* implementation.

One thing that has not been mentioned so far is the input method. *Besunder* is a PC game but PC games can be played with either controllers or keyboard and mouse. From the beginning it was decided that a controller would be the game input system; support for keyboard and mouse was never in the picture. This decision was made not only because a platformer move set maps very well to a controller, but also because by using a controller one could program haptic feedback (rumble) into the player's experience. Thus, the game functions fully with either an Xbox One or PS4 controller.

Moving and Jumping

The simplest actions to code are the walking and jumping. The player's rotation and walking are controlled by the left thumb stick. Reading this controller component yields a 2-D vector (*Vector2* in Unity's API). The vector components are used to determine the direction and its value to calculate speed. The idea for the walking is simple: make the player walk in the direction they are facing to. If the player wants to go to the left, they push the thumb stick to the left and the player's avatar will rotate left quickly as it starts moving in the direction. The first implementation moved at a fixed speed but with a later update a small speed curve was added to accelerate and decelerate. Here the first movement bug was encountered. After a letting the thumb stick go, it resets to its resting position, but in Unity its value is not always 0. Probably due to the spring that drives the thumb stick, it can hold a small residue value that is enough to gently pull the player in a direction even though the user is not using the control. This was solved by setting the value to 0 when a minimal threshold is not satisfied anymore.

A fixed height jump was also added at this stage with the intention to rework it later to give the player the ability to jump different heights depending on how long they would hold the button down. Since the game uses the *Character Controller* API, gravity needs to be coded manually. Thus, the gravity and jump simulation was packaged into the same movement modifier. Alas, here the jump was an upwards vector of a set intensity that was slowly overcome by an increasing counter force that acted as gravity. This force resets upon ground contact, thus the gravity implementation can be controlled like a light switch. Soon after, the gravity switch on was extended to all instances where the player loses contact with the ground for some time; for instance, when the player walks off an edge or is hit by an enemy. Finally, since the game wants the player to experience great air control, one can move while airborne. This was achieved by letting the player "walk" while in the air while playing an airborne animation.

Next, varying jump sizes were added. Initially we thought it was a complicated behavior to code the solution turned out to be quite simple. The way this works is by adding a counter jump force that is added alongside gravity when a player jumps, and the button is released. The earlier the button is released, the faster the jump upwards vector will be overcome by counterforce and gravity. It should be noted that this counterforce only acts when the overall vector is facing up, after all we do not want it to accelerate the fall, only limit the vertical reach.

Switching to Rigidbody

Then a shift in design happened. Up to now the player is controlled with the *Character Controller* API. Concurrently, the network implementation was being developed and started using the game's character controller. The network used Photon, a 3rd party solution for networking in Unity. It used a component call *Photon Transform View* to update the local version of the other player for a client, unfortunately this produced jagged movement, especially when stepping and riding a moving platform. Under the advice of our mentors, the decision was made to use a *Rigidbody*. The belief was that letting Unity handle internally some of the aspects of the movement could help smooth out the behavior. This turned out to be the right call in the end; in the meantime, some modifications had to be done to the movement system. Under the *Character Controller* API, the player was moved through a single call of the Move() function; with the *Rigidbody* API the velocity had to be used. Since the system was not designed to conserve momentum, this meant we had to set the velocity every frame.

Additionally, this meant that Unity's default gravity always acts upon the player, but fortunately due to scale, its effect is barely noticeable and did not require a rebalancing or major rework of any of the existing systems.

The Power Trio: Dash, Slingshot and Ground Pound

So far, the character controller can walk and jump. It does its job correctly, but it is not overly exciting. Powers are needed to better the user's experience. The first and the simplest one is the dash. As its name suggests, when activated the player speeds up instantly and bolts in the direction they are moving for a moment. In the final version of the game, the dash could be used to push enemies afar. A dash reset was also established, to be reused the player needs to touch the ground thus dashes are chainable when grounded but while airborne. This decision was made to not overshadow the slingshot mechanic whose primary objective is to provide quick long range movement coverage.

The slingshot is the main "gimmick" of the game and the materialization of the "apart//together" theme. With it, players can cover great distances to be reunited, come to each other's help, or reach otherwise unattainable terrain. The premise is simple, the slingshot is a co-op mechanic where a player is shot a high speed towards their partner. However, its development was also the longest and hardest with 5 major reworks over the course of 6 weeks.

Things started out roughly, the initial implementation was unnecessarily complicated. It required both players confirmation to engage in the move, the first one to press the button would act at the target and the second player press would enable the projectile state. When player 2 releases the button they are shot towards their partner. This was a badly designed system, it was unintuitive, and required RPC calls to get the other player's state over the network. It was made worse by the fact that this early in the game visual feedbacks were not remotely a priority in the development pipeline and the only way to know your state is to try the mechanic in the first place. Additionally, there was a minor bug in which the direction vector was drawn at press time, instead of release which meant that if the 1st player moved after the 2nd player had pressed the button the latter would still be shot towards the original player 1 position.

The next iteration simplified things greatly. The state confirmation system was abandoned, and the vector draw call was scheduled at released time instead. It would be then normalized and scaled back up using a constant power coefficient to be then sent to the handler every frame the move is active so it can execute the motion. Things were not perfect; we have not spoken about the move's termination clause. We do not want the player to possibly get stuck moving at high speeds towards infinity. The first solution was to stop the motion upon landing. This brought many issues. To better understand this, look at Figure 3.

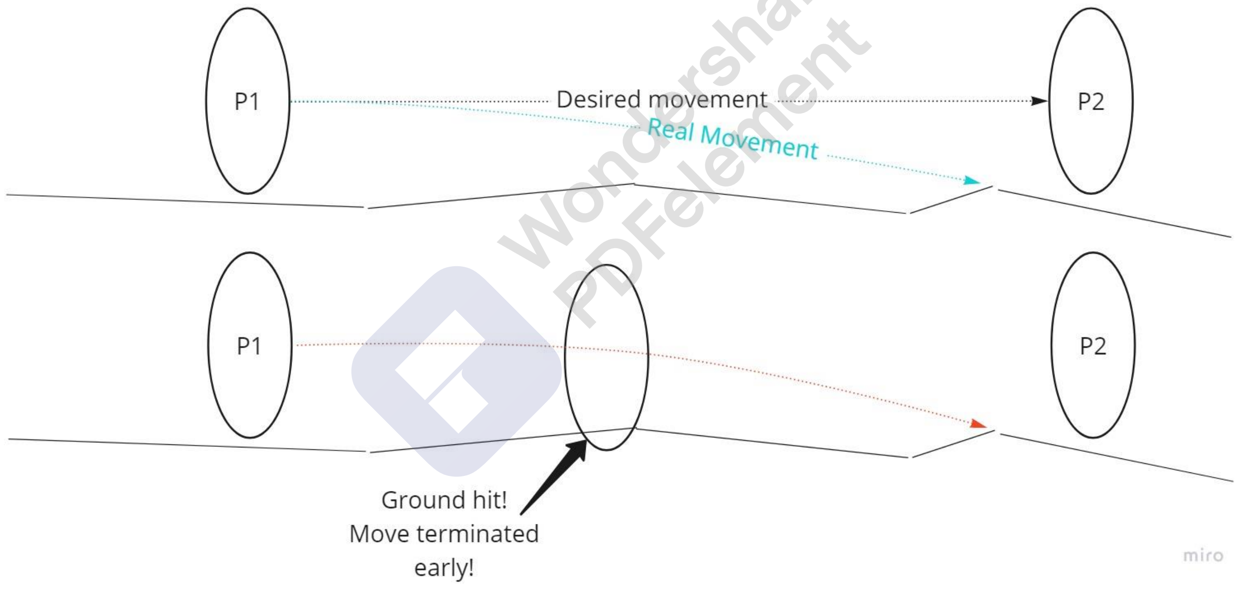

Figure 2: Example of the early termination behavior.

Notice how when the players are relatively distanced apart from each other the direction vector is nearly horizontal. Factoring in the code simulated gravity one can see that the player will be pulled towards the ground, ending the movement early. Even if we filter out the gravity vector using the movement handler, there still is Unity's built-in gravity. It is much weaker but since the ground of the islands is uneven now the move stops prematurely in what seems to be a non-deterministic way. Both outcomes are awfully bad. A simple non solution we went for then was to draw the direction vector to a target above the other player's head. This measure reduced the occurrence of an early stop but did not solve the issue.

Additionally, it altered the behavior of the slingshot. Consider the opposite of being far away from your teammate. The closer the 2 players the more vertical the direction vector is. The model is consistent, but it is not intuitive to the player who expects to be gently pulled towards their teammate when in proximity, not shot up to the sky. A new model would be required.

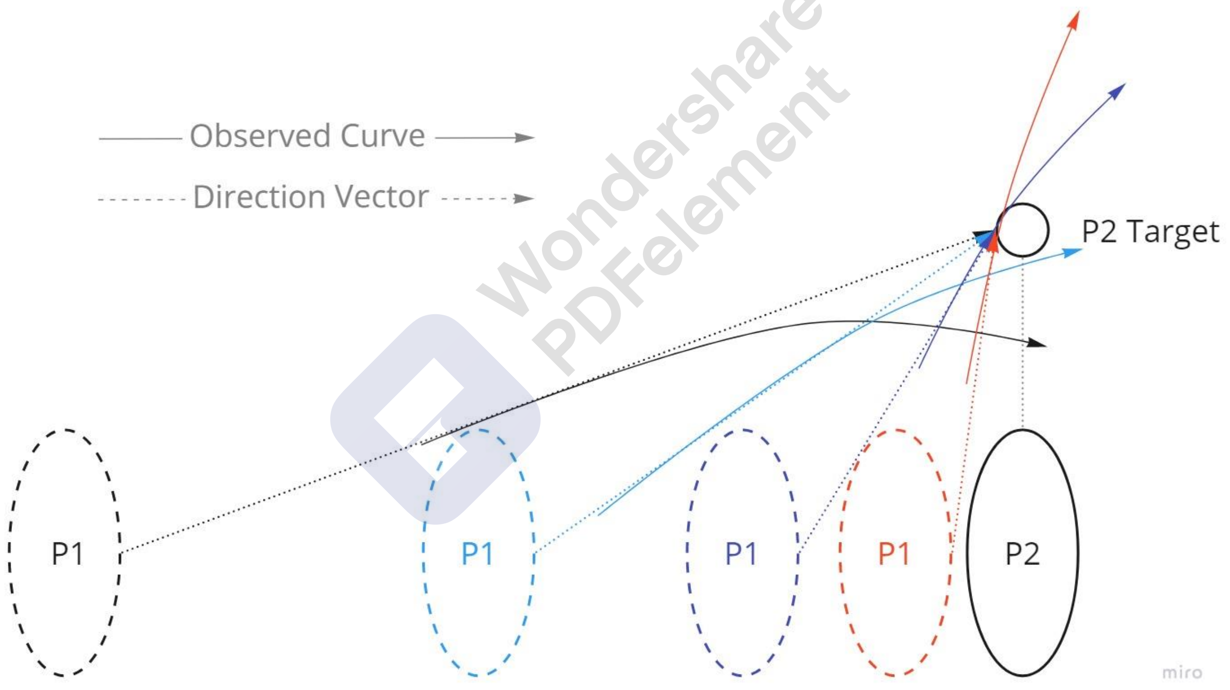

Figure 3: Example of the undesired curve behavior at proximity.

Up to this point every slingshot version shared one thing in common: they were all modeled to describe a parabola trajectory. In other words, gravity was enabled all the way through. This would not be the case for this iteration. The new model separated the move in 2 parts: (i) an uninterruptible dash towards the other player with gravity disabled and (ii) the fall after gravity is reenabled. The target was lowered to be around head level of the other player and the distance dependent power scaling was reworked (with some lower an upper bound for consistency). The first part of the move acted as a long uninterruptible *super*-like dash towards the target. The second part of the movement had 2 trigger clauses, only 1 of them needing to be satisfied to trigger the fall. The first was a distance test. If the player got within a certain range of the target, then gravity was enabled, and the fall would start. The alternative clause was a failsafe; the 1st part would run on a timer, after which the fall would start regardless. This was added because, if the target player moved fast enough, they could avoid getting within trigger range for the main stop clause. This was the first version of the slingshot that was "release" ready.

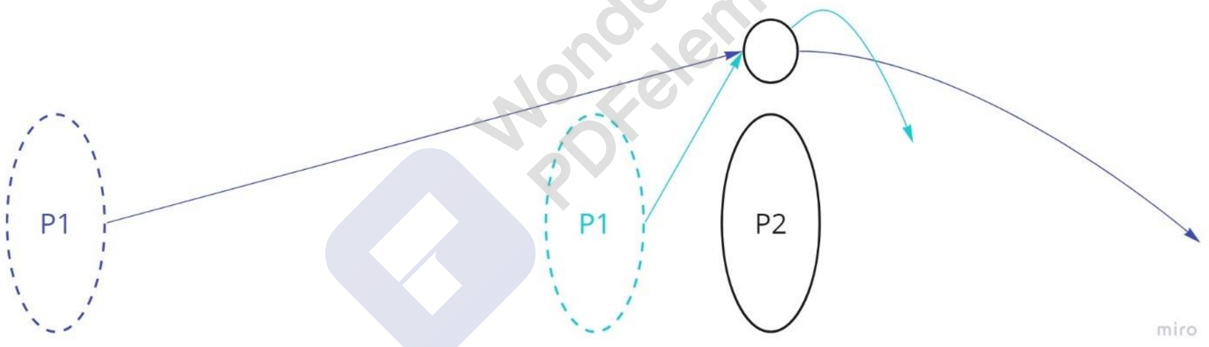

Figure 4: The final slingshot version provides a much more intuitive curve.

Finally, upon request from the designer the last iteration saw the addition of a jump reset when the move is executed, meaning now players gained a new jump when slingshot.

The final power move was the ground pound. Its inception was purely accidental but ended being crucial to the combat experience. Recall the problem exhibited in Figure 4. A negative effect of being shot high in the sky is that it locks the player out of the game until they fall back down. The ground pound was added to deal with this. The ground pound is an airborne-only priority move. Whenever a player is airborne and has control the player can choose to ground pound to stop their motion and head down straight for the ground. Players smash every enemy nearby the point of contact kicking them away. This also became a good complement to the slingshot. Now players had a way to react to overshooting a platform. It became a well-rounded offensive and defensive move that replaced the punch.

The Knockback

The knockback is the only movement modifier that is not controlled by the player. A knockback is an effect inflicted upon the player by the combat manager through Unity's AddForce() method. The original implementation disabled the movement handler's script temporarily too. This produced a knockback trajectory that looked like a right-angled triangle. The player would be shot up extremely fast but then be pulled down even faster. This happened because even though the movement handler was disabled the jump script was not. Therefore, when the player became airborne gravity started growing in intensity, but it had no effect on the player. Until the handler was reenabled and gravity was strong enough to push the player back down instantaneously. This not only looked wrong but also took out any consequence for getting hit, as the player would always land mid airborne and recover.

This was solved by making the knockback a priority movement modifier in the system. When hit, the combat manager calls a method in its script that produces an up backwards facing vector to push the player back. Additionally, when hit, the player loses control until they land again. It is the only airborne state where the ground pound is not available. This greatly improved the combat experience.

Conclusion

Developing *Besunder*'s character controller was an enormous challenge. It required countless hours of research and testing to achieve its current state. Comments on it have been positive, pointing out the smooth and responsiveness it exhibits, as well as satisfying abilities.

There are areas to be improved, however. As of now many actions rely on each other states to block or trigger. This leads to a plethora of bool states being shared across scripts, in a chaotic-like fashion.

Either way, the team wants to move ahead to develop a commercial version of the game. This is an ambitious objective. If a release is targeted, the system needs to be cleaned up even further to reach a higher standard of production and fidelity. That said, in the end, it not only did not hinder the player's experience but went as far as to be one of the game's strengths. Thus, the implementation is considered to have been a success.

[REFERENCES]

Dapper Dino (2020, June 23rd) How To Make A Modular Character Controller - Advanced Unity Tutorial [Video] YouTube https://www.youtube.com/watch?v=-PCvfltKguE