TÉCNICO LISBOA

Modelling Progression in Video Games

Pedro Miguel Henriques Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Dr. Carlos António Roque Martinho

Examination Committee

Chairperson: Prof. Dr. José Carlos Martins Delgado
Supervisor: Prof. Dr. Carlos António Roque Martinho
Member of the Committee: Prof. Dr. João António Madeiras Pereira

May 2016

“...if we’ve learned from a mistake
and become better for it,
shouldn’t we be rewarded for the learning,
rather than punished for the mistake?”

Braid, Jonathan Blow (2008)

To my parents and my sister,
for their unconditional support through every step of my life.

Acknowledgements

I would like to acknowledge and thank the most influential individuals who have supported me, not only during the course of this dissertation, but throughout my Master's degree or even life.

First of all, I would like to express my gratitude to my supervisor Prof. Dr. Carlos Martinho, for his unwavering support, guidance, insight and excitement throughout this project. His passion for video games and game design is contagious and was the main driving force that made this dissertation possible.

I would also like to thank Levin "Seneral" G., the sole developer and maintainer of Node Editor Framework for Unity, for his support in helping us adapt his tool to suit our needs. His project was the catalyst that allowed this dissertation to use a visual programming tool to represent its model of progression.

I want to thank all the testers that helped shape this project to the current state it stands: Nuno Monteiro and the remaining employees at Bica Studios; André Silva and Safal Kapoor, game designers at Miniclip; André Lima, Francisco Nina and Gonçalo Delgado, students at IST-Taguspark.

A very special thanks to my "partners in crime" Pedro Oliveira, Pedro Lucas, João Higino and Nuno Franganito. Destiny decided that the five totally different "foreigners" should stick together and be friends. Thank you for always being present and for all the nonsense conversations we have all the time. It really made a difference! Thanks in particular to Pedro Lucas for being a fellow friend, colleague and for sharing the experience, stress and motivation in developing a dissertation related to video games. We did it!

anTUNiA has been an incredibly important part of my university life. What started as a simple musical hobby to forget how hard it was to study at a university, evolved to be one the things that always made me keep on going and helped me become the person I am today. Although I am no longer an active member of this musical brotherhood, I have the obligation to thank this institution for everything it gave to me. I will never forget the friendships, concerts and festivals all across the country. A special thanks to Pedro Gonçalves and to Pedro Oliveira, for always being close friends throughout all these years.

I would like to thank my entire family for all the support during all these years, especially since the start of this dissertation. Thank you for always making me feel really loved and for motivating me in becoming a better man. I love you all!

Joana, words cannot describe the positive impact you had in my life since we met. Thank you for being my best friend, my confident, my better-half and for always being there whenever I needed you. I will always be eternally grateful to you.

Finally, it is time to thank the most important people in my life: my father my mother and my sister. Thank you for never doubting me and for always being able to provide everything I ever needed to make me what I am today. Sorry for not being the most attentive son and brother, but trust me when I say you are the most precious things in my world. I owe you all. Thank you for everything!

Abstract

Games of today are extremely dependent on procedurally generated content to engage players in replaying a game. Most endless-running platformer games tend to generate their content based solely on the duration of the current playthrough, ignoring completely the skill of the player. In order to progress, the player must adapt itself to the game, instead of the game to the player.

We propose a model for progression in video games, using the level of mastery of the player when using the different mechanics and overcoming the different challenges provided by the game. The model is used to determine which features (mechanics, challenges) will be presented next to the player in order to maintain the player engaged and in flow.

In this dissertation, we implement this model for an endless-running platform game. A custom editor was developed to allow for level designers to specify their own game logic and the rules for guiding the progression of the game as the mastery of the player evolves. The different game features are organised in a graph with conditional transitions specifying the threshold for the level of mastery needed to enable other features down the graph. As the skill of the player evolves, the game will adapt itself in providing features of the appropriate skill level, with the final objective of increasing the level of replayability, fun and engagement thanks to the constant adaption of the game to the skill of the player.

Keywords: progression, player skill, level adaptation, procedural content generation, games

Resumo

Os jogos de hoje em dia dependem bastante do conteúdo gerado, de modo a cativar os jogadores a jogar várias vezes o mesmo jogo. A maioria dos jogos endless-running de plataformas gera o seu conteúdo com base na duração da sessão de jogo, ignorando completamente a habilidade do jogador.

Neste trabalho é proposto um modelo de progressão em videojogos através da medição das taxas de sucesso do jogador em ultrapassar os vários desafios de um jogo. O modelo será usado para determinar os próximos desafios do jogo a serem apresentados ao jogador numa determinada sessão, de modo a manter sempre o jogador cativado e em *flow*.

Nesta dissertação, este modelo é implementado para um jogo endless-running de plataformas. Uma ferramenta personalizada de edição foi desenvolvida por forma a permitir que um level designer especifique diferentes lógicas de jogo e regras que guiem a progressão do jogo à medida que a mestria do jogador evolui. Os diferentes desafios do jogo são organizadas num grafo com transições condicionais que especificam o nível de mestria necessário para desbloquear outros desafios mais abaixo no grafo. À medida que a habilidade do jogador evolui o jogo adapta-se e fornece desafios com um nível de habilidade apropriado, com o objetivo final de aumentar a repetibilidade, diversão e cativação graças à constante adaptação do jogo à habilidade do jogador ao ultrapassar os vários desafios.

Palavras-Chave: progressão, habilidade do jogador, adaptação de níveis, geração procedimental de conteúdo, jogos

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem

Procedural content generation (PCG) has been used throughout the history of games, beginning with Rogue[1] in 1980, to automatically create game content using algorithms. The generated game content has ranged between dungeon generation (e.g. Diablo series[2]), map generation (e.g. Civilization series[3]) and weapon generation (e.g. Borderlands series[4]).

Games of today are extremely dependent on custom generated content (e.g. maps, inventory items, skins) to engage players in replaying a game. The most commercialised games are the ones who are able to provide fun and engagement for longer periods of time.

One of the features a game can have to be more successful is *replayability*, the ability to provide the players different content every time the game is played. This explains why most successful games are online multi-player (e.g. Counter-Strike[5] or Dota[6] series - FPS and MOBA genres), since a player is always encountering other players with different skill sets, or games with huge open-worlds, where a player can explore and find something new all the time (The Elder Scrolls[7] and Grand Theft Auto[8] series, or Minecraft[9]).

For single-player games, fun and engagement come with the progression of the challenges offered to the player and how these are overcome and interiorised by the player. With the exception of open-world games, most single-player games lack replayability, which decreases the amount of fun and engagement a player can retrieve from a game.

Furthermore, most games fail to present an adequate level of difficulty for any given player. Single-player games tend to have a basic difficulty setting (easy, medium, hard) and always offer the same challenges to every player playing in the same setting. This often leads to mismatches between real player ability and overall game difficulty and breaks the feeling of progression, as players fail to find appropriate challenges to their skill set. Classifying players in such a simple way is not the most adequate, because progression may vary between players which will make them experience the same game in different ways.

1.2 Hypothesis and Approach

Current games tend to treat *game flow* only regarding the challenges presented to the player, while the original definition of game flow, as developed by M. Csikszentmihalyi[10], defends two dimensions: challenge and skill. In this work, we will model the progression of a player in a game, not as a difficulty function, but as a ratio between the two dimensions of game flow, effectively bringing the skill dimension

back into the problem of modelling the progression of a player.

We propose a model for progression in video games, using some of the features provided by the game, such as mechanics and challenges, alongside with the rate of success of a player in each component. In this dissertation, we will implement this model for a side-scrolling endless-running platform game.

We theorise that this model allows for an increased level of replayability in single-player platform games and increased fun and engagement, because every consequent challenge is provided according to the current player skill within the game. Multiple game sessions, which should translate on increased skill, will provide each time more difficult challenges to the player. Independently of their skill levels, our model seeks to make the players always feel challenged and engaged with the game.

The game should adapt itself to the player and not the contrary. For this to happen, a custom editor tool was developed to allow for level designers to specify different adaption rules for the game, in order to guide the progression as the mastery of the player evolves.

With this dissertation, we hope to achieve a progression model that is robust enough to be potentially used in different types of games and that voids the need of using pre-determined difficulty settings, but modular enough to allow for level designers to specify when and how the progression of the game should change according to the skill of the player.

1.3 Contributions

This work will survey the state of the art of research on PCG and player modelling, drawing comparisons between models of game mechanics and level generation patterns and how these can be used, in conjunction with the players' expectation of fun, engagement and/or frustration to automatically generate platforming levels that exploit these feelings. We will create a model for progression in video games, in order to guide procedural content generation to create appropriate challenges to the set of skills displayed by the player, and will implement the model in the context of an endless-running platformer game. Finally, we will test our approach with users in order to validate our model.

We hope, with this work, to bring the skill dimension back into the way PCG is used in games, by autonomously generating challenges to players, according to the set of skills displayed by them, which will allow for increased replayability in single-player games and for supplying the players with challenging content adapted to them.

1.4 Organisation

In the previous sections, we presented the problem, motivation and contributions of this work.

In the following, we present a review of literature, where we explore the concepts of PCG, models of difficulty adjustment, level-generation in platformer games as sequences of challenges of different formats and complexity and the use of game features as indicators of components of the player experience. We then present our proposed model to the referred problem and we describe the implementation of the model in the context of game and an auxiliary editor tool.

We finish this dissertation by describing our evaluation methods and results, drawing our final conclusions and some future work to improve the capabilities of the model presented.

Chapter 2

Related Work

This section surveys previous work in *user modelling* and *procedural content generation* in games.

We will present several approaches on how to specify game mechanics and how these can be used as building blocks of a game level. We will also present some works of level generation in platform games and how these can be related with emotions to provide automatically generated levels which expose the players to specific ones.

We will start with an article by Daniel Cook, titled *The Chemistry of Game Design*, where the author describes how we can use a psychological model of the player to help us specify a chain of all the possible mechanics in a game. Next, we will explore the different definitions of Togelius et al. for *Procedural Content Generation* in order to contextualise and define the type of PCG used in our approach. Following that, we will present one of the first approaches of *Dynamic Difficulty Adjustment*, by Hunicke and Chapman, related to the supply and demand of inventory items in a game. Next, we will explore a work by Pedersen et al. where the authors present a model for player experience by measuring different types of data, using gameplay metrics and surveys. The following two works describe how we can use patterns and combinations of them to generate obstacles in a game, increasing the replayability and allowing for a more fit adaption of a difficulty/flow curve. The last work serves as an example for comparison of how procedurally generated content is used in commercially distributed games to provided content for a player in a platform-running game.

At the end of each referenced work we present a brief discussion where we argue, relate and discuss the pros and cons for the various presented approaches and which ones we will use and adapt for our model.

Finally, we will provide a short summary of the discussions of each work, in order to better introduce our model, which we will detail in the next section.

2.1 Player Modelling

2.1.1 *The Chemistry of Game Design* Daniel Cook

Daniel Cook[11] argues the need of building a testable model of game mechanics as a way of providing new opportunities for game balancing, original game design and the broader application of game design to other fields.

It is discussed that previous attempts have been to treat games as self-contained logical systems and presents a new approach which relies on the need of a working psychological model of the player, described as an entity that is driven to learn new skills high in perceived value, and chains of skill atoms which compose a directed graph of the game mechanics.

Player model

Skill is defined as a behaviour that the player uses to manipulate the world. These skills can be physical, such as lock-picking, or conceptual, such as navigating a map. Cook[11] defines *driven to learn* referencing the human psychological aspects of playing as a default and instinctual action for humans, which makes us engage into and learn more intricate hobbies as we grow older. We are rewarded for learning by experiencing joy, which is derived from the act of mastering knowledge, skills, tools and the ability to manipulate our environment for the better. *Perceived value* is defined as the fact that players pursue skills with high perceived value over skills with low perceived value. As humans, we instinctively engage into play, because it provides us with the opportunity to learn behaviours that might turn advantageous later in life. This means that we play already expecting the utility that might come with it and we stop playing when we fail to find this utility. Therefore, the perception of value is more important than an objective measured value.

Skill atom

Cook[11] defines *skill atom* as a self-contained atomic feedback loop, made of remixed tokens, verbs, rules, etc., which describes how the player is able to gain a new skill. Each skill atom is composed of four main elements:

- **Action:** The player performs an action;
- **Simulation:** As a consequence of the action, an ongoing simulation is updated;
- **Feedback:** Lets the player know how the simulation changed state;
- **Modelling:** The player updates their mental models based on the success of their action, depending on the feedback provided.

As the game is played, each atom may be looped several times as the player experiments the skill to see if it does something useful. When several skill atoms are linked together to form a directed graph, called a *skill chain*, one can visually represent how the players learn the usefulness of each skill in the chain. Consequently linking more and more skill atoms will build a network that describes the entire game. Cook[11] defends that a skill chain can be used to model any game, due to its general notation, and has the ability to better describe the player experience, instead of simply the mechanics of the game, which provides a more appropriate description of the important moments of the gameplay. Through the

analysis of a skill chain, it is possible to retrieve useful information regarding the state and progress of the player as they engage the game, some of them being:

- **Mastered skills:** Skills that have been recently mastered;
- **Partially mastered skills:** Skills that the player is toying with, but has not yet mastered;
- **Unexercised skills:** Skills the player has yet to attempt, when they can't perform the actions needed to understand it, making impossible the exercising or mastering of the skill;
- **Active skills:** Skills that the player is actively using. Although the player only experiences the joy of mastery once, they may choose to continually exercise an already mastered atom as a tool to advance their knowledge to help them master an atom further down the chain;
- **Burned out skills:** Skill atoms that the players have lost interest in exercising, because they failed to find any interesting use for it. When a burnout happens early on the skill chain, large portions of the player's potential experience might become inaccessible.

Discussion

Daniel Cook[11] proposes a way of modelling game mechanics using the game actions available to the player as entities that can be grouped into skill chains, showcasing the dependencies between these actions and the level of mastery of the player in each skill chain. This approach is useful as a tool to describe the full spectrum of game mechanics and how these are linked to create more complex actions. The work performed by Daniel Cook[11] serves as inspiration to our work. We intend to abstract the skill chain / tree technique and to relate it with the different features of a game: mechanics, challenges and paces. The evaluation for the mastery will also be revised in order to include the broader types of features we will use.

2.1.2 *AI for Dynamic Difficulty Adjustment in Games*
Robin Hunicke and Vernell Chapman

Hunicke and Chapman[12] explore the computation and design requirements for a *dynamic difficulty adjustment* (DDA) system. The described system manipulates the stock of items in-game, being stated that the abundance or scarcity of inventory items directly impacts the player's experience. Adjusting the supply and demand of game inventory, the authors, using a custom developed tool named *Hamlet*, were able to control the overall game difficulty.

The game was developed using Valve's *Half Life* game engine to develop a custom environment, where the authors then embedded the *Hamlet* system, a set of libraries used to monitor game statistics, defining and executing policies.

Figure 2.1: Example of a custom environment using Hamlet

Hamlet dynamically evaluates the difficulty of the obstacles given to the player based on their performance, while the game is running. Using this data, Hamlet is able to estimate a player's state in the future and, when it predicts an undesirable state, game settings are adjusted as necessary. The *flow model*, developed by M. Csikszentmihalyi[10], is used to estimate when and how to intervene, since it is not desired to present the game to the players with states too challenging or too easy.

Observing trends on how a player's inventory is used, through established metrics for assessing statistical data (e.g. damage taken and usage of health power-ups over time) the system is able to find potential shortfalls or abundance of inventory and adjust the game accordingly. These adjustments actions may be reactive, when the adjusted elements are directly visible to the player (e.g. accuracy of enemies currently attacking the player), or proactive, when applied to elements not currently on screen. Each given action has an associated cost, enabling the possibility of gradual but effective adjustment, and is calculated using metrics directly related with the overall performance of the player in the game, in the level, in the current encounter and how many times the system has assisted the player.

Different sets of actions and costs can be combined in order to create adjustment policies, which have the ability to control the supply and demand of inventory goods, according to the expected player experience. Hunicke and Chapman[12] separate these policies into two sets: *comfort zone* and *discomfort zone*. The first is tweaked to allow for steady demand and predictable supply of goods, while the latter aims to challenge more experienced players, characterised by goal-oriented supply and sporadic high demand. Hamlet intervenes iteratively. Using trial and error to gradually accommodate the game to the player, Hamlet will take into account if the previous actions were successful or not in order to decide when to intervene again.

Discussion

Hunicke and Chapman[12] are considered some of the catalyst researchers regarding adaptive difficulty in video games. In the referenced work, they propose a framework that evaluates the inventory usage by the player and manipulates the supply and demand of each item. The work of these authors serve as motivation and inspiration for the work we pretend to do. We intend to adapt this work to our own, supplying the demands of in-game obstacles according to the player's skill. We will explore proactive adjustments to the procedurally generated playthrough, taking the overall performance of the player using specific mechanics to overcome specific challenges at a specific pace. We hope to develop a progression model with the ability of generating different playthroughs of the same difficulty, for players of the same skill levels, and be able to offer each player different playing experiences related to their own playing style, analogous to the *comfort* and *discomfort zones* discussed by Hunicke and Chapman[12].

2.1.3 *Modeling Player Experience in Super Mario Bros*
Chris Pedersen, Julian Togelius and Georgios N. Yannakakis

Pedersen et al.[13] investigate the relationship between level design parameters of platform games, individual playtesting characteristics and player experience and propose a model of player experience, derived from gameplay interaction, which can be used as a fitness function for game content generation.

This work was tested in a modified version of Super Mario Bros. - Infinite Mario Bros.[14] - which has automatic random generation of levels, further tweaked to use a few selected game level parameters to be introduced in all generated levels. The model was trained using different types of data:

- **Controllable features:** Parameters used for level generation, which affect the type and difficulty of the levels. The features used were number of gaps, average width of gaps, spatial diversity of gaps and number of direction switches, and were chosen by consulting game design experts, because these are common to almost all platform games. The combination of these features allow for 2^4 different variants of the game;

- **Gameplay features:** Features which depend on the player's skill and playing style, extracted from playing data logged during gameplay. These characteristics include completion time, time spent on miscellaneous tasks, information on collected items, killed enemies, information on how the player died and were chosen because they cover a large amount of the game playing behaviour dynamics;

- **Reported player experience:** A game survey was asked to each player after two game levels, each with different controllable features with the objective of making the players rank the games in order of emotional preference.

The collected data was used to approximate a function which maps controllable features and gameplay features to the emotion preferences of the player, using *neuro-evolutionary preference learning*. The mapping between reported emotions and input data is approximated with a neural network. The learning is achieved through evolution, using a genetic algorithm implemented with a fitness function that measures the difference between the player's emotional preferences and the relative magnitude of the neural network output.

The model should depend on as few features as possible, which makes it easier to analyse and to extend to other games. The feature selection is used to find the feature subset with the most accurate player model and can be performed using *n Best Individual Feature Selection* (nBest), the *Sequential Forward Selection* (SFS) or the *Perceptron Feature Selection* (PFS) schemes. The authors state that all three schemes are incomplete, because they are not guaranteed to find the optimal feature set, due to being variants of hill-climbing.

Pedersen et al.[13] discuss the correlations between playing order, controlled feature, and gameplay features and the reported emotions of fun, challenge and frustration:

- **Fun:** The seven features correlated with fun suggest that players enjoy a fast-paced game with near-constant progress, lots of movements, lots of enemy killing and lots of pickups. This definition agrees with the original concept of flow[10];
- **Challenge:** Eighteen features are significantly correlated with challenge, which shows how much easier it is to predict challenge than to predict fun. The correlated features also show that almost none of them are shared with fun;
- **Frustration:** Twenty-eight features are significantly correlated with frustration. Almost all of the top ten correlated features are shared with challenge. The correlated features suggest that a frustrated player is more likely to spend time standing still and thinking about how to overcome the next obstacle, and is less likely to grab pick-ups. Frustration seems to be very well predicted from not winning the level.

The authors conclude that this model has good predictors for all three emotions, but fun and challenge are still not as well predicted as frustration, which points to the need for better models and/or features. The results gathered shall be able to be applied, to some extent, to any platform game.

Discussion

Pedersen et al.[13] propose a model to generate platform levels from players' experience when confronted with specific types of level design, in a modified Super Mario Bros. game. The model proposed by the authors was trained using different types of game features. Since our tool allows a level designer to use the first two types of features to guide the progression of the playthrough and that the level designer can perform tests with users to evaluate each implemented solution, we can determine that all of three types of game features described in this model will be used to determine the player progression in our game, in order to procedural generate adequate playthroughs for the player.

2.2 Procedural Content Generation

2.2.1 *What is Procedural Content Generation? Mario on the borderline*
Julian Togelius, E. Kastbjerg, D. Schedl and G. N. Yannakakis

Togelius et al.[15] clarify the definition of *procedural content generation* (PCG), by comparing it with other forms of content generation in games. PCG in games is usually referred as automatically creation of game content using algorithms, but this definition is far from precise. The authors compare various forms of content generation to create a better definition of PCG, and describe two versions of a level generation method that does content generation in a not typically way, in order to test PCG as it has been newly defined.

Forms of content generation

Starting with *offline player-created content*, Togelius et al.[15] state this form of content generation, which uses content editors, is not PCG, because the content is created by human players. However, there are PCG algorithms that support significant user input, which are used as a cooperative tool to create content. With *online player-created content*, which uses in-game building features, the same thing happens. If a human has any kind of control on how the content is generated and intends to create content in the game, then it is not procedural content generation.

PCG is also not random, in the sense that a level generator doesn't place each element randomly without any kind of structure. This would make the content unplayable. Most of the existing content generators include assertions and create contents adhering to a set of rules, which ensures playability and promotes engagement. A better interpretation of randomness in PCG is that the generators include strong constraints on what kinds of content can be generated, but the content can be varied using a pseudo-random process, within these constraints, which happens in most of the existing PCG implementations.

The last compared form of PCG involves the adaptation of the content to the player's previous behaviours. This is called *adaptive* or *player-driven* PCG. This type of PCG is currently in active research in academia and is absent in commercial games. Some of the features of adaptive PCG is the adjustment of the difficulty of the newly generated content, based on an estimated player skill, or the generation of more content similar to content the player seems to have liked in the past. It ceases to be considered PCG when the player can easily predict the generated content on the next iteration of adjustment.

Togelius et al.[15] present two versions of a level generation model based on previous work by the same authors. The updated version of the level generation has offline and online versions, related to the different types of created content, as detailed above. The offline version of the level generator begins by providing a randomly generated level using the standard level generator of Infinite Mario Bros[14]. Although being an easy level, this approach allows for diversity on how the players play the level. Each player action of the level is recorded. The next level starts with a copy of the previous level and is modified according to the actions of the player, based on a set of rules that map a player action to a modification of the level at the position where the action was performed. The online version of the level generator uses the same rules as the offline version with two additionally rules that map pickups collected to the creation of enemies and killing of enemies to the creation of coins on the left or right of the position. The major difference to the offline is that, instead of generating a new level after finishing the current one, the level modifications are performed immediately after the action was taken.

The authors state that both versions of the level generator allow the player to predict and intentionally shape how the modifications to the level will be performed to their taste, although on the offline it meant the player had incredible memory capacity which makes this practically irrelevant. On the online version, because the player has the ability to create an interesting or simple plane level, it is appropriate to think of this version of the level generator as an interactive level editor and the generated levels as player-generated content.

Togelius et al.[15] redefine PCG as *"(...) the algorithmic creation of game content with limited or indirect user input. Note that this definition does not contain the words "random" and "adaptive", as PCG methods could be both, either or none"*.

Discussion

Togelius et al.[15] contextualise the different types of content generation and relate them with *procedural content generation* (PCG). In our work, we expect to create a tool able to generate content based on indirect user input (in this case, the player's skill). Togelius et al.[15] note that there are several types of content generation based on user input that is not labelled as procedural, such as using created content as blocks for creating levels.

Although in this work we intend to use challenges hand-crafted by us, we assume that these challenges can be procedurally generated. The focus of our work, as defined by Togelius et al.[15], is *adaptive* or *player-driven* PCG. In order to respect the definition of the authors, we must be able to generate content that the players cannot predict.

Togelius et al.[15] also refer that some types of level generators can't be qualified as procedural, if their main attribute is randomness. In our work, we intend to use the input of the player's skill as a mean of evaluating the proper challenges and paces to be used in the generation process. According to Togelius et al.[15], this method can be labelled as procedural, since there are strong constraints on what kinds of features can be placed, while allowing each feature to be pseudo-randomly chosen based on the available pool.

2.2.2 *Procedural Level Design for Platform Games* Kate Compton and Michael Mateas

Compton and Mateas[16] propose a four-layer hierarchy to represent platform game levels with a focus on repetition, rhythm and connectivity, as a mean to extend gameplay and replayability of platform games. The authors state the difficulty of successful level generation for platform games, due to the fact that even very small changes to a level may turn it into a physically impossible challenge for players. The proposed solution models the hierarchical elements used to build a level and the relationships between different platform levels.

Hierarchical elements

Using the *flow* definition of Csikszentmihalyi[10], Compton and Mateas[16] defend that level design in platform games relies heavily on rhythmic actions, which help the player reach a state of heightened concentration. Appropriate placement of obstacles result in a rhythmic sequence of player movements, which ease the player to time their actions. Varying the repetition of elements, while maintaining their rhythmic positioning helps the player feel engaged for a longer period of time, even using few game components.

A component is the basic unit with which we can build a level. By grouping several components into a longer sequence, we build a pattern. Patterns are responsible for providing the rhythmic feel to a level and are divided into four types:

- **Basic patterns:** One or more of the same component;
- **Complex patterns:** A basic pattern, but with some placement tweaks, regarding the distance between obstacles;
- **Compound patterns:** Two basic patterns are alternated in between. This type of pattern introduces a rhythmic pattern at a higher level of abstraction;
- **Composite patterns:** A sequence of two components, which require a coordinated action. To be beaten, the player must know how to overcome each component, individually, and synthesise it into one continuous action.

The usage of patterns allow for constructing strictly linear sequences. A higher layer, called *cell*, arises with need of building non-linear structures. A cell is an encapsulation of a pattern, where the only important property is if the end of a pattern allows the player to reach the beginning of the other. Having only this constraint, it is possible to create a platform level with varied types of branching and parallel paths, only by using cells, called cell structures.

Level generation

For generating a level, Compton and Mateas[16] modelled an algorithm that uses a context-free grammar, using the various types of hierarchical elements as symbols. Using this method, generating a level is as simple as generating a string. Each level begins with a cell structure, which can be sub-divided into more cell structures. Each cell is then given a pattern, which can be also sub-divided. For each pattern used in-game, the system brute-forces all the possible patterns from a sub-set of component types and number of components, where the optimal one is chosen using a simple hill-climbing algorithm towards the target difficulty.

The difficulty of a generated level is calculated using the edges of each component used. The system takes into account the distance between components, estimating the temporal window the player can use to perform a successful jump from one to another. This way the difficulty of a jump can be approximated.

Discussion

Compton and Mateas'[16] work discusses *procedural level generation* in platform games as a sequence of automatically generated building blocks of different sizes. These methods allow for replayability of the game, due to the recycle of the different challenges that allow new compositions of challenges with a lower feeling of repetition. This approach will also allow us to grant a feeling of rhythm to the game, an important characteristic in games of the genre of endless runners. For the level generation, we will use a mix of the available paces, mechanics and challenges in the game, to provide compositions of challenges with a difficulty approximated to the current level of mastery of the player.

2.2.3 *A Multi-level Level Generator* Steve Dahlskog and Julian Togelius

Dahlskog and Togelius[17] extend a previously devised methodology for pattern-based level generation, using Super Mario Bros.[18], which consists on a bottom-up approach that uses patterns as building blocks. The levels generated by the previous method lacked a sense of progression and unity. The proposed model introduces a new macroscopic structure, increasing the total number of possible building blocks types:

- **Micro-patterns:** One of the possible vertical slices of the level, one tile wide;
- **Meso-patterns:** A sequence of micro-patterns, usually with the length of the screen. A meso-pattern is helpful to understand the content of a game;
- **Macro-patterns** A group of meso-patterns and can be two or more screens wide. On a macro-pattern it is possible to visualise the relation between different meso-patterns and to provide a level designer with a better-controlled difficulty curve.

The search-based approach for level generation uses a genetic algorithm with single-point mutation and one-point crossover operators. The size of each population is the same as the length of the desired level. At each generation, the half of the population members with the lowest fitness are discarded, while the remaining half are parents for breeding pairwise. The generated offspring is also subject to mutation. In the authors' previous work, their mutation operator consisted on exchanging a micro-pattern for another. A new improved operator is presented, which replaces a sequence of 5 micro-patterns on a random position.

The original fitness function, based on string search, measures the presence and order of patterns. Each level is assigned a fitness value based on the presence of specific sub-strings representing known meso-patterns. As each sub-string may vary in length and complexity, some patterns are harder to find in the solution space than others. It is then needed to tweak the fitness function to the desired outcome. With the introduction of the macro-pattern, Dahlskog and Togelius[17] defined a fitness function which rewards sequences of meso-patterns that represent known macro-patterns. If an ordered sequence of meso-patterns correspond to a macro-patterns, then the sequence is chosen for breeding.

Both the old fitness function that promotes meso-patterns and the new fitness function that promotes macro-patterns generate visual appealing levels. The updated function generates levels with more large-scale structures, but executes up-to ten times slower.

Discussion

Dahlskog and Togelius'[17] work discusses pattern-based level generation in platform games, using structures of different types and sizes as building blocks of a level. The presented method, which is an extension of a previous work by the same authors, solve the problem of progression and unity by using a new macroscopic structure which allows for a better-controlled difficulty curve.

In our work, we intend to use different types of challenges, which we assume can be automatically generated, to construct a game level. We will assume that micro- and meso-patterns to be our challenges. Macro-patterns will be generated by concatenating a number of different challenges, with pre-determined distances between each one of them. Each one of these distances will be calculated from the levels of mastery associated with each one of the available paces provided by our game.

2.2.4 *Procedural Content Generation in Gravity Guy* Nuno Monteiro (Miniclip/Bica Studios)

Regarding commercially distributed games, it has been difficult to obtain literature relative to how procedural content generation is used from a model of the player to generate adapted content.

Nuno Monteiro lectured a workshop class regarding the development of platform-runners using Cocos2D for iOS devices, for the *Technology for Games and Simulation*[19] university course at IST.

In order to showcase some *Do's* and *Dont's* in platform-runners, Monteiro used Miniclip's Gravity Guy[20] as the main example, throughout the workshop.

Gravity Guy[20] is an arcade and a side-scrolling game in which the player controls Gravity Guy by tapping the screen (or clicking or pressing the space bar on the computer versions) to switch gravity. The objective in this game is to run as far as possible while avoiding obstacles than can trap him (and also be killed by a police officer), falling or flying off the screen.

Regarding the procedurally generated levels in Gravity Guy[20], Monteiro stated that the Miniclip's approach to generate levels with different difficulties follows an industry standard to the endless-runner genre and is defined by the player speed, distance between platform sections and an overall difficulty value, related to the distance travelled by the in-game character during the level gameplay. Each level is then generated by concatenation of sections, created in advance by the developers and categorised in difficulty pools of sections. When a new section needs to be appended, the level generator takes the three variables into account to calculate the difficulty pool from where to grab the next section and the section is selected randomly from that pool.

Discussion

Gravity Guy[20] served as an inspiration for the development of our custom platformer testbed game, due to being made by Miniclip, one of the partners of the Technology for Games and Simulation[19] course. Being both games of similar genres (endless-running platformer), we intend to use Gravity Guy[20] as an industry standard comparison tool regarding how the level generation is done and how the difficulty of the levels is adapted related to the expected value by the player, according to their skill. We hope to conclude that our is method of difficulty adaptation and player progression is as viable as the one performed currently by the industry, regarding how the levels are generated and the game challenges are offered to the player, while offering a better replayability value.

2.3 Discussion Overview

We will build our model based on the work of Daniel Cook[11]. We will model the different game features (i.e. mechanics, challenges and paces) and measure the level of mastery of the player for each one of them. The different game features will be arranged in a directed graph and the transitions between nodes will be made when the mastery of the player is within a specified threshold.

Through the work of Togelius et al.[15], we were able to classify our work as *adaptive or player-driven PCG*, because we will use the players' skill to enable mechanics, challenges and paces that can appear in the game.

Hunicke and Chapman[12] presented a *Dynamic Difficulty Adjustment* approach for a FPS game, adapting the difficulty of the game by manipulating the supply and demand of inventory items. Although not directly related, it still served as an inspiration to our work.

The work of Pedersen et al.[13] proposes a model to generate platform levels from players' experience when confronted with specific types of level design. This model was trained using the feedback of hundreds of players to specify which features in the game provided the best experience for the players. This worked allowed us to understand which features our game should allow to be tweaked by the level designers using our editor.

The following two works, by Compton and Mateas[16] and Dahlskog and Togelius[17], describe how different lengths and combinations of patterns can model the difficulty and replayability of a game. We intend to use these notions of pattern / meta-pattern to define how the different challenges and paces will appear in our game.

The final presented work, by Nuno Monteiro[20], was the inspiration for the development of the game in this dissertation. Being a professional endless-running game, we intend to use this game as the industry standard of how procedural level generation is done in this type of games and to compare it with our approach.

Chapter 3

Progression Model

Our model aims to evaluate the player's skill level with the mechanics, challenges and paces of a game in order to provide mechanics, challenges and pace adequate to the skill of the player. The progression model is based on the work of Daniel Cook[11].

We revised the original definition of *skill atoms* to include challenges and paces provided by the game, in addition to the mechanics (or skills) the player can execute. In the original work, the author only considered skill trees.

To better understand the differences between the different types of atoms, we need to define the following concepts:

- Mechanic: a skill that the player can execute in order to overcome a challenge;
  (examples: jumping, shooting, opening a door)
- Challenge: an obstacle that the game provides to the player;
  (examples: ladder, powerful enemy, explosive barrel)
- Pace: the rate at which the game will provide challenges to the player.
  (examples: 1 challenge per second, 3 seconds after each challenge)
- Mastery: the level of proficiency and dexterity that the player has related to a specific mechanic, challenge or pace.
  (examples: uninitiated, mastered)

We extended Cook's[11] levels of mastery to include an additional *frustrated* level and revised the original definitions to better fit their usage with our different types of atoms:

- Uninitiated: atoms that never appeared to the player;
- Initiated: atoms that appeared to the player but still have a low level of mastery;
- Partially Mastered: atoms that the player is currently toying with, but has not yet mastered;
- Mastered: atoms that appeared several times and were successful overcome;
- Burned Out: atoms that the players have lost interest in exercising, because they appeared an elevated number of times and were almost always successful overcome;
- Frustrated: atoms that the players have lost interest in exercising, because they appeared an elevated number of times but were rarely successful overcome.

At the start of the game, every atom starts with the *Uninitiated* level of mastery. Every time an atom appears for the first time its internal value changes to the *Initiated* level of mastery. While the game is played, the internal value for the atom's mastery may change whenever the player is consecutively successful or not. The following figure represents the flow chart for the various levels of mastery.

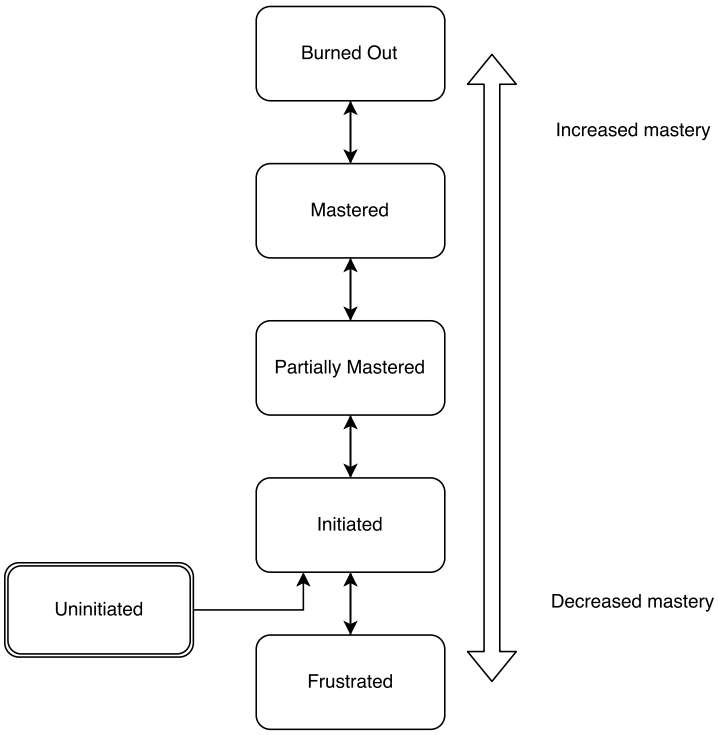


Figure 3.1: Flow chart of the different levels of mastery

Since we have different types of atoms, the way their masteries are calculated also differ: the mastery of a *challenge* takes into account how many times the player was successful or not in overcoming it; the mastery of a *mechanic* is calculated based on whether that specific mechanic was used to try to overcome a challenge and if it was successful or not; the mastery of a *pace* evaluates if the player was successful or not in overcoming a group of obstacles with that specific pace.

Having defined how the different levels of mastery are related to an atom, we can now describe how these can be put together to guide the game to adapt its difficulty depending on the skill level of the player. We revised the original definition of Cook's[11] *skill chains* to use our mechanics, challenges and paces atoms.

In our graph, we will assume that an active atom means that what it represents may happen in the game. We must then ensure that at least one of each type of atoms (i.e. mechanic, challenge and pace) must be active at the same time. At any point in time there must be at least one active mechanic, challenge and pace. We can then use transitions to define when the next atoms will become active, based on the level of mastery of the current active atoms.

When the game starts, the graph will have only the initial atoms active. While the game is being played and the mastery of each atom changes, the child atoms become active while its transition rule, based on the mastery of the parent atoms, is valid. Once these transition rules become invalid, the child atoms (and their respective children) will become inactive.

The following figure represents a sub-graph unlocking several mechanics. Challenge and Pace atoms are omitted for convenience.

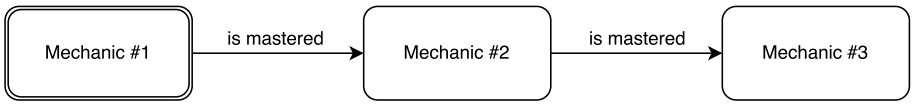


Figure 3.2: Example sub-graph enabling several mechanics sequentially

The game starts with only *Mechanic #1* active. After several challenges successful overcome with that mechanic, its level of mastery eventually reaches mastered. This will then activate Mechanic #2, which can now be used in conjunction with the first mechanic. After a few more challenges successful overcome with this new mechanic, its level of mastery eventually reaches mastered as well. Mechanic #3 is now active and can be used.

It is important to note that Mechanic #3 will only remain active as long as both Mechanic #1 and Mechanic #2 are active as well. If for some reason the level of mastery for the Mechanic #1 decreases and stops being mastered, both Mechanic #2 and #3 will become inactive. This example, although very simplistic, allows us to understand the complexity and possibilities to define the progression rules for the game.

We can expand the previous example in order to integrate challenge atoms and add more complex logic for the transitions. The following figure represents a sub-graph unlocking several mechanics and challenges. Pace atoms are omitted for convenience.

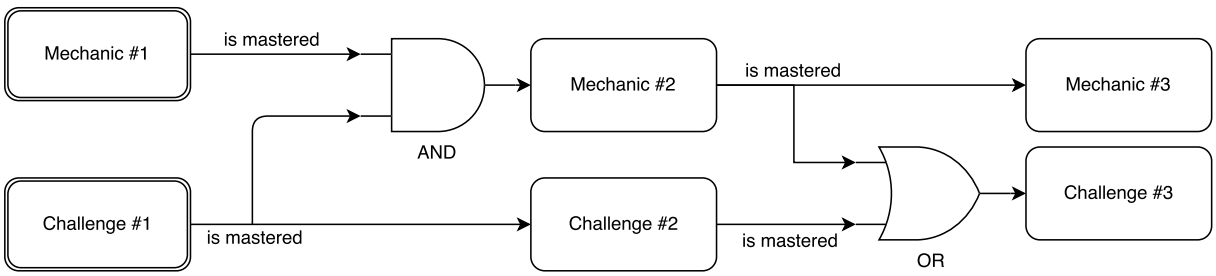


Figure 3.3: Example sub-graph enabling several mechanics and challenges in parallel

In this example, when Challenge #1 is mastered then Challenge #2 becomes active. To activate Mechanic #2 both Mechanic #1 and Challenge #1 must be mastered. To activate Challenge #3 at least one of Mechanic #2 or Challenge #2 needs to be mastered.

Allowing these type of transitions exponentially increases the possibilities for different sub-graphs and subsequently different progression rules for the game.

In addition to provide several mechanics and challenges to the player, our game needs to measure the level of mastery of the player. For this to happen, the game needs to track the successes and failures of the player when overcoming challenges with specific mechanics.

Each mechanic, challenge and pace has its own window of attempts that registers the last *N* attempts related to that feature. Each one of these attempts has a different weight contribution in the totality of the window. The window of attempts is always ordered from oldest to most recent one. The weights for each entry in the window of attempts is used to specify which attempts will have a biggest impact in the calculation of the score. We assume that a success has a value of 1, while a failure has a value of 0. We can then calculate the average success of a feature using the following formula:

$$AverageSuccess = \frac{\sum_{a=0}^{nAttempts-1} attempts[a] * weights[a]}{\sum_{a=0}^{nAttempts-1} weights[a]} \tag{3.1}$$

This always yields a result between 0 and 1, that we can then map, in conjunction with the number of attempts in the window, to a usable level of mastery listed in figure 3.1, as in the following formula:

$$Mastery = \begin{cases} BurnedOut, score = 1 \wedge nAttempts \geq windowSize \\ Mastered, score \geq 0.9 \wedge nAttempts > windowSize * 0.8 \\ PartiallyMastered, score \geq 0.5 \wedge nAttempts > windowSize * 0.6 \\ Frustrated, score \leq 0.1 \wedge nAttempts > windowSize * 0.8 \\ Initiated, nAttempts > 0 \\ Uninitiated, nAttempts = 0 \end{cases} \tag{3.2}$$

Chapter 4

Implementation

4.1 Node Editor

A custom graph editor was developed in Unity, based on the work of Levin ”Seneral” G.[21], to facilitate the construction of different graphs. We can then connect this tool to a custom Unity game (also developed by us) and watch in real time how the mastery for the different atoms evolve over time and what atoms are active at each moment, which tells us how the game is adapting itself to the progress of the player.

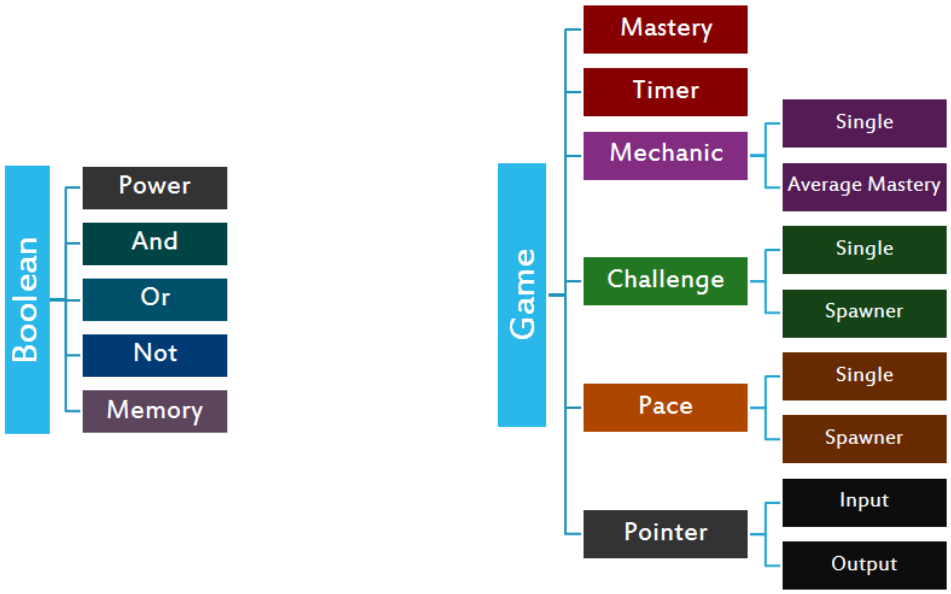
The following figure shows our editor in action displaying an entire graph of atoms and which of them are active in the game.

Figure 4.1: Example of a complete graph

Our editor supports 10 different types of game nodes (or atoms) and 5 helper nodes to implement more complicated logic for the transitions.

Each one of these nodes have an internal Boolean value which is sent to the nodes connected to their output. Most nodes will directly output the value of its input (and internal value), while the remaining of them have a more complicated internal logic and conditions that need to be resolved to output a Boolean value of *True*.

The following figures display the organisation of the node selection menu for our 15 different nodes:



(a) Boolean Nodes

(b) Game nodes

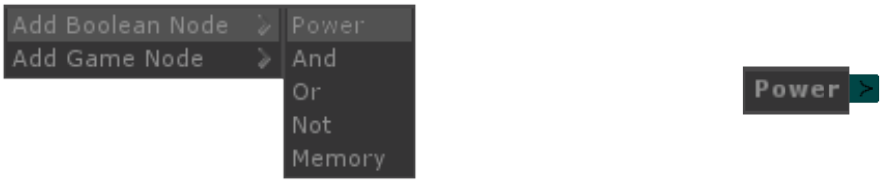
Figure 4.2: Hierarchy of the node selection menu

4.1.1 Boolean Nodes

Boolean nodes are helper nodes that allow us to implement a more complicated logic for transitions between game nodes. These are simple implementations of the following Boolean operators: True, And, Or, Not and D Flip-Flop.

Power Node

The *Power Node* is used as a node that always outputs the Boolean value *True*. This means that any node connected to the output of this one will start active. It is usually used to define which game nodes will be active at the start of the game.



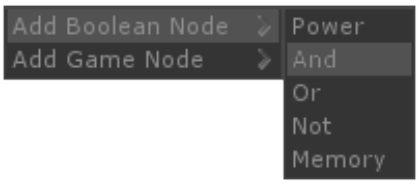
(a) Menu representation

(b) Visual representation

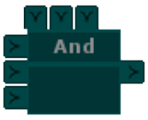
Figure 4.3: Representation of the Power node in the editor

And Node

The *And Node* is a representation of the Boolean operator *And*. It only outputs *True* when all of its (connected) inputs are *True*.



(a) Menu representation

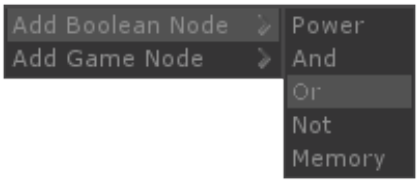


(b) Visual representation

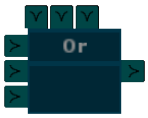
Figure 4.4: Representation of the And node in the editor

Or Node

The *Or Node* is a representation of the Boolean operator *Or*. It only outputs *True* when one of its (connected) inputs is *True*.



(a) Menu representation

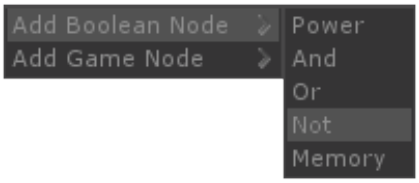


(b) Visual representation

Figure 4.5: Representation of the Or node in the editor

Not Node

The *Not Node* is a representation of the Boolean operator *Not*. It outputs the inverse of its input, meaning that it only outputs *True* when its inputs is *False*.



(a) Menu representation

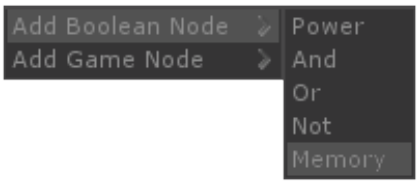
(b) Visual representation

Figure 4.6: Representation of the Not node in the editor

Memory Node

The *Memory Node* is a representation of the Boolean logic *D Flip-Flop*. Once its input is *True* it will always output *True* thereafter, even if its input changes to *False*.

As an example, this is extremely useful if we want to permanently enable a new challenge when the player reaches a certain level of skill. Even if the level of skill of the player decreases drastically, we will still be able to use that



(a) Menu representation

(b) Visual representation

Figure 4.7: Representation of the Memory node in the editor

4.1.2 Game Nodes

Game nodes represent the various atoms described in the Progression Model chapter as well as some utility nodes to measure masteries and to control the paces and challenges that will appear in the game. These nodes have a direct impact in how the game plays and adapts to the player. Only the active game nodes (which input is True) are taken into account when generating content in the game for the player. When visualising in real-time the graph with our tool, all the active game nodes are easily identified by having a green outline all around them.

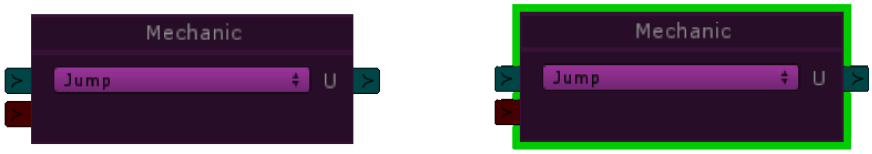


Figure 4.8: Comparison between a disabled and an enabled node

Blocker Input

All the mentioned game nodes that directly affect what can happen in-game (Mechanic node, both Challenge Nodes and both Pace nodes) have an extra input called *blocker input*. When a True value is connected to that input, its function is to forcefully disable the node but to allow its internal value to propagate to the next nodes.

As an example, we would use this feature if we wanted to disable a mechanic when a new one becomes active. For that to happen, we would only need to connect the output of the new mechanic to the blocker input of the old one.

The blocker input can be easily identified by its red colour.

Mastery Node

As described in the previous chapter, each atom (mechanic, challenge or pace) has an internal property which measures the level of mastery of the player regarding to that atom. It is then needed some sort of control node that measures when the mastery of an atom is above or below a threshold. That is the objective of the *Mastery Node*. With this node we can know when the player is sufficiently skilled playing our game and make the graph react according to it, enabling or disabling features of the game to suit the difficulty of the game to the player.

The Mastery Node has two parameters which define the threshold for when its output shall be True. The first parameter specifies the relational operator (i.e. $<$, = or $>$) and the second parameter defines the level of mastery for the threshold (i.e. one of the levels of mastery defined in Figure 3.1). For convenience, we assume that *Uninitiated* is between *Frustrated* and *Initiated*.

This node can only be used when its input is one of the Mechanic, Challenge or Pace nodes.

(a) Menu representation (b) Visual representation

Figure 4.9: Representation of the Mastery node in the editor

Timer Node

The *Timer Node* is used as a timer to activate the next node after the specified amount of seconds. If the *Reset on disable* toggle is checked, the timer resets to the initial value every time the node becomes inactive. If it is unchecked, the timer pauses when the node is inactive and resumes when active.

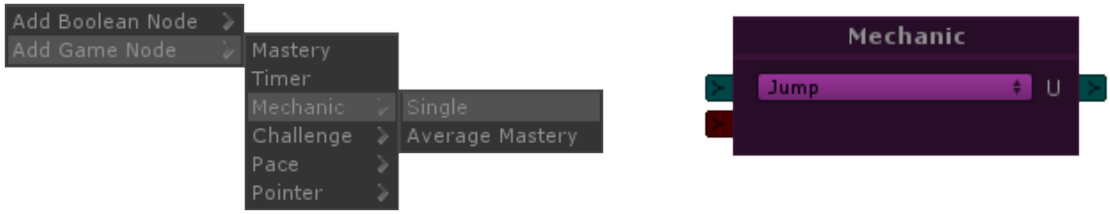
This type of node is useful when we want to delay the activation of the following nodes. As an example, we may wish to enable a new challenge only if a player can keep the skill level for at least a specified amount of time.

(a) Menu representation (b) Visual representation

Figure 4.10: Representation of the Timer node in the editor

Mechanic Nodes

This type of node is used to enable or disable the different game mechanics provided by the game that the player can use. Its only customisable parameter is which mechanic that specific node affects. Only when a *Mechanic Node* is active does its specified mechanic become available to the player. This is useful if we want to introduce different mechanics along the playthrough, instead of allowing all the mechanics since the beginning.



(a) Menu representation (b) Visual representation

Figure 4.11: Representation of the Mechanic node in the editor

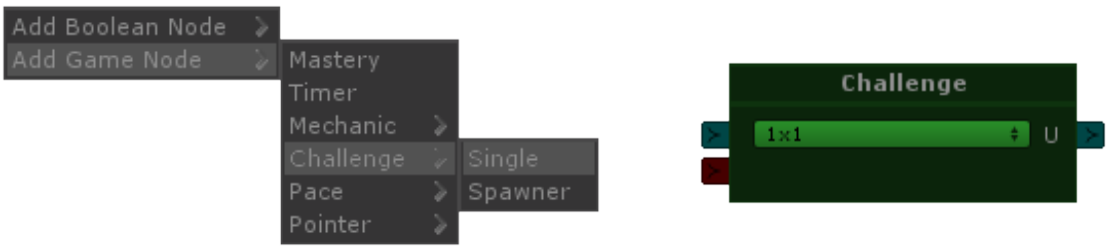
The following *Mechanic Average Mastery* works as an utility node and allows us to calculate the weighted average for the masteries of the specified mechanics. The column *mW* is used to define the different weights for the mechanics, in order to specify which ones will contribute more to the calculation of the average mastery.

(a) Menu representation (b) Visual representation

Figure 4.12: Representation of the Mechanic Average Mastery node in the editor

Challenge Nodes

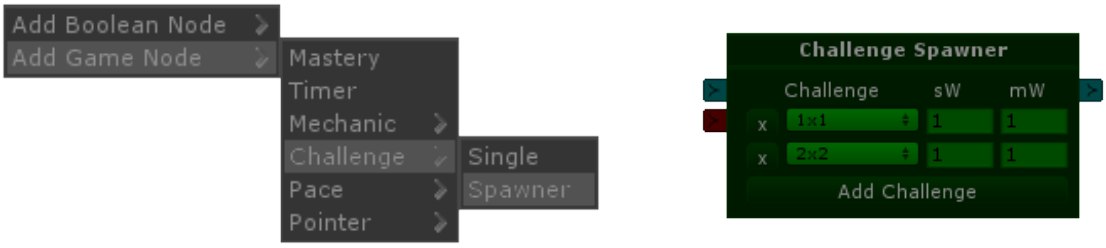
The *Challenge Node* is a representation of the *challenge atom* described in the previous chapter. It controls if a specific challenge of the game will be available in-game.



(a) Menu representation (b) Visual representation

Figure 4.13: Representation of the Challenge node in the editor

The previous node does not guarantee by itself that a challenge will appear in-game. For that to happen we also need this *Challenge Spawner Node*. It works similarly to the *Mechanic Average Mastery Node* in the way that the *mW* column is used to define the weights for the calculation of the average mastery. Its additional *sW* column is used to define which of the listed challenge will have a greater chance to appear in-game. If a *Challenge Node* specifying one of the challenges is inactive we assume that its *sW* value is 0, so it will not appear in-game.

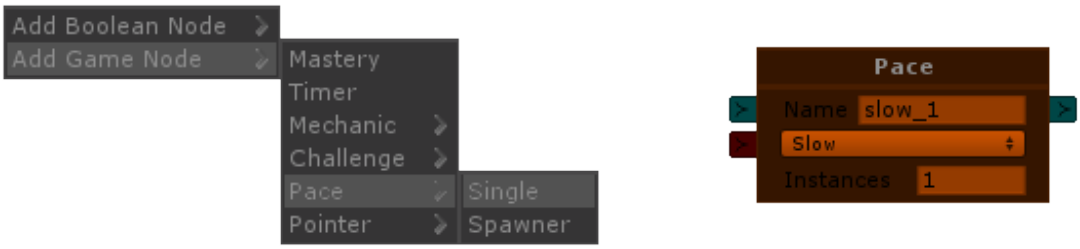


(a) Menu representation (b) Visual representation

Figure 4.14: Representation of the Challenge Spawner node in the editor

Pace Nodes

Similarly to the previous described *Mechanic Node* and *Challenge Node*, the *Pace Node* is the representation of a *pace atom*. It controls the rate at which the challenges appear in-game. A name must be specified in order to select the correct pace in the complimentary *Pace Spawner Node*. The other two parameters specify the rate of appearance of the challenges and the amount of challenges that will certainly appear at that rate.



(a) Menu representation (b) Visual representation

Figure 4.15: Representation of the Pace node in the editor

The complimentary *Pace Spawner Node* is similar to the *Challenge Spawner Node*. It is used to specify different paces with different spawn and mastery weights.

(a) Menu representation (b) Visual representation

Figure 4.16: Representation of the Pace Spawner node in the editor

Pointer Nodes

Pointer Nodes are utility nodes used to simplify the visualisation of the graph. Instead of connecting the same node to other nodes distant from it, creating this way a lot of overlapping transitions and visual noise, we can use pointer nodes to act as a variable holding the result of the output of any node.

Pointer Input Nodes have as an input the Boolean value of the node connected to them. If we then specify a name for the node we can reference that value by using a complementary *Pointer Output Node*.

(a) Menu representation (b) Visual representation

Figure 4.17: Representation of the Pointer Input node in the editor

A *Pointer Output Node* has only one parameter that is the name of one of the other Pointer Input nodes. When that happens, the node will then output the value of its complementary input node.

(a) Menu representation (b) Visual representation

Figure 4.18: Representation of the Pointer Output node in the editor

4.1.3 Content Generation

As previously stated, spawner nodes are responsible for generating content for the game. For that to happen, at least one of each type (challenge and pace) of spawner nodes must be active. Furthermore, at least one of the entries in each spawner node must be active as well, meaning that the respective challenge or pace atom must be active, and his spawning weight (sW) must be greater than 0.

It may happen that more than one of each spawner type is active at the same time. In this case, the editor will randomly choose with equal probability one of the active spawner nodes of each type and then randomly choose, according to the different weights, which pace and challenge to generate.

4.2 Testbed Game

In order to implement and test the model of progression defined in this work, we implemented a custom game using Unity which directly connects to our node editor. This game is a side-scrolling endless-running platform game which features some simplistic mechanics and a short library of challenges.

The players plays as a white cube that must overcome all the challenges thrown at them with the objective of surviving for the longest period of time possible.

The following figure shows a current screenshot of the game in action.

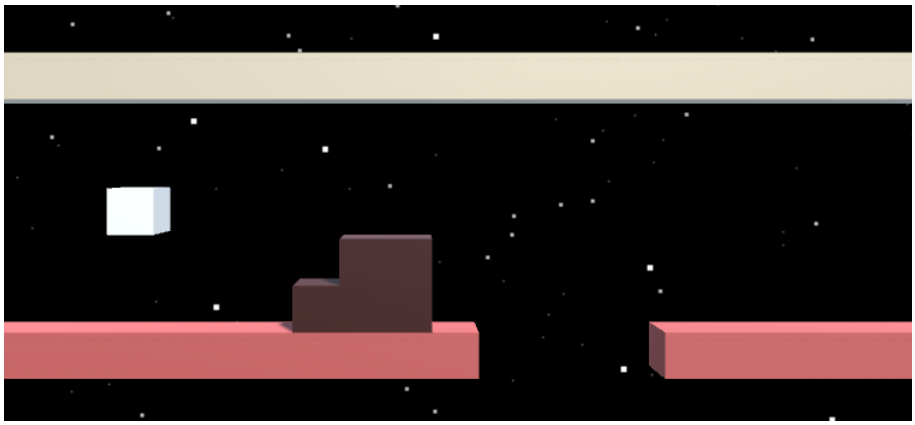


Figure 4.19: Current appearance of our platform game

4.2.1 Mechanics and Challenges

There are two possible ways of losing in our game. They are colliding with an obstacle or falling through the ground.

In this section we will present all the different mechanics and challenges implemented in our game that can be directly manipulated using the graphs built with our custom node editor.

Jump mechanic over the 1x1 challenge

The *jump* mechanic is used to jump over a short obstacle, such as the *1x1* challenge. These are considered the easiest mechanic and challenge in our game.

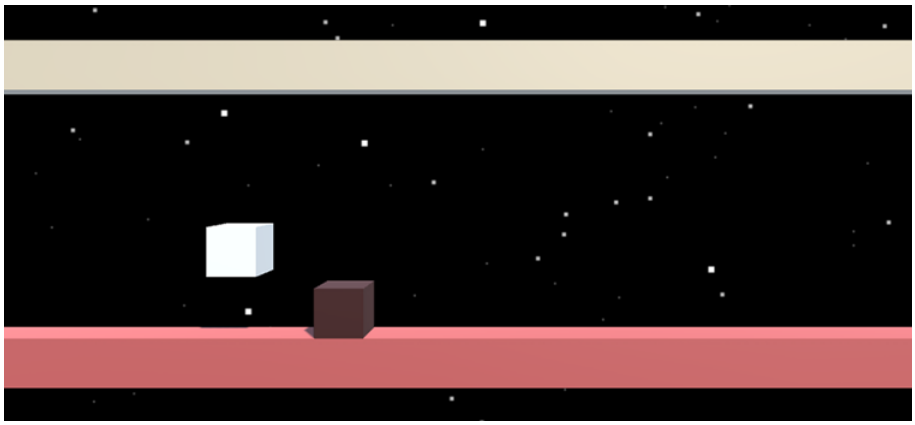


Figure 4.20: Player jumping over the 1x1 challenge

Double jump mechanic over the 2x2 challenge

The *double jump* mechanic allows to jump over taller (and shorter) obstacles, such as the *2x2* challenge. This mechanic can be triggered mid-air right after the player used the *jump* mechanic. This means that if the jump mechanic is not active then the player will never be able to double jump, no matter if it is active or not.

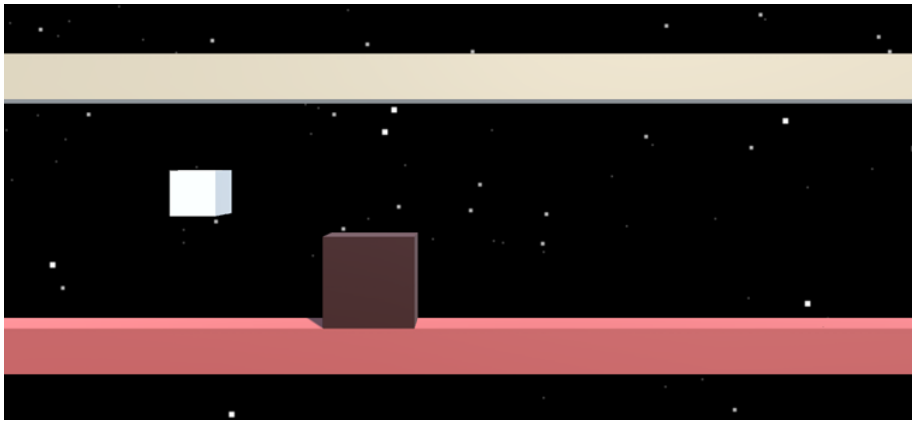


Figure 4.21: Player double-jumping over the 2x2 challenge

Dash mechanic through the thin-wall challenge

The *dash* mechanic is used to pass through challenges, such as the *thin-wall* challenge. Dashing challenges are blue coloured to distinguish them from the other challenges. This mechanic cannot be used to dash through the *1x1* or *2x2* challenges, because of their different colour.

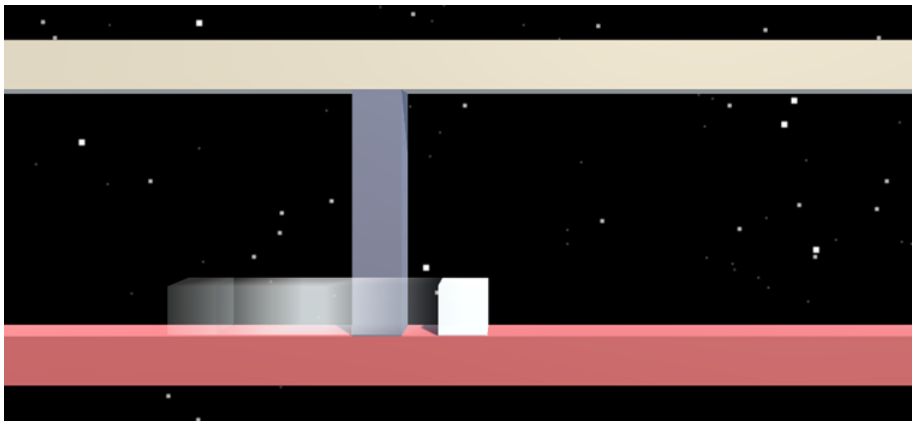


Figure 4.22: Player dashing through the thin-wall challenge

Jump or dash mechanics over the hole challenge

Both the jump and double jump mechanics, as well as the dash mechanic, can be used to clear the *hole* challenge.

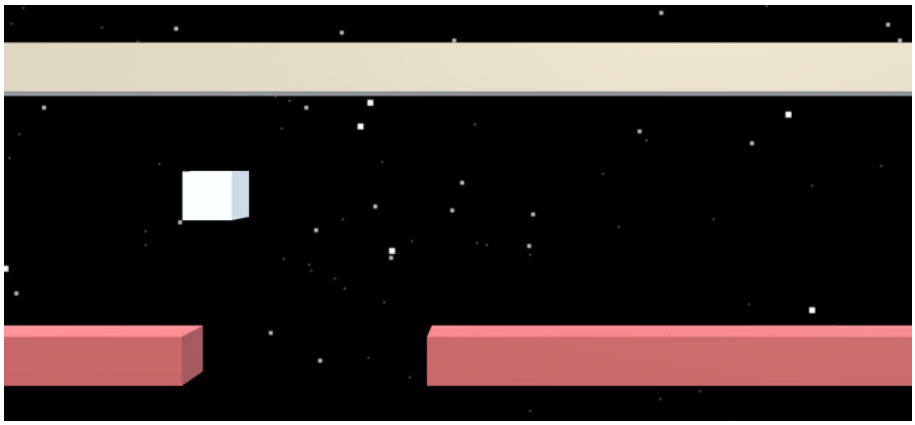


Figure 4.23: Player double-jumping over the medium-hole challenge

Slide mechanic under the slide-wall challenge

The last type of mechanic is the *slide* mechanic. It can be used to pass under obstacles, such as the *slide-wall* challenge.

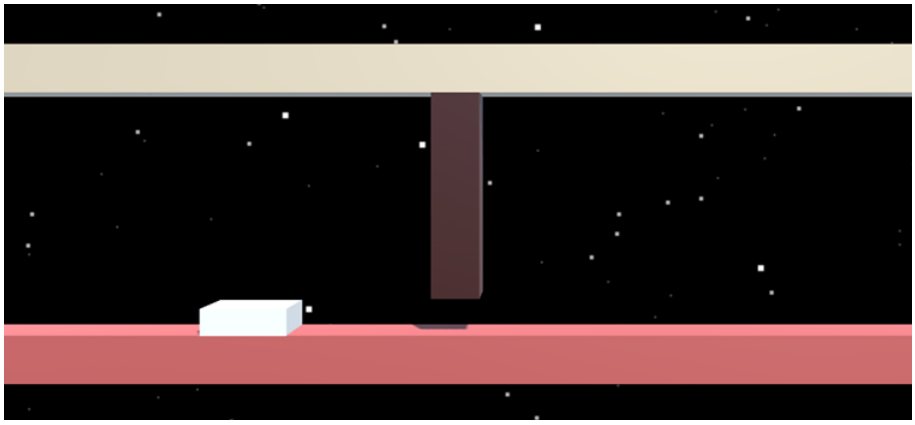


Figure 4.24: Player sliding under the slide-wall challenge

It is also possible to combine different mechanics and use them at the same time. The following figure shows an example of this: the player used the jump and the dash mechanics at the same time to be able to overcome both obstacles. This creates more diversity in the way mechanics can be used to overcome consecutive challenges.

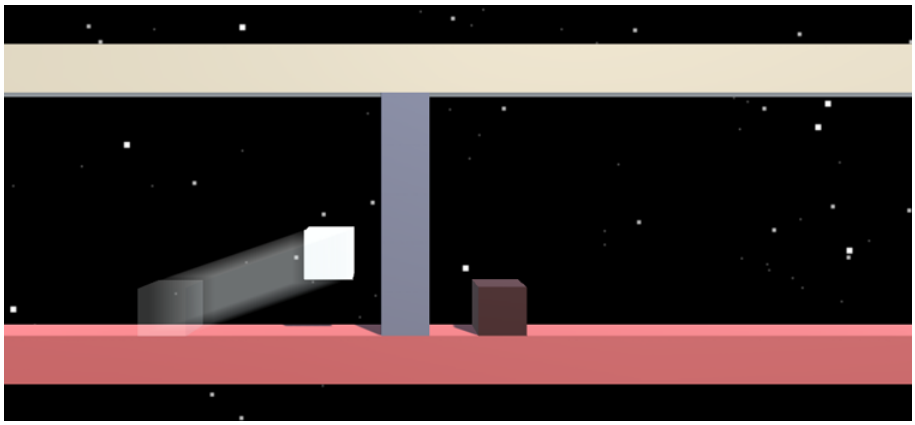


Figure 4.25: Player dash-jumping through the thin-wall and over the 1x1 challenge

4.2.2 Player Mastery

As explained at the end of the chapter 3, we use weights associated to a window of attempts in order figure out the level of mastery of the player in a specific feature of the game.

An example of values for the weights can be seen in the following figure.

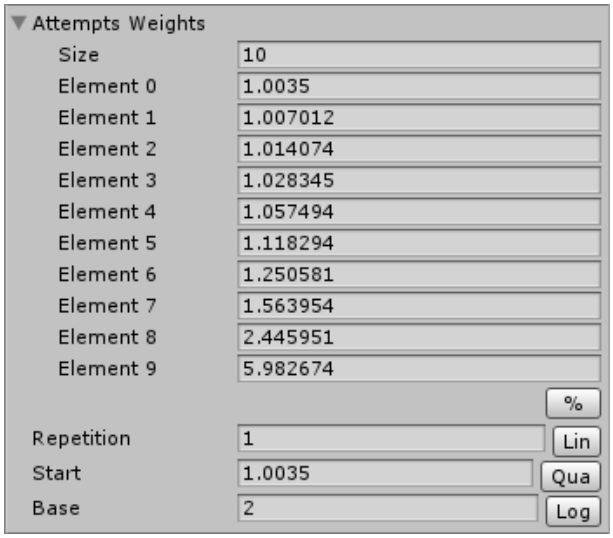


Figure 4.26: Weight values for the window of attempts

It is possible to modify the size of the window. As you can see in our figure that value is set to 10, meaning that the game will only keep track of the last 10 successes or failures per feature. Next, it is displayed the values for the weights. These can be input manually or set according to one of the four default configurations: equal probability, linear scale, quadratic scale or logarithmic scale.

Equal probability is visually represented by the button % and means that all attempts will have the same value for their weight.

Linear scale (*Lin*) produces incrementing values for the weights, controlled by the *Repetition* parameter. With Repetition set to 1, it generates the weights 1, 2, 3, etc. With Repetition set to 2 it generates 1, 1, 2, 2, 3, etc.

Quadratic scale (*Quad*) generates values following the formula

$$y[x] = y[x-1]^2 \tag{4.1}$$

meaning that the weight value of a specific index is the weight value of the previous index squared. Its only parameter *Start* defines the value for the weight of the first attempt.

Logarithmic scale produces values according to the formula

$$y[x] = \log_b x \tag{4.2}$$

meaning that the weight value of a specific index is the logarithmic value of the index to the base specified in the *Base* parameter.

As described in the previous chapter, we use the formula 3.1 to calculate the score of the player for a specific feature across a window of attempts, where we assume that each success has a value of 1, each failure has a value of 0.

To decide if a given attempt is successful or not, the player must try to overcome each one of the features (challenge, mechanic or pace) in the game. The following figure shows the limits of each feature that are taken into account when deciding if an attempt on a feature was successful or not. For each type of feature, the condition to decide if an attempt was successful is different:

- Challenge: an attempt to overcome a challenge is considered successful if the player doesn't collide with the challenge or falls through it (i.e. if the player can exit the vertical red limit , otherwise it is considered a failure);
- Mechanic: an attempt to use a mechanic is successful if the player uses that specific mechanic when successfully overcoming a challenge (i.e. if a mechanic is used between the green and red vertical limits and the player exits the vertical red limit, otherwise it is considered a failure);
- Pace: an attempt to overcome a specific pace is considered successful when the player successfully overcomes a challenge provided at that pace (i.e. if the player can successfully overcome all the consequent challenges with the same distance in between).

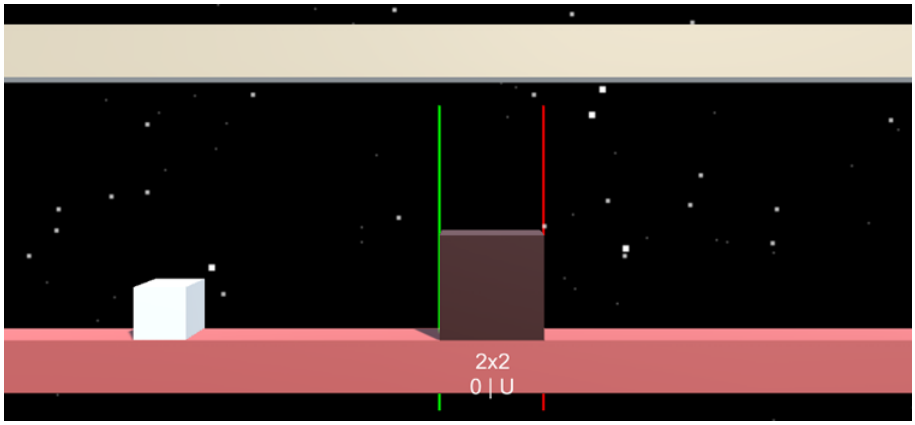


Figure 4.27: Start and end limits for an attempt on a 2x2 challenge

4.3 Progression Example

The following figure depicts an example of a working progression graph. A larger version of the figure can be seen in Appendix A.

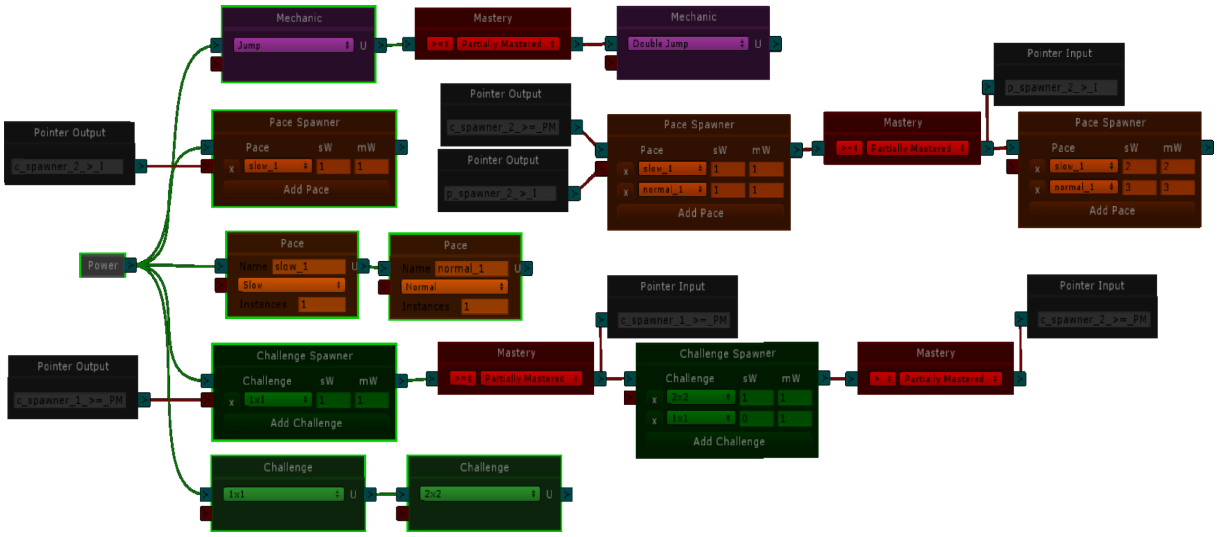


Figure 4.28: Example of a working progression graph

The enabled nodes at the start of the playthrough can be identified by the nodes with a green outline. In this example, these nodes are *Mechanic: Jump*, *Pace: slow_1*, *Pace: normal_1*, *Challenge: 1x1*, *Challenge: 2x2*, *Challenge Spawner: 1x1* and *Pace Spawner: slow_1*. At this moment, the game will constantly provide *1x1* challenges with a *slow* pace (large distance between challenges) and only allows the player to use the *Jump* mechanic.

When the mastery of the player increases to *Partially Mastered* in the *Jump* mechanic then the game will enable the *Double Jump* mechanic. Similar to this, when the mastery of the player increases to *Partially Mastered* in the *1x1* challenge then the game will enable the new *Challenge Spawner: 2x2 and 1x1* and will disable the old *Challenge Spawner: 1x1*. Both nodes should be enabled at the same time if we are using the same values for the weights of the window of attempts. From now on the game will only spawn *2x2* challenges (note *sW* = 0 for the *1x1* challenge) and will allow the player to use both *Jump* and *Double Jump* mechanics. If the mastery of the player decreases below the specified threshold for the *Jump* mechanic then the player will not be able to use the *Double Jump* mechanic until the mastery condition is verified again.

When the weighted average mastery for the *1x1* and *2x2* challenges is greater than *Partially Mastered* then the game will enable a new *Pace Spawner: slow_1 and normal_1* and will disable the previous *Pace Spawner: slow_1*. From now on the game will provide *2x2* challenges at *slow_1* and *normal_1* paces, with the same probability. If the average mastery of the player for the *1x1* and *2x2* challenges decreases below *Partially Mastered* then the game will fallback to the *Pace Spawner: slow_1* node.

When the weighted average mastery for the *slow_1* and *normal_1* paces is greater or equal than *Partially Mastered* then the game will enable a new *Pace Spawner: slow_1 and normal_1* and will disable the previous one. From now on the game will provide *2x2* challenges at both paces, privileging the *normal_1* pace due to its higher *sW* value. If the average mastery of the player for both *slow_1* and *normal_1* paces decreases below *Partially Mastered* then the game will fallback to the previous *Pace Spawner: slow_1 and normal_1* node.

Having mastery conditions and making use of the blocker input, we make sure that there are fallbacks to the progression graph which will cover most of the uses cases when the player suddenly decreases

its level of mastery in one of the features of the game.

Summarising this progression graph, the game starts by providing 1x1 challenges to the player and only allowing the use of the *Jump* mechanic enabled. When the player increases its mastery in both the mechanic and the challenge then the game will give a new *Double Jump* mechanic to the player and will start providing only *2x2* challenges in order to test the player in the newly acquired mechanic. From now on, the game will consequently increase the pace of the game until it uses more frequently the *normal_1* pace. If at any given moment the players starts to decrease its mastery then the game will slower its pace.

Chapter 5

Evaluation

In order to validate and improve our model and its implementation we evaluated the usability and quality of the implemented solutions by the testers, on various occasions during the development of this dissertation.

In this chapter we present the procedures, results, changes and insights for each moment of evaluation. We start with preliminary evaluations, informal studies conducted during the development stage. Next, we will describe in detail a evaluation of the usability of our approach. Finally, we explain our qualitative evaluation, conducted to measure the quality of the solutions implemented by the testers. To end this chapter, we will make an overview of the insights gained from the different evaluations.

5.1 Preliminary evaluations

A preliminary evaluation was conducted several times during the development of this work. Their objectives were to guide the implementation of our model in order to determine what changes and additions needed to be implemented in order to improve it.

These evaluations didn't have a formal procedure. The reached results and consequent changes emerged from informal discussions and interactions with the tool.

5.1.1 Results

Thanks to the several informal evaluations we were able to find some flaws with the tool that made more difficult or prevented us from implementing specific progression solutions.

The first flaw that we found was the impossibility of keeping a node transition active when a node or a mastery condition became inactive (i.e. once a transition is active it will be active forever, independently if its parent node becomes inactive or not). This flaw was detected when we decided to unlock a mechanic forever after a set of conditions were satisfied. The problem emerged when one of those conditions suddenly became inactive and we couldn't use the newly acquired mechanic anymore.

The second issue that we were able to identify was related to the *Challenge Spawner* and *Pace Spawner* nodes. Initially, we only had a *Weight* column, responsible for specifying the weights for both the spawning of a feature and the calculation of the weighted average of all the specified features in a spawner node. The issue arose from the fact that the users of the tool would be limited when using this type of node, because they might not want to use the same value for both calculations.

For progression solutions with a higher number of nodes, we found that it was harder to visualise how and why the different nodes were connected. The issue was that the lines representing node

connections would sometimes overlap or be hidden behind existent nodes, which made it difficult to sometimes see which nodes were connected by a specific line.

Finally, the last issue that we were able to find was that there wasn't an easy way to make sure that only one *Spawner* of each type was enabled at a time (a requirement of our tool). This would be useful if we wanted to have two copies of the same *Spawner* node with different weights and wanted to switch between them, making sure that only one of them is active.

5.1.2 Changes

Regarding the first flaw, the solution was to implement the *Memory* node. The following figure shows the functionality of this node, when a mastery condition is satisfied enabling a new mechanic and then becomes unsatisfied but still allows the usage of previously enabled mechanic.

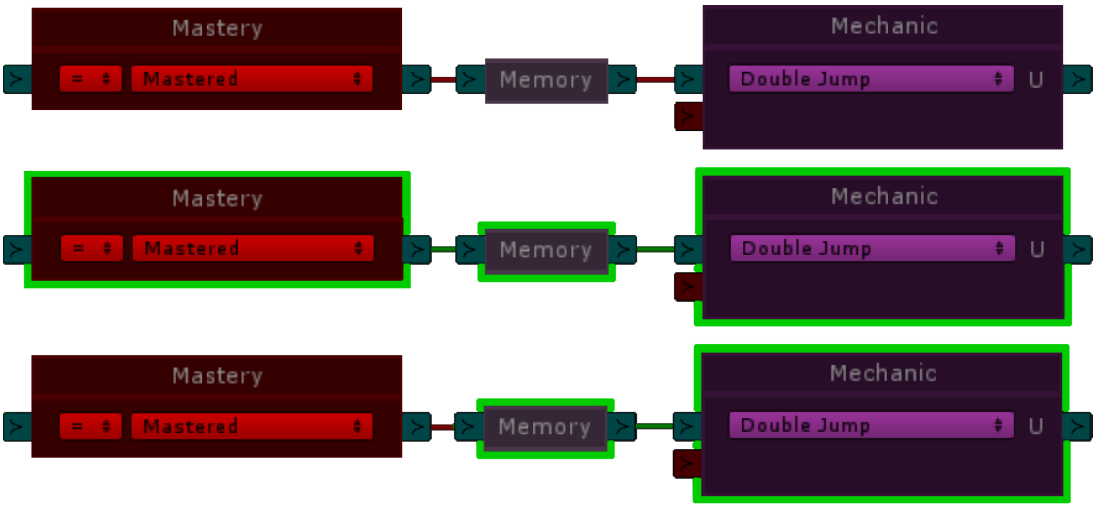


Figure 5.1: Example usage of the Memory node

For the second issue, we separated the existent *Weight* column into *Spawn Weight (sW)* and *Mastery Weight (mW)* columns. With this solution, we were now able to specify different values for the weights of spawning and calculation the weighted average mastery of a spawn node. The following figure displays the previous and the current version of a *Challenge Spawner* node. The same changes were applied to *Pace Spawner* nodes.

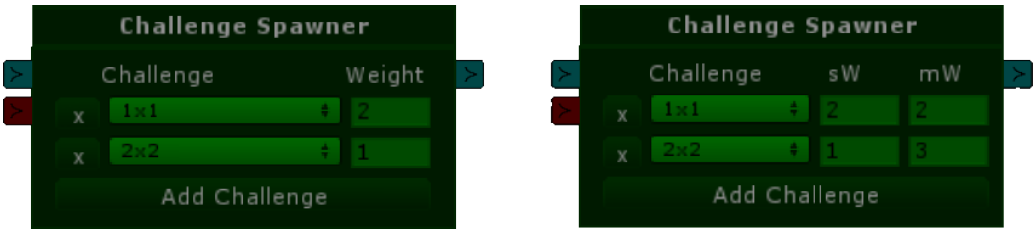


Figure 5.2: Comparison between the previous and current version of the Challenge Spawner node

We weren't able to implement a robust solution that would take care of all the issues with the third flaw. Our simpler solution was to implement *Pointer* nodes, a group of nodes that works like variables. We use a *Pointer Input* node to save the result of a node/transition into a variable with a specific name and then use its complementary *Pointer Output* with the name of the variable that we want to read from. This solution helps in reducing the clutter of hidden or overlapping transitions, but increases the number of nodes existent on the screen. A better solution, as requested by the testers in the other evaluations

and specified in the chapter 7. Future Work, would be to have a smart alignment algorithm that would draw the lines without overlap with other lines or nodes. The following figure shows an example of the same partial progression solution without and with *Pointer* nodes.

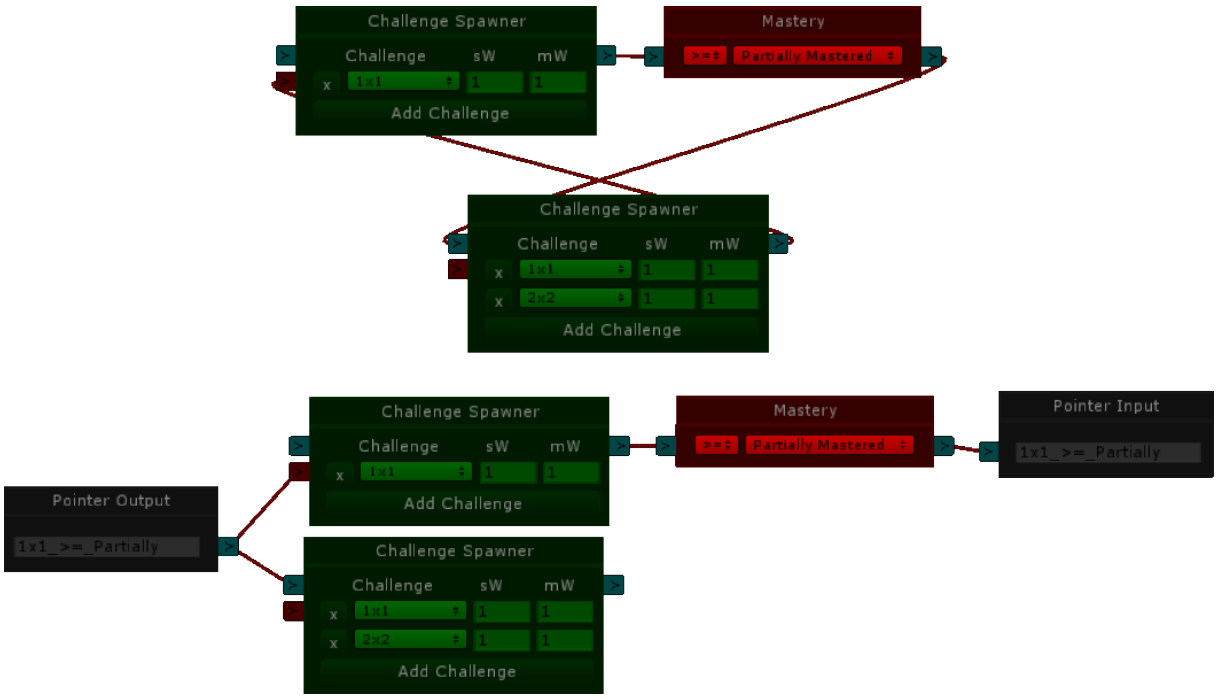


Figure 5.3: Example usage of Pointer nodes

For the last identified issue the solution was to create a *blocker input* for all *Mechanic*, *Challenge* and *Pace* nodes. When we connect the output of a node *B* to the *blocker input* of the node A, we have the guarantee that as long as the node *B* is active then the node *A* will be disabled. The following figure displays an example of usage for the *blocker input*, where it can be used to toggle between different *Challenge Spawner* nodes depending on the mastery of the player when learning the newly introduced *Medium-Hole* challenge.

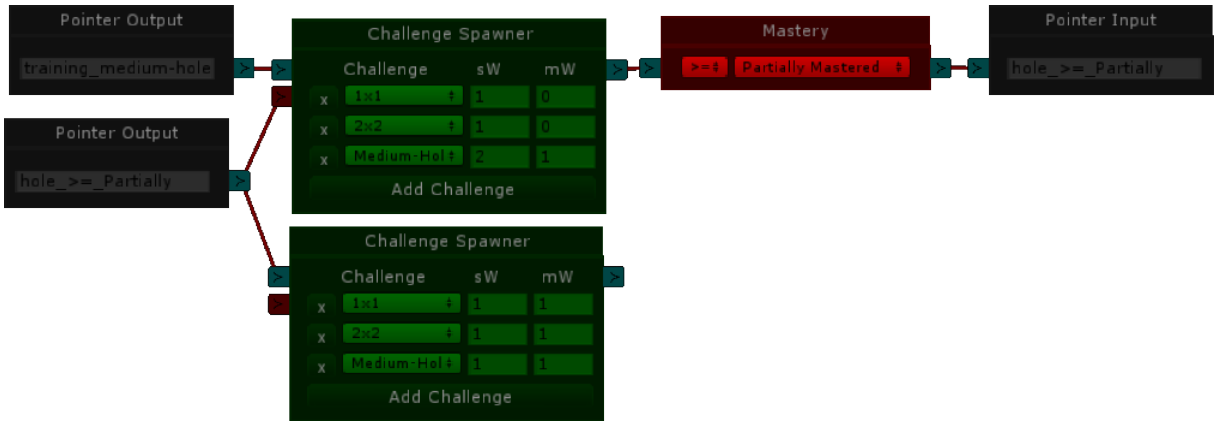


Figure 5.4: Example usage of the blocker input

5.2 Usability evaluation

After implementing all the required changes detected in the preliminary evaluations, we decided to conduct a study with some users with the objective of finding out some usability issues within our tool when executing simple tasks.

5.2.1 Procedure

For this study, we looked to find students with different university backgrounds and different levels of experience using game development tools. We had four participants which we could categorise in two distinct groups:

- two students of Game Design with knowledge in the creation process of a game and use of game development tools;
- two students with no background in Computer Science and with no experience in Game Design or tools for game development;

The objective of this evaluation was to visualise the interaction of each user with our tool and to determine usability issues that prevented or made more difficult the execution of each task. The reasoning behind the choice of these groups of participants was that we wanted non-professionals with zero or limited experience in game development, because it would be easier for these participants to have problems interacting with the tool due to their lack of professional experience.

At the beginning of each session we made a short demonstration of our game, showing the different challenges of the game and the respective mechanics used to overcome each one of them, so that the participants could relate the names of the different mechanics and challenges to the ones presented in the editor tool. Next, we showed our editor tool, describing the functionality of each node and how they affected the game. After this initial contact with the tool we presented a template of the minimal working progression solution, which enabled the use of the *Jump* mechanic to overcome *1x1* challenges at a *Slow* pace. The following figure displays the template presented to the participants.

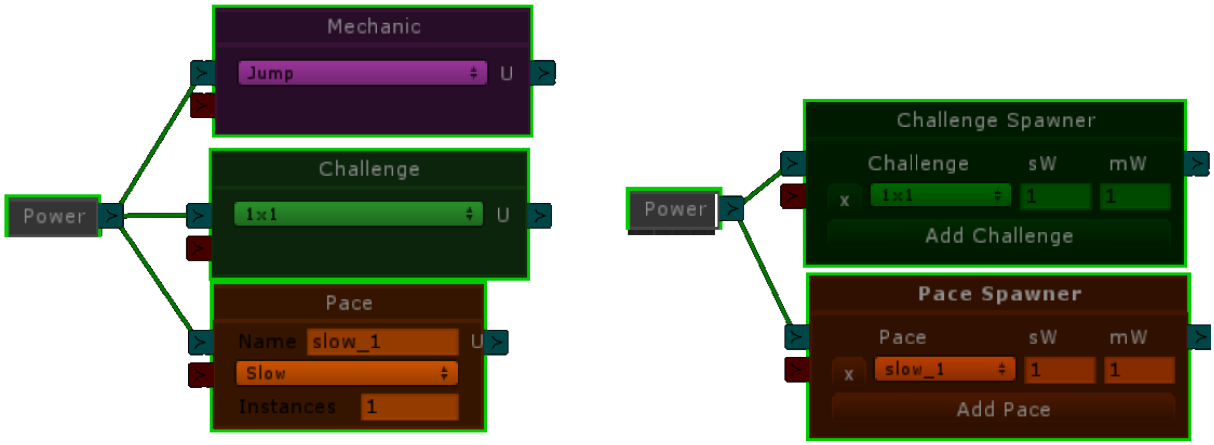


Figure 5.5: Minimal working progression solution / Starting point of Task #0

Afterwards, we presented the list of tasks that each participant would have to execute and explained that each task would have as starting point the solution for the previous task. The starting point for the first task was the template presented above. The (already revised) list of tasks (and respective expected solutions) can be consulted in Appendix B. For this evaluation only the tasks #1 to #6 were presented to each participant.

We then started the practical experiment, where we let the participants execute the six given tasks and to make questions/comments whenever they felt appropriate. We estimated each session to last about 60 minutes, in order to give plenty of time to allow each participant to execute all the tasks.

There was no intervention by the observer, except when requested by the participants. After participants were finished with these tasks, an unstructured interview was conducted in order to gather some additional feedback by the participants.

5.2.2 Results

We noticed that the group of students with no background in Computer Sciences executed some tasks differently than our expected solutions, although they yielded the same result. These participants never measured the combined levels of mastery from the *Spawner* nodes, opting to measure the mastery of each individual *Challenge* or *Pace* node, and didn't use *Pointer* nodes to better organise their solutions.

After all the sessions with the participants, we noticed that the group of students with background in Game Design were able to execute each task faster than the other group and executed each task accordingly to our expected solutions.

All participants commented during the execution of the tasks that the organisation of the node selection menu, specifically the *Spawner* nodes, was confusing and increased the time to pick the correct node. The participants also made remarks regarding the way the transition connecting two nodes were drawn. When two nodes were close to each other, the drawn transitions were excessively curved and added unnecessary "artefacts" to each solution. There were also comments regarding the vocabulary used in the description of each task, which sometimes induced doubts to the participants which felt the need to clarify the description with us.

Some participants also noted the lack of a complementary node to measure the weighted average of different mechanics.

5.2.3 Changes

Regarding the fact that the group of students with no background in Computer Sciences executed some tasks differently than our expected solutions, we attributed this phenomenon to a possible incomplete or confusing introduction to the tool given by us. We decided that for the next evaluation we would improve the presentation of the tool and include some simple examples of usage for the *Pointer* nodes, *blocker* inputs and calculation of a weighted average mastery from *Spawner* nodes.

The organisation of the selection menu was revised accordingly to the feedback given by the participants. The following tables show a comparison between the two versions of the menu. The previous menu is presented on the left and the current menu (the same as the Table **??**) on the right.

Boolean	Power	
	And	
	Or	
	Not	
	Memory	
Game	Mastery	
	Timer	
	Mechanic	
	Challenge	
	Pace	
	Spawner	Mechanic
		Challenge
		Pace
	Pointer	Input
		Output

Boolean	Power	
	And	
	Or	
	Not	
	Memory	
Game	Mastery	
	Timer	
	Mechanic	Single
		Average Mastery
	Challenge	Single
		Spawner
	Pace	Single
		Spawner
	Pointer	Input
		Output

Table 5.1: Comparison between the previous and the current node selection menu

The solution for the excessively curved transitions for nodes close to each other was to draw direct transitions between these nodes. The following figure displays the previous and the current version of drawing short transitions between nodes.

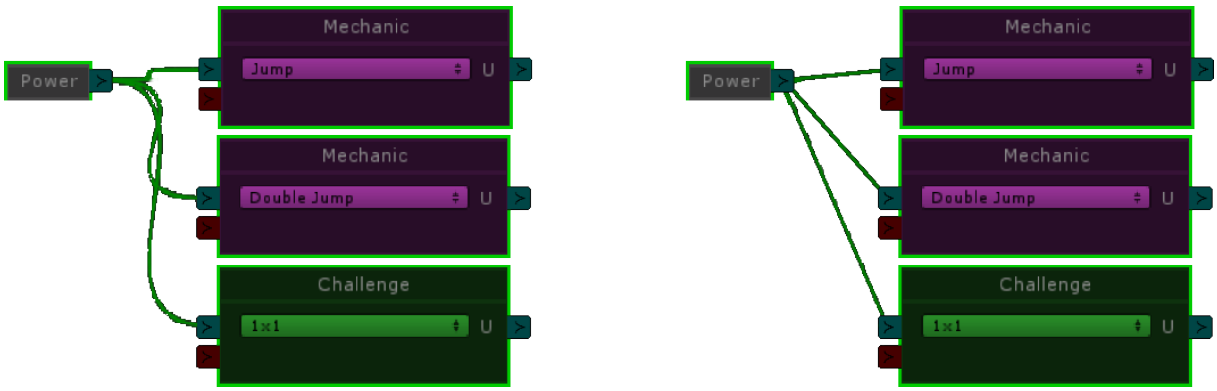


Figure 5.6: Comparison between the previous and current versions of node transitions

Regarding the vocabulary used in the description of each task, we rewrote the description of the tasks where the users had more difficulties, especially the tasks which included measuring levels of mastery from *Spawner* nodes or from the individual ones. We also added two more tasks that used the remaining types of nodes not explored by the users, the *Timer* node and the new *Mechanic Average Node* suggested by the participants.

5.3 Qualitative evaluation

The third and last moment of evaluation was mostly a qualitative evaluation, in order to measure the quality, efficiency and usefulness of the model, tool and implemented solutions for each task.

5.3.1 Procedure

In this study, we looked to find a mix of students and professionals with background in Game Design and experience using game development tools. We had a pool of six participants which we categorised in two distinct groups:

- three students of Game Design with knowledge in the creation process of a game and use of game development tools (different);
- three game developers with professional experience in Level and Game Design;

Regarding the group of three professional game developers, two participants were Game Designers at Miniclip[22] and the third participant was Lead Engineer at Bica Studios[23].

The objective of this evaluation was to visualise the interaction of each user and the solution for each task, especially the "creative" task #8, and to determine creative and efficiency issues that prevented or made more difficult the implementation of a specific progression model. The reasoning behind the choice of these groups of participants was that we wanted to compare the creative process between professionals and non-professionals, in order to earn common (and particular) feedback in each group that would help us determine the changes (if any) needed to improve the quality, efficiency and usefulness of the tool and solutions.

The initial demonstrations of the game and tool were similar to the procedure carried in the previous evaluation. We presented a short demonstration of our game, showing the different challenges of the game and the respective mechanics used to overcome each one of them, showed our editor tool describing the functionality of each node and how they affected the game and finally presented a template of the minimal working progression solution, which enabled the use of the *Jump* mechanic to overcome *1x1* challenges at a *Slow* pace.

Differently from the previous evaluation, we implemented in real-time and explained some examples of usages related to *Pointer* nodes, *blocker* inputs and calculation of a weighted average mastery from *Spawner* nodes, in order to prevent the interpretation and usage issues found during the previous evaluation.

Afterwards, we presented the list of tasks that each participant would have to execute and explained that each task would have as starting point the solution for the previous task. The starting point for the first task was the template presented above. The (already revised) list of tasks (and respective expected solutions) can be consulted in Appendix B. For this evaluation all the eight tasks were presented to each participant.

We then started the practical experiment, where we let the participants execute the given tasks and to make questions/comments whenever they felt appropriate. We increased the size of each session to 90 minutes, in order to accommodate the addition of the two new tasks since the previous evaluation.

There was no intervention by the observer during the execution of a task, except when requested by the participants. During the execution of each task we noted down any difficulties, issues and differences from our expected solutions, as well as the duration of each task. After the execution of each task the participants tested and explored their implemented models of progression. We also conducted a structured interview after the participants were finished with all the tasks. The feedback template for this evaluation can be seen in Appendix C.

5.3.2 Results

The average times for completing each task were as follows:

Task	#1	#2	#3	#4	#5	#6	#7	#8
Duration (minutes)	2	3	4	6	7	8	10	15

Table 5.2: Average times of completion for each task

The duration of each task was measured since the user started reading the description of the task until the end of the playthrough of the implemented solution in the game. As each task adds more nodes and mastery conditions it inevitably increases the duration of playthrough respective to that task, because more (and increasingly harder) mastery conditions must be met.

After the tests with all the participants it was noticeable that, although all solutions were equivalent because they affected the game in the same way, there were a few differences between how professionals and non-professionals solved each task. The solutions of the non-professionals were similar to our expected ones, in the sense that they enabled and disabled (blocked) all expected nodes as specified in each task. The approach by the professionals was to activate all *Challenge* and *Pace* nodes at the start and to enable/block their respective spawners. The solutions also differed in which node output was used to block another node. Non-professionals tend to block a node right after activating another one, as if it was a sequential operation, while professionals tend to block and enable nodes at the same time, as if it was a parallel operation. The following two figures show example implementations of Task #4, the first from a non-professional and the other from a professional. In these figures we can see the differences between the solutions, as described.

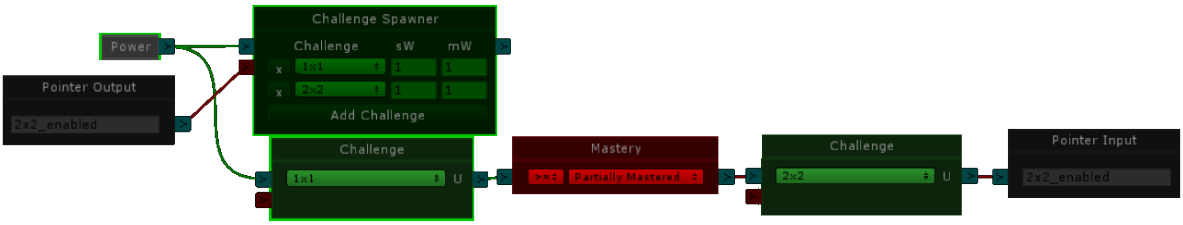


Figure 5.7: Non-professionals' solution to Task #4

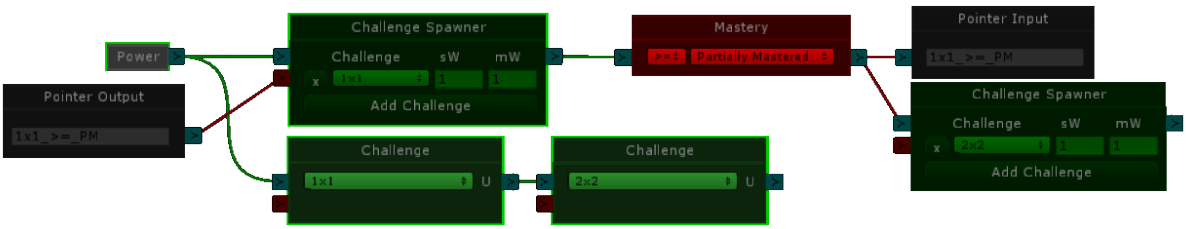


Figure 5.8: Professionals' solution to Task #4

For Task #8, we expected the users to extend their solutions in rather different ways and to test the capabilities/limitations of the tool. Interestingly, all the implemented solutions were based in enabling the remaining challenges and in increasing the pace of the game as the players increases their levels of mastery across all the features of the game. Most of the implementations in this task were duplicated groups of nodes, but with increased thresholds for the mastery nodes.

Despite the simplistic solutions, we were still able to improve our understanding on how a designer would use the tool without constraints and what improvements should be made in order to increase the efficiency of the interactions. We noticed that the participants (most of times) preferred to use the *Duplicate* node function instead of inserting a new node from the contextual menu. We also noticed that for task #8 the users copied several times the same bits of the existing solution, instead of trying to implement something different. This made us believe that there should be a way to clone and move groups of nodes, which would drastically increase the productivity for the implementation of any type of creative tasks.

In the structured interview, after the execution of all the tasks, the participants gave us precious comments regarding the quality, efficiency and usefulness of the tool.

Starting with the "negative" feedback, the participants noted three serious problems that they considered to greatly frustrate them and to negatively affect their productivity, but that do not make the implementation of any of the requested tasks impossible. The participants also provided possible solutions to these issues.

The first identified issue was related to the *Duplicate* node command, which creates an empty copy of the selected node. All the users expected and wanted the *Duplicate* node command to create a clone of the selected node, copying the same values for the parameters of the original node. This usability (but also efficiency) issue wasn't detected earlier due to the fact that in the previous evaluations the participants weren't asked to execute tasks #7 and #8.

The second identified issue was the lack of selection of multiple nodes at once. The explanation given by the users for these issues was that almost any node logic can be reused in other places, so there should be an easy way to select a group of nodes, duplicate them and move them to the desired location in the canvas. These comments come to verify our observations when the participants were executing the different tasks, especially task #8.

The last serious issue identified by the participants was related to the transitions drawn between nodes. As the number of nodes increases within a canvas the harder it becomes to visualise it, due to the all the clutter created by overlapping lines and lines partially hidden behind existing nodes. We had already identified this issue during the preliminary evaluations, but weren't able to implement a robust solution. Although the users acknowledged the fact that the *Pointer* nodes could be used to mitigate this problem, they suggested the implementation of smart alignment for the lines, in order to eliminate overlapping lines and lines hidden behind existing nodes, in conjunction with a manual alignment feature for fine tuning whenever needed. This suggested solution had been also identified by us during the preliminary evaluations.

Despite the aforementioned issues, all the participants provided positive comments regarding the tool. The most appreciated feature was the excellent mental picture of what happens in the game and the ability to visually debug in real-time each solution, while the game is being played. According to the users, this allows for easily check if the implemented model is valid and correct, and if any tweaks should be made to improve the experience for the player. All participants also referred that the tool is easy to master and has a very intuitive interface. It was easy for the participants to understand the differences and peculiarities between nodes and how they should be used. Although sometimes there were some questions regarding the functionality of a specific node, all participants quickly understood its function once they added it and tested it to see how it affected the game.

The professional participants gave additional feedback, due to their additional experience in game development and using visual tools of this type. They considered that this tool is very adequate for level designers, because they are used to use editor tools and visual programming. They also mentioned the modularity of the editor, meaning that the same final result could be achieved by different combinations of nodes and connections used, which is a good representation of the expressiveness of the tool. Finally,

they appreciated the fact that our tool allowed the possibility of measuring burn-out and frustration levels of the player, which according to them is something difficult to measure in all types of games, and the fact that these “special” levels of mastery could also be used to implement a progression model for the player.

5.3.3 Remarks

This evaluation session was very important to gather real feedback from “people in the field”, i.e. participants that are used to create games and levels of all sorts of difficulties, especially the professionals. Their feedback was decisive in order to validate the quality, efficiency and usefulness of the model and tool.

On the usability and productivity levels, the participants found some flaws that aren’t usually found in this type of tool. The ability to select multiple nodes and to deeply clone them is found in all the visual programming tools and was expected to exist in our tool.

The model and respective implementation were appraised by all the participants, which were quick to mention other games that could really benefit from them. Most of the mentioned games were developed by the participants or by the companies they work for and included other types of games that weren’t endless-runners, such as arcade or puzzle games.

Also very appraised is the expressiveness of the tool, because it is possible to implement the same end solution using different nodes and transitions. This was considered extremely valuable by the professionals, because it shows that there isn’t a correct way of making things, facilitating the usage of this tool by every type of people, even with the most different backgrounds.

The favourite feature of the model and tool was how easily it was to visualise, debug and understand the implemented progression models and directly map it to what is happening the game. According to the participants, the fact that we decided to use such a visual and easy to use approach to tackle the problem should speak for itself to explain the usefulness and quality of the tool.

All the participants also mentioned several improvements and features that they would like to see in the tool. Although being very valuable feedback, we consider that these improvements and features do not compromise the usability, quality or usefulness of the tool, so we decided to present them in the section 7. Future Work.

5.4 Summary

Through these evaluation sessions, we were able to draw some important conclusions regarding the usability, quality, efficiency and usefulness of the model and tool.

As the tool evolved during its development cycle, it became more robust regarding the provided functionalities and how these affected the game.

The several preliminary evaluations were important to tweak and improve the usability of the tool. They allowed us to develop the tool in order to reach a desired level of usability so that the tool could be tested by other people in specific tasks, for the usability evaluation.

During the usability evaluation, we gained crucial feedback regarding the usability of the tool. This evaluation session allowed us to identify usability problems that we weren’t able to identify before, especially regarding the node selection menu and the way that node transitions are drawn. These problems were then fixed for the next evaluation.

For the qualitative evaluation, we were convinced that we wouldn’t find any more usability issues, due to our extended knowledge of the tool and due to the issues (eventually solved) already detected during

the usability evaluation. Unexpectedly, due to the introduction of the new tasks #7 and #8 in this evaluation, some usability and productivity issues were found regarding the selection and cloning of multiple nodes. We blame this in the fact that we should have validated the changes from the previous usability evaluation with a second usability evaluation, that would have found earlier the problems detected in the tasks #7 and #8.

We consider that the qualitative evaluation was the most important source of feedback for this dissertation, because it allowed us to test the tool directly with professional developers and other students used to develop their games, i.e. “people in the field”. The received feedback regarding the quality, efficiency and usefulness of the tool makes us believe that this dissertation is a step in the right direction in creating a robust model able to represent the possibilities of progression of a player and having a game adapt its content to the skill of the player.

Chapter 6

Conclusion

We started this project with the simple idea of bringing player skill to the dimension of *Procedurally Content Generation* (PCG), in a way that we could create a game without difficulty settings and that it adapted itself to what the player can do at each moment of a playthrough.

All the performed evaluations made us believe in the potential of our approach and our node editor tool. All the users were able to quickly understand all our concepts, to implement all the tasks they were asked to and were able to quickly find progression issues (such as when a graph path would never become active) and to suggest fixes and improvements to a solution in order to turn the playthrough of the game more interesting and enjoyable.

Also fortunate was the fact that all the users provided great feedback for features and improvements to be implemented in our tool that they felt would increase the productivity and expressiveness of each interaction, resulting in better solutions for guiding the progression of the player in each playthrough. These suggestions can be seen in the next chapter Future Work.

We feel that the next logical step for this project is the implementation of the features described in the next chapter and to perform a new evaluation, in order to validate the current and the new features using a quantitative approach.

We believe that this work was a good first step in the right direction to solve the problem of using player mastery in *PCG* and that we created a tool that can be easily adapted and used by level designers for generating content for other endless-running games.

However, we weren't able to validate if the playthroughs generated by our model had levels of replayability, fun and engagement similar to commercially distributed games of the same genre. We hope that additional qualitative and quantitative evaluations will answer to this question. This stands as future work for this project.

Chapter 7

Future Work

Despite the positive feedback in the latest evaluations we were still able to find some improvements to the editor, which we are sure will enhance the usability and productivity in future interactions of the users with the tool. This being said, we then propose below the improvements to the tool that should be implemented as future work.

Starting with the serious issues identified during the last evaluation, which we describe in detail at the end of section 5.3, we propose: the implementation of an improved *Duplicate* node command, in order to create a clone of an existing node with the same values for its parameters as the original node; the addition of a multiple selection tool, which would allow the user to delete, move or duplicate more than one node at once; and an improved line drawing algorithm, which draws the line connections between nodes in a way that creates no overlapping lines or lines hidden by existing nodes, which should be complemented with a manual alignment feature for fine tuning as desired by the user.

To complement these issues, there were some suggestions mentioned by all the users that would improve their productivity when using the tool and that would extend the existing functionality of the tool to support more types of in-game features or events that could be used to create more interesting node logic for enabling/disabling features of our game. We then propose the following improvements: implementation of keyboard shortcuts, such as delete, duplicate or move nodes/transitions; addition of an *Average Mastery* node for paces and challenges, that would work similar to the existing *Mechanic Average Mastery*; and the addition of a *Challenge Sequencer* (or *Composer*) that allows the user to create more complex challenges using already existing ones.

Professional users suggested the implementation of more complex features that would potentially increase the expressiveness of the tool. Their suggestions were: the implementation of an *Event Listener* node that would be associated with a *Counter* node, which could be used to measure the number of a times a specific uncommon event occurred in the game and to modify the game accordingly, such as the number of times that the player hit the ceiling or jumped over the *2x2* challenge without touching it; to add support for *context-based masteries*, because there is a difference between measuring the individual mastery of each challenge that a player successful overcomes and to measure the mastery of two consecutive challenges where the first one was used as a ladder step to jump over the second one, where the player displays a higher level of mastery if is able to land on top of the first challenge; and to abstract an existing canvas as a complex logic unit, that can be used in other canvas as a normal *game node*.

The following improvements were not suggested by the testers, but by the authors of this work. We would like this work to incorporate changes that would extend the functionality of the editor in order to allow its use in other types of games. Our suggestions are: to extend the existent code base in order to facilitate the measurement of any kind of game metrics and respective nodes to use with our editor, such

as time played, times lost, etc.; export of code stubs from existing canvas to allow the use of canvas created by our tool with other game tools and programming languages; additional tests with users using a quantitative approach, in order to validate and improve the current and to be implemented features of our editor and game and to validate the replayability, fun and engagement, which remained unanswered in this work; generalisation of the work presented in this document, with the objective of testing the feasibility of using our tool with other genres of games that could dynamically adapt its content based on the performance of the player, such as puzzle games (like Angry Birds[24]) or *run-and-gun* games (like Metal Slug[25]).

We would also like to see this work adapted and being used to modelling the progression of others aspects of a game. For example, it would be interesting to adapt the playing speed of the background music depending on the current active pace or modify the ambiance of the game depending on the currently enable challenges or mechanics. We would also like to see a better effort in modelling a *flow curve*, based on the current skill of the player, for the generation of the current level. Although it is possible with this tool to generate playthroughs constantly adapted to the player, it isn't yet possible to reliably model a playthrough that has "peaks" of difficulty in order to introduce new mechanics or challenges and we feel that this contribution would potentially solve the unanswered questions regarding the replayability, fun and engagement of games using our model.

Bibliography

[1] M. Toy and G. Wichman, "Rogue (videogame)," 1980.

[2] Blizzard Entertainment, "Diablo (videogame series)." http://battle.net/d3/. [Online as of 06/May/2016].

[3] Sid Meier, "Civilization (videogame series)." http://civilization.com/. [Online as of 06/May/2016].

[4] Gearbox Software, "Borderlands (videogame series)." http://borderlandsthegame.com/. [Online as of 06/May/2016].

[5] Valve Corporation, "Counter-Strike (videogame series)." http://valvesoftware.com/games/csgo.html. [Online as of 06/May/2016].

[6] Valve Corporation, "Dota 2 (videogame)." http://dota2.com/, 2013. [Online as of 06/May/2016].

[7] Bethesda Game Studios, "The Elder Scrolls (videogame series)." http://elderscrolls.com/, 2013. [Online as of 06/May/2016].

[8] Rockstar Games, "Grand Theft Auto (videogame series)." http://rockstargames.com/grandtheftauto/, 2013. [Online as of 06/May/2016].

[9] Mojang, "Minecraft (videogame)." http://minecraft.net/, 2000. [Online as of 06/May/2016].

[10] M. Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*. Harper & Row, 1990.

[11] D. Cook, "The chemistry of game design." http://gamasutra.com/view/feature/1524/, July 2007. [Online as of 02/Jun/2015].

[12] R. Hunicke and V. Chapman, "AI for dynamic difficulty adjustment in games," in *Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*, 2004.

[13] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling player experience in Super Mario Bros.," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pp. 132–139, IEEE, 2009.

[14] M. Persson, "Infinite Mario Bros. (Super Mario Bros. open-source clone)," 2008.

[15] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is procedural content generation?: Mario on the borderline," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, p. 3, ACM, 2011.

[16] K. Compton and M. Mateas, "Procedural level design for platform games.," in *AIIDE*, pp. 109–111, 2006.

[17] S. Dahlskog and J. Togelius, "A multi-level level generator," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pp. 1–8, IEEE, 2014.

[18] Nintendo, “Super mario bros. (videogame),” 1985.

[19] “Technology for games and simulation.” http://fenix.tecnico.ulisboa.pt/disciplinas/TJS511/2014-2015/2-semestre/. [Online as of 02/Jun/2015].

[20] Miniclip, “Gravity Guy (videogame).” http://miniclip.com/games/gravity-guy, 2010. [Online as of 06/May/2016].

[21] Levin ”Seneral” G., “Node editor framework for unity.” http://github.com/BasteRainGames/Node_Editor/. [Online as of 14/Apr/2016].

[22] Miniclip. http://corporate.miniclip.com/. [Online as of 06/May/2016].

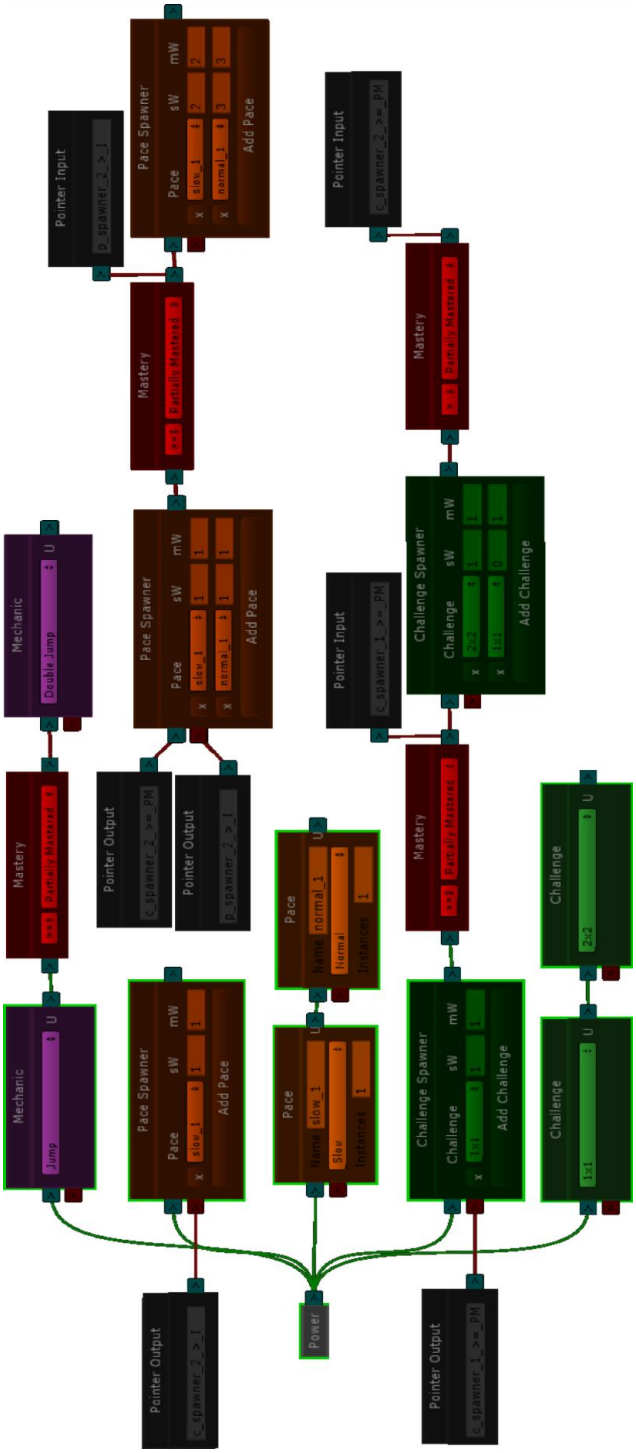
[23] Bica Studios. http://bicastudios.com/. [Online as of 06/May/2016].

[24] Rovio Entertainment, “Angry Birds (videogame series).” http://angrybirds.com/. [Online as of 06/May/2016].

[25] SNK Playmore, “Metal Slug (videogame series).” http://metalslug10th.com/. [Online as of 06/May/2016].

Appendix A

Example of a Progression Graph



Appendix B

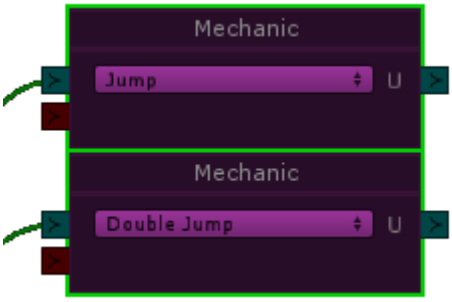
User Test Guide

Task #1

Enable the *Double Jump* mechanic.

Expected solution:

1. Add a new *Game* > *Mechanic* > *Single* node
2. Select the mechanic *Double Jump*
3. Connect the *Power* node to this node



Task #2

Enable the *Double Jump* mechanic, when the mastery of the *Jump* mechanic becomes <u>greater or equal</u> than *Partially Mastered*.

Expected solution:

1. Add a new *Game* > *Mastery* node
2. Connect the *Mechanic: Jump* node to this node
3. Select the >= operator and the *Partially Mastered* level of mastery
4. Connect this node to the *Mechanic: Double Jump* node

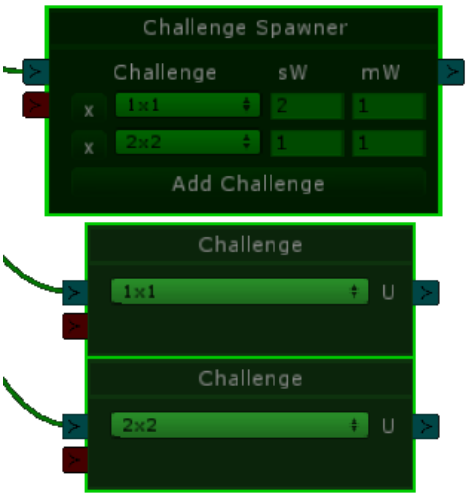
Alternatively, it is possible to use a *Boolean* > *Memory* node to prevent that the *Double Jump* mechanic becomes disabled once enabled if the specified level of mastery falls bellow the threshold.

Task #3

Enable, in parallel and with different probabilities, the spawn of *1x1* and *2x2* challenges.

Expected solution:

1. Add a new *Game* > *Challenge* > *Single* node
2. Select the *2x2 challenge*
3. Connect the *Power* node to this node
4. Locate the existent *Challenge Spawner* node
5. Press the *Add Challenge* button
6. Select the *2x2 challenge*
7. Modify the weight values



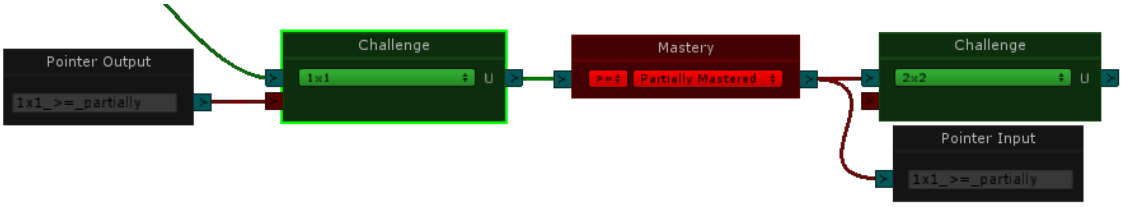
Task #4

Enable the spawn of **only** *2x2* challenges, when the mastery of the *1x1* challenge becomes <u>greater or equal</u> than *Partially Mastered*.

Expected solution:

1. Add a new *Game* > *Mastery* node
2. Connect the *Challenge: 1x1* node to this node
3. Select the >= operator and the *Partially Mastered* level of mastery
4. Locate the existent *Challenge: 2x2* node
5. Connect the *Mastery* node to this node
6. Connect the created *Mastery* node to the blocker input of *Challenge: 1x1*

Alternatively, it is possible to use *Pointer Input / Output* nodes to better organize the nodes and connections between them.

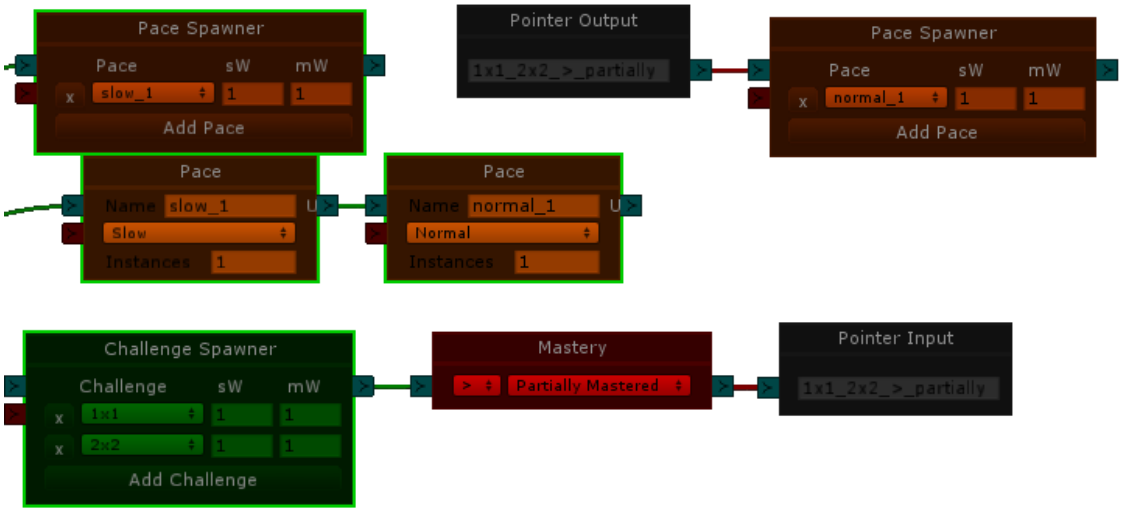


Task #5

Enable, in parallel and with the same probability, a faster *pace* of challenges, when the **average** mastery of the *1x1* and *2x2* challenges becomes <u>greater</u> than *Partially Mastered*.

Expected solution:

1. Add a new *Game* > *Mastery* node
2. Connect the existent *Challenge Spawner* node to this node
3. Select the > operator and the *Partially Mastered* level of mastery
4. Add a new *Game* > *Pace* > *Single* node
5. Write a descriptive name and select the *Normal* or *Fast* pace speed
6. (Optional) Modify the number of instances (challenges) for this node
7. Connect the *Pace: slow_1* node to this node
8. Add a new *Game* > *Pace* > *Spawner* node
9. Press the *Add Pace* button
10. Select the newly created pace
11. (Optional) Modify the weight values
12. Create and use *Game* > *Pointer* > *Input/Output* nodes to connect the new *Mastery* node to the new *Pace Spawner* node

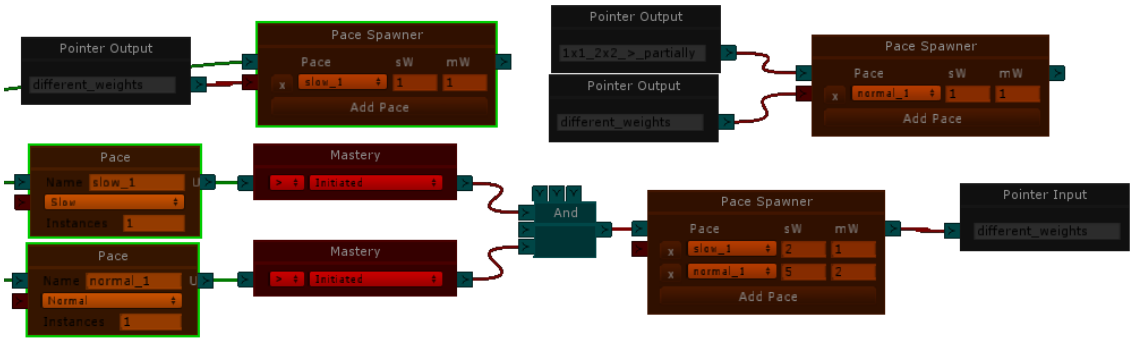


Task #6

Modify the weight values for the spawn and mastery of the existent *paces*, when the **individual** mastery of each *pace* becomes <u>greater</u> than *Initiated.*

Expected solution:

1. Add two new *Game* > *Mastery* nodes
2. Connect the existent *Pace* nodes to these nodes
3. Select the > operator and the *Initiated* level of mastery in each one
4. Add a new *Boolean* > *And* node
5. Connect the two new *Mastery* nodes to this node
6. Add a new *Game* > *Pace* > *Spawner* node
7. Press the *Add Pace* button twice
8. Select one of each created *paces* per line
9. Modify the weight values
10. Create and use *Game* > *Pointer* > *Input/Output* nodes to connect the new *Pace Spawner* to the blocker input of the other two existent *Pace Spawner* nodes

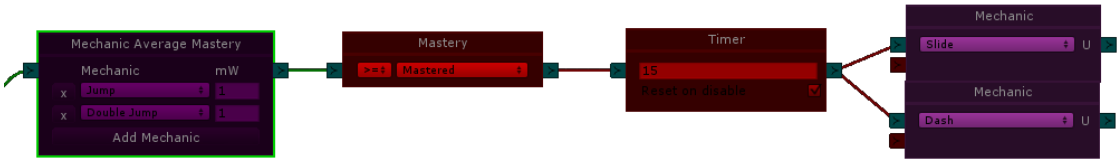


Task #7

Enable the *Slide* and *Dash* mechanics, after 15 seconds have passed since the **average** mastery of the *Jump* and *Double Jump* mechanics became <u>greater or equal</u> than *Mastered.*

Expected solution:

1. Add a new *Game* > *Mechanic* > *Average Mastery* node
2. Press the *Add Mechanic* button twice
3. Select the *Jump* and *Double Jump* mechanics in each line
4. Connect the *Power* node to this node
5. Add a new *Game* > *Mastery* node
6. Connect the new *Mechanic Average Mastery* node to this node
7. Select the >= operator and the *Mastered* level of mastery
8. Add a new *Game* > *Timer* node
9. Modify the timer value to *15* seconds
10. Connect the new *Mastery* node to this node
11. Add two new *Game* > *Mechanic* > *Single* nodes
12. Select the *Slide* and *Dash* mechanics in each node
13. Connect the new *Timer* node to these nodes



Task #8

Modify the progress made until now to your liking, in such a way that playability becomes more interesting.

Appendix C

Template of User Feedback

#0 Test Subject

Age:	Sex:	Profession:
Proficiency:	Computer – How good?	Game Development – How good?

Task #N:

What was made differently?

Were there any difficulties?

Were there any issues?

How long did it take?

What was the level of satisfaction with the tool, regarding how the final solution influences the playability of the game?

How did the initial impulse compare in contrast with the reached solution?

Which nodes/features/rules were more difficult to grasp?

What did you try to do but weren't able to?

Which features are missing and should be implemented?

What would you change in the editor in order to improve its usability?

Other comments.