

https://dspace.org/

Monster camera: creating perpetually unique horror experiences using disempowerment, reflection and uncertainty to imply a threat within video games

Spreeth, Nicolaas Remmert

2020
https://hdl.handle.net/10539/30796

Downloaded from a DSpace repository. https://dspace.org/

Monster Camera: Creating perpetually unique horror experiences using disempowerment, reflection and uncertainty to imply a threat within video games

Nicolaas Remmert Spreeth

Student number: 1932154

Supervisors: Ray Whitcher

Course: WSOA 8000 – MA Dissertation

Title

Monster Camera: Creating perpetually unique horror experiences using disempowerment, reflection and uncertainty to imply a threat within video games.

Keywords

- Horror themed video game.
- Suspense through disempowerment and reflection.
- Procedural horror event generator.
- Research tool.

Non-plagiarism Declaration

I declare that no author's work was plagiarized in the development of the procedural systems integrated into the video game "Interloper". Modified third-party assets were used to develop the game, however. These authors are credited in chapter 9.

Signed by Nicolaas Remmert Spreeth – 23 October 2020



I dedicated this essay to my Mom and brother in law,

who both passed away from cancer recently.

You inspire me to do the impossible.

I want to thank my family for supporting me throughout this journey.

Table of Contents

1. Introduction 1
    1.1 Background 2
    1.2 Objectives 12
    1.3 Clarification of Terms 19
2. Suspense 21
    2.1 Suspense versus Tension 21
    2.2 Modelling Suspense 22
    2.3 Effective Components of Suspense 25
        2.3.1 Lighting 25
        2.3.2 Colour 26
        2.3.3 Obscurity 28
        2.3.4 Sound 28
3. Disempowerment 30
    3.1 Manichean moral duality 30
    3.2 Fear and Hope 32
4. Interloper Overview 37
    4.1 Narrative Overview 37
    Monster Camera 53
        4.1.2 Monster Camera - Placement and Movement 60
    4.2 Procedural Content Generation 61
5. Development Breakdown 66
    5.1 Event Graph 66
    5.2 Main Menu 69
    5.3 Level Scripting 73
        5.3.1 Office 73
        5.3.2 Car Spawner 75
    5.4 Monster Camera Spawn System 82
    5.5 Monster Camera Events 85
        5.5.1 Monster Avatars (Character Pawns) 85

5.5.1.4 Head Bob 86
5.5.2 Monster Camera Macros 87
5.5.3 Timers 88
5.6 Player Avatar 90
5.6.1 Breathing System 94
6. Design Improvements 95
7. Conclusion 97
8. References 100
9. Additional Credits 103
10. Appendices 107
Appendix 1: UE4 Technical Reference 107
Appendix 2: Blueprint Node Reference 117
Appendix 3: Event Setup Menu 130
Appendix 4: Level Scripting – Office 140
Appendix 5: Level Scripting – Flashlight 141
Appendix 6: Level Scripting – Procedural Car Wheel 146
Appendix 7: Level Scripting – Car Wheel Rotation & Colour Selection 148
Appendix 8: Level Scripting – Procedural Car Placement 151
Appendix 9: Monster Camera Spawn System 153
Appendix 10: Monster Camera Events – Monster Cam Seek 167
Appendix 11: Monster Camera Events – Monster Cam Look 173
Appendix 12: Monster Camera Events – Monster Cam Kill 174
Appendix 13: Monster Camera Events – Head Bob 176
Appendix 14: Monster Camera Macros – Movement Initial 177
Appendix 15: Monster Camera Macros – Movement Next 188
Appendix 16: Monster Camera Macros – Kill Player 194
Appendix 17: Monster Camera Macros – Disable Light 200
Appendix 18: Monster Camera Timers 201
Appendix 19: Player Avatar 205
Appendix 20: Player Avatar – Breathing System 209

1. Introduction

As a frequent player of horror game experiences, I am always surprised by how these games tend to funnel players toward a cleverly hidden “arena” where you must confront your nemesis directly. One can argue that the atmosphere can contribute to overall suspense, but that the behaviourally significant suspense creation starts from the moment the player sees approaching foes and concludes as soon as they attack you or when you manage to damage them. You can only repeat this process a limited amount of times before you get used to the pattern and lose all sense of suspense in the process.

Many horror games are available on the market today that relies on suspense to drive the terror that the narrative represents. These games borrow from a wide variety of sources, including films, novels and each other to make it easier for players to relate to the experience. Unfortunately, this limits what can be done in the horror genre due to a lot of these sources being extremely linear in nature. This paper will explore how horror games tend to rely on a combination of direct enemy encounters and tightly choreographed spaces designed to provide players with the illusion of openness. Such areas act like funnels that drive the player down a predetermined path, even if multiple lanes exist on this path. The game usually gives the player a means, including environmental advantages and tools such as weapons, to engage and hopefully combat the grotesque monsters approaching from several directions at once.

1.1 Background

Richard Rouse III indicates, in his 2009 article *Match Made in Hell: The inevitable success of the horror genre in video games*, that an illusion of openness is achieved in horror games by combining enemy encounters with choreographed events that funnel the player in a predetermined direction (Rouse, 24). Games such as *Alan Wake*, Microsoft Studios (2010), the *Dead Space* series, (Electronic Arts - 2008 - 2013), suffers to a certain extent from this affliction. These games try to break up the monotony of these encounters by placing similar enemies in different environments, changing their attack patterns or changing each monster's appearance.



Figure 1. A mutant bursting out of a vent in Dead Space.

Dead Space (Electronic Arts - 2008 - 2013), is a game where the player portrays a space engineer that must investigate a derelict space mining vessel. Once on board, the player comes across a crew that has mutated into vicious tentacled monsters. To kill these foes, the

player must use various mining tools to cut off their appendages. The player encounters these enemies in predetermined combat areas and claustrophobic corridors. The game finds unique ways to randomise encounters with these monsters when visiting the same locations multiple times, due to the game requiring the player to backtrack to complete specific objectives. Enemies usually jump out of vents in the wall or grates in the floor, so the developers devised a smart technique to randomly choose a different vent, each time the player re-enters the same room (figure 1).



Figure 2. Synthetic enemies in Alien: Isolation.



Figure 3. Synthetic enemies unperturbed by fire in Alien: Isolation.



Figure 4. Using a flame-thrower on an Alien in Alien: Isolation.

Alien: Isolation, Sega (2014), which is a first-person survival horror game, took this a step further by tracking what a player does and augmenting the attack patterns of the alien to make each encounter uniquely terrifying. The game makes use of a novel technique that globally tracks the movements of the player and determines how long the interval between each alien encounter should be, boosting the suspense and dread of the excruciating quiet periods. Weapons are abundant in the game, but each one comes with several caveats introducing an engaging puzzle element when managing the arsenal you have. Traditional weapons such as the revolver and shotgun work relatively well against the human and synthetic enemies (figure 2) but are entirely ineffective against the alien. As soon as the player fires these weapons, the noise immediately summons the alien, forcing the player to flee and hide. Fire-based weapons such as the Molotov and the flamethrower which are ineffective against the second most prevalent enemy, the synthetics (figure 3), are very effective at momentarily chasing the alien away (figure 4). This armament puzzle continually keeps players on their toes, adding to the suspense that occurs whenever the alien is crawling through the vents waiting for its opportunity to pounce at a moment's notice. Additional to the suspense created by this puzzle is the limited effectiveness of the flame-based weapons against the alien. When the player manages to scare the alien off with a well-timed flamethrower burst, it runs away only to return more aggressively after the short lull, further adding to the overall suspense the player needs to endure. The moments of silence between each confrontation seems to be *Alien: Isolation's* primary driver of suspense. The alien's erratic behaviour is the primary reason for this suspense, in that the player is unable to predict when the alien will appear, making each step unnerving.

Call of Cthulhu: Dark Corners of the Earth, Bethesda Softworks and 2K Games (2005) is a first-person survival shooter that is similar to *Alien: Isolation* in that the game also requires you to use a vast arsenal of weapons to dispatch monstrous enemies in confrontations. It adds the same type of inventory management puzzle in that ammunition is always scarce, so you are forced to engage enemies selectively. When wounded during a pitched battle, you must tend to your injuries that when left untreated will have a direct impact on your ability to effectively control your avatar. In addition to the mechanics mentioned above, there is a sanity system that gradually adds visual and auditory hallucinations to your experience; the more severe your injuries are, and the more you look at monsters. When these effects are combined, they compound your inability to control the avatar, significantly increasing the suspense experienced. The increased lack of control, which was designed to increase immersion, can hamper the enjoyment that some players may have due to its intrusive nature. This is a contentious issue, however, due to the differing tastes of people and the niche nature of this game.



Figure 5. The top-most image is the perspective of the player's avatar in Call of Cthulhu, while the bottom image shows the perspective of the monster stalking the player.

When comparing *Call of Cthulhu: Dark Corners of the Earth* to *Alien: Isolation*, quiet times are used in a similar way with most of this time being clumped together in the introductory stages of the game. These quiet times allow you to explore the world while soaking in the atmosphere, story and the stakes involved. The player's task is to investigate the fictional coastal town of Innsmouth. The residents, which will eventually turn against you, are incredibly xenophobic when you arrive and shun most attempts at conversation.

Coupled with this is an entity that stalks you. When walking down one of Innsmouth's alleyways, a hidden trigger would start a prescripted event that strips all control away from you while simultaneously fading to the perspective of the unknown entity that lurks on the rooftops (figure 5). These random voyeuristic moments where total control is lost and you, as the protagonist, are being depicted as the prey being stalked by a predator significantly ramps up the felt suspense. The effectiveness of these events borrows a lot from the Lovecraftian cosmic horror tropes of an unknowable terror that lurks just out of sight. The only problem with them is their fixed nature. As soon as you restart the level and play through the same location, the same scripted events will trigger. The more you replay the same area, the less impactful these events usually are.



Figure 6. An enemy flanking the player's avatar in Alan Wake.

Alan Wake, Microsoft Studios (2010), which is a third-person survival horror shooter, has a more dynamic way of signalling to the player that a threat is approaching, boosting the suspense. The level appears quite large with relatively large playable areas and vast vistas that appear traversable. Even though the level seems to be a sizable playable space, the developers can invisibly direct your flow of movement without you noticing it as a direct requirement of the linear narrative story arc. The level, which is essentially one linear funnel from objective to objective, occasionally bulges out into a larger invisibly ring-fenced areas or designated conflict zones. These zones, which are populated with vision and movement blockers including trees and buildings, force you to encounter numerous groups of enemies attacking from several directions at once. The game has a unique way of signalling to the player that they have entered one of these dangerous areas. They combine three effects, including a post screen effect that distorts the entire world ethereally, strange moving shadows covering the whole level and a high-pitched howling sound. These effects significantly ramp up the suspense in these areas, especially in the moments when the enemies are not attacking. As soon as several "taken" enemies approach, the player immediately loses control and the camera that usually hovers right above the protagonist shifts to the perspective of the group of enemies approaching from a vulnerable, unguarded angle just out of view (figure 6). As soon as you regain control, and the camera resets back to the third person position above your avatar, the enemies have gained enough ground to have an advantage over you. I will argue that this mechanic is a form of cheating in favour of the enemy agents, but it has an immense impact on the suspense felt during these brief moments. What makes these even more terrifying is the fact that the enemies will keep on spawning, regardless of how many you dispatch making ammunition conservation extremely important.

One can, with reasonable certainty, assume that these two perspectives do have a substantial impact on how the player perceives the suspense when comparing the third-person perspective of *Alan Wake* with the first-person perspective of *Call of Cthulhu: Dark Corners of the Earth*. I can, considering the experiences I had playing these two games, argue that the brief moments of reflective in-action, where the player loses all control for a few moments, feel equally suspenseful to me.

Additionally, the voyeur camera from *Call of Cthulhu: Dark Corners of the Earth* is an exciting game mechanic that can be exploited in a more dynamic way to induce suspense repeatedly. It makes the player aware of a sinister force that stalks the players, but not what its agenda is or even how it appears. When this voyeur camera prevents players from controlling the actions of this entity directly or prevents them from manoeuvring their controlled avatar out of its way, the resulting disempowerment and uncertainty could significantly boost the suspense felt by players. In their 2004 article *Helpless Spectators: Generating Suspense in Videogames and Film*, Jonathan Frome and Aaron Smuts conclude that "limiting interactivity at key points" can turn players into "helpless spectators like those that watch films" (31).

The efficiency of these mechanics will not be the central focus of this study, however. The tool introduced by this paper will merely enable researchers or anyone with a keen interest in this field to test these ideas out. The tool forms part of a video game that will be developed as part of the creative practice component of this research and will contain several adjustable variables that will enable interested parties to test how disempowerment and uncertainty affect human players. The tool does not provide the functionality to test these emotional responses, however. It merely acts as a stimuli generator that can be used in conjunction with additional tools that test human biological responses.

The main testing variables that this tool will provide will include the following:

- The number of "Monster Camera" event spawns.
- The duration between each spawn.
- The duration until the antagonist stops exploring the level and starts getting aggressive.
- The amount of time that will have to lapse before the antagonist gets more aggressive.
- The amount of time a player can stay in a darkened location until the antagonist attacks.
- The amount of time a player can remain in a lit area until the antagonist destroys a light source.
- The amount of time that must elapse from the moment the "Monster Camera" spawn system activates and the moment the system drops a key in the level that allows the player to escape.

The game will be called "*Interloper*" and set in an underground parking garage in the 1980's. It will be a single level game that will spawn suspenseful events from a limited pre-established library and modulate them to appear unique with each encounter. Except for some introductory and concluding areas, the playable gameplay area is entirely open-ended. The premise is simple; find the security guard's keys and open the steel gate, which leads to the street, before the evil entity kills you. The entire narrative sequence will be explored later on in this paper. It should be noted, however, that for a better understanding of the themes explored and mechanics employed, that the game should be played before the remainder of this paper is read in its entirety.

1.2 Objectives

The objective of this research project is to develop a video game that will serve as a versatile tool that researchers can use to test several hypotheses regarding the origins of suspense and how people are affected. These hypotheses will not be covered in this paper, however. It will be up to each researcher to determine how the tool will be used to conduct their research. It should be noted that this tool will only generate visual and auditory stimuli that could affect the audience when experiencing it. It also does not have facilities built into it to test or measure any human responses to these stimuli. These facilities will have to be developed separately by anyone interested in using this tool.

The video game "*Interloper*", with the embedded research tool mentioned above will be the creative practice-based output for this research project. Several methods designed to induce suspense indirectly will be combined into a coherent, usable whole and added to the tool. A video game environment will be populated with events from this tool using a procedural content generation method that will be modulated to make them appear unique with each encounter.

Examples of PCG include "Pseudo-Random Number Generators", "Generative Grammars", "Image Filtering", "Spatial Algorithms", "Modeling and Simulation of Complex Systems", and "Artificial Intelligence" (Hendrikx et al. 21-24). This project will use a variant of the "Search-based Procedural Content Generation" method advanced by Hendrikx et al. and instances of the "Pseudo-Random Number Generators" mentioned by him. These methods will be detailed later in the paper.

The following processes are required to achieve the research objectives:

- Develop a novel method that tracks the movement of the player in a video game environment that provides the suspense building tool with valuable information that will enable it to spawn suspenseful events that could stimulate a response.
- Use disempowerment and uncertainty as the primary qualifiers that will determine whether a spawned event is considered suspenseful. The player should be able to infer the intent of the antagonist from its actions but be left uncertain as to how the threat will manifest physically.
- By combining the player's inability to physically see an embodiment of the antagonist with a moment of disempowerment that show the antagonist's sinister intent from its perspective should enhance a sense of uncertainty, and by extension cause suspense.

The practical implementation will not be built to test a player's response to the spawned events, which will be the purview of future research. This restriction is due to the limited scope of this research project. The objectives of the practical implementation are as follows:

- To develop a convincing horror video game environment that follows the principles of a suitable model of suspense. The game will be set in a dimly lit underground parking garage as it is a modern horror setting that is used in both Western films including *P2* (2007) and *Scream* 4 (2011) and Eastern films such as *Midnight Garage* (2015). The universality of this setting should work well as a backdrop for "*Interloper*". Western films are made in Western countries such as the United States of America. Eastern Films are films made in Asian countries, including Japan and China.
- To determine whether it is possible to create a suspense-building tool inside this video game that makes use of a limited subset of suspense-building elements.
- To determine whether it is possible to modulate the events spawned by this tool in unique ways to make each appearance unique.



Figure 7. A scene from "Interloper".

The central outcome of this research project is a practical artefact that is a contribution to knowledge in and of itself and requires a methodology that enables this objective. Design Science will be used as the methodological foundation on which the creative practice implementation will be built. Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes Jr's 2015 book *Design Science Research: A Method for Science and Technology Advancement* sets out a thorough outline of the various methodological applications and will be used to explore the best approach. Dresch et al. describe Design Science as a pragmatic science that seeks to either "improve existing systems" or "solve problems" in a generalised way that facilitates the development of new "artefacts" that contribute to knowledge or "improve theories" (56-57). Design Science "operationalises" research when an "artefact" is the objective (67). Scientific rigour is achieved by making the artefact "pragmatically valid", which means the artefact will work as intended (57).

According to Dresch et al., "natural and social phenomena" are the purview of traditional science. In contrast, Design Science inserts an artificial object into the natural realm, influencing and being influenced by its laws directly (58-59). The primary goal of Design Science is the creation of an artefact that is based on research that was conducted scientifically (Dresch et al., 61). The artefact that this research will introduce is the game "*Interloper"*. It will adapt several theories into game mechanics and present them in a neat package that researchers can use to put these very theories to the test and perhaps even expand on them.

"Abduction" is the scientific method used in Design Science (61). Dresch et al. describe "Abduction" as a "creative process" where hypotheses are developed to explain certain phenomena (61). Testing these hypotheses can be conducted in future scientific experiments, so it is not a requirement during the Abduction process (61). Other methods can,

however, be combined with the Abduction method as required. A "Deductive" method can, for example, use "previous knowledge" to contribute to the understanding of an artefact that was created (Dresch et al., 62). "*Interloper*" will make use of the abduction process to present several hypotheses regarding the process of suspense and place it within a tool that can put these hypotheses to the test. A deductive process will be employed to apply previously established knowledge to the delivery of these abductive hypotheses in a convincing way.

Dresch et al. explore the methodology employed by several authors to compare how they approach the implementation of the Design Science Method. Due to the relative simplicity of the problem, the most streamlined method will be chosen and applied when developing the practical component. The "Design Cycle" methodology explored by Dresch et al. appears to be the most applicable.

The Design Cycle methodology was set forth by Vijay Vaishnavi, Bill Kuechler and Stacie Petter, eds. in their 2019 article *Design Science Research in Information Systems* and is an improvement on Hideaki Takeda, Paul Veerkamp and Hiroyuki Yoshikawa's method in their 1990 article *Modeling Design Process*. The method consists of five steps, according to Vaishnavi, Kuechler, and Petter, editors (11).

The first step is the "Awareness of Problem" step and requires an awareness of a research problem that requires a "problem-solving focused" solution and will, according to the authors, exist in the form of a proposal. How the product will be evaluated will also need to be addressed in this step (Vaishnavi, Kuechler, and Petter, editors 11-12). The proposal that was submitted previously should fulfil this requirement. The problem to solve is in the creation of a video game that convincingly uses disempowerment, reflection and uncertainty to produce a suspenseful experience that is perpetually unique each time that it is played.

The second step, according to Vaishnavi, Kuechler, and Petter, editors, is the creative "Suggestion" step and it is crucial in this step that a "Tentative Design" starts forming using elements that are either new or that already exists (12). The "Interloper Overview" chapter will formally exist as this research project's "Tentative Design".

"Development" is the third step and is where the physical embodiment of the proposed design and algorithms are created to put the proposed design to the test and show that it is indeed valid (12). The video game "*Interloper*" created for this research project will be the "Development" instantiation of the methodological design and will be explored in the "Development Breakdown" chapter.

The fourth step is the "Evaluation" step (Vaishnavi, Kuechler, and Petter, editors 12). According to the authors, it is essential that the artefact follows explicit instructions and criteria outlined in the proposal and that any quantitative and qualitative deviations are noted (13). The Evaluation step also requires an "analytic sub-phase" that generates fluid hypotheses about how the artefact might affect behaviours when introduced to an audience. This fluidity is at odds with the "positivist stance" of most research in that it does not generate a fixed answer to a hypothesis that usually concludes the research (13). Design science is recursive in that the evaluation, construction and eventual operation of the artefact feeds back into the Suggestion step, which Vaishnavi, Kuechler, and Petter, editors, terms "circumscription" (13). The "Evaluation" step usually leads to changes to the design or additional questions regarding the utility of the design. This interrogation sometimes leads to additional research and updates to the "Suggestion" and "Development" steps (13). The "Design Improvements" chapter dealing will cover this step in its entirety. This chapter will also generate several "fluid hypotheses" that will predict the possible outcome of this tool when introduced to players.

"Conclusion" is the last step, according to the authors. It is usually reached when the "research cycle" is completed, and the minimum requirements are met where the researcher deems the artefact "good enough" after several revisions. According to Vaishnavi, Kuechler, and Petter, editors, the compilation of a written record of the processes used to reach the final product, and the placement of knowledge generated into two different categories will occur. "Firm" is the first category and refers to facts that are learned and can be applied repeatedly (13). "Loose ends" is the second category and refers to abnormal results that can lead to future research (13).

Finally, the results of the study need to meaningfully contribute to knowledge, according to Vaishnavi, Kuechler, and Petter, editors (13). This contribution is achieved through "reflection" and "abstraction" (13). The researcher needs to reflect on the knowledge that was gained, the processes that functioned and those that failed. Next, the researcher needs to draw "broad and generally applicable conclusions" from the acquired knowledge through abstraction (Vaishnavi, Kuechler, and Petter, editors 15). This step will be covered in the last chapter "conclusion", which will explore what was learned in the development of the game "*Interloper*". It will also reflect on the knowledge that was gained when developing the game.

1.3 Clarification of Terms

UE4 / Unreal Engine 4

Unreal Engine 4 is a game engine that was developed by Epic Games. User can use it for free when the product is non-commercial. When the product, which was developed using UE4, is sold, royalties are due to Epic Games.

Game Engine

A game engine is a software platform that contains several features that allow a developer to quickly develop a video game without having to develop the engine first.

Baked Lighting Information

Unreal Engine 4 simulates physically accurate lighting by baking this information into a new texture that is in a separate lightmap layer on each mesh in the level.

Level / World

A level is a space that player can explore and interact within a game. It contains various elements that constitute its contents, including meshes, sounds and simulated lighting.

Mesh

A mesh is the physical representation of an object that exists in the game world or level. It usually consists of a list of points in a three-dimensional space, that when combined, creates a mesh of faces that are used to represent the object in the level.

Non-linear Environment

A non-linear environment is an environment that does not force a player down a specific route or hallway. It provides an open environment that allows the players to choose their direction of travel and the order in which they perform actions. A sandbox can be used as an analogy for this type of environment.

PC / Personal Computer

A personal computer is a device that consists of several electronic components that work together to provide an environment for software to function. Video games run on personal computers.

Real-time

It is a state in software that denotes live computation. Real-time computation in games usually means that new computations are executed every frame.

Texture

A texture is the visual representation of real-world surfaces, such as a brick wall, that is applied to various meshes in the game world.

2. Suspense

The practical output requires an efficient model of suspense upon which the various suspense building components can be built. The approach adopted will mix a variety of sources into an efficient foundation that will fit within the requirements of the creative practice.

2.1 Suspense versus Tension

Before securing the model that will be used as the foundation for the practical component, the difference between suspense and tension needs to be clearly defined first. It can be argued that a wide variety of sources can cause both suspense and tension. These sources can include certain types of music, sporting matches, the entire gamut of both film and television genres, some novels, and even video games. Negative suspense and tension will be the only type of suspense and tension that will be examined when compiling the practical output due to the horror genre being the context within which "*Interloper*" will be set. The other causes of suspense and or tension do not fall within the scope of this project.

In their 2015 article *Toward a general psychological model of tension and suspense* Moritz Lehne and Stefan Koelsch posits their definitions of suspense and tension. According to the authors, suspense can be defined as the anticipation a person feels when, in a narrative, two "opposing outcomes" are possible (2). It can be argued that an element of danger is usually involved when these two outcomes are established.

The authors define tension as the "diffused state of anticipation" a person feels when the anticipated event remains unspecified (2). When relating to tension specifically it can, therefore, be argued that tension is the physical manifestation of the stress that ensues when a person encounters a suspenseful scenario.

Suspense will, therefore, be used to as the primary descriptor when dealing with the scenarios depicted in "*Interloper*". Tension will merely describe the biological mechanism that causes a person to react when a genuinely suspenseful situation is encountered.

2.2 Modelling Suspense

The practical output requires an efficient model of suspense on which the various suspense building components can be built. Moritz Lehne and Stefan Koelsch's model of tension and suspense in their 2015 article *Toward a general psychological model of tension and suspense*, will be used as the primary model. This model will form the basis of the practical output as it fits the mould of the creative practice the best. Other models will, however, be investigated to see how it either integrates with or supports Lehne and Koelsch's model.

There is a universal theme that is central to suspense, namely the "emotions of fear and hope" (Lehne and Koelsch, 6). In their general model of tension and suspense, the resultant expectation emanating from an initiating event, which in conjunction with previous knowledge, leads to an "outcome space" that determines how much the "best-case scenario" diverges from the "worst-case scenario" (7). The amount of suspense generated by said initiating event is determined by the "emotional valence", or the divergence of "affective values of anticipated events", between the "best case" and "worst-case" scenarios, also described as "hope" and "fear" (7). Lehne and Koelsch further note that the areas of the brain that are stimulated during "predictive processing", such as with "expectation" and "anticipation", also causes tension (8). Moritz Lehne and Stefan Koelsch's model should be able to induce "negative tension" when used within a horror context (6). The model is, therefore, the ideal foundation on which the practical implementation can be built.

H. Porter Abbott defines suspense similarly in his 2001 book *The Cambridge Introduction to Narrative*. According to Abbott, an audience requires closure when a narrative "conflict" arises and that a state of "imbalance" usually precedes the closure that audience members seek, which adds to the suspense (53). The other component that enters the equation is "surprise" and that a narrative usually oscillates between suspenseful moments and surprises. Abbott further mentions that the narrative does not always end positively. A bad ending is sometimes satisfactory on its own (53). Narratives use specific "proairetic codes" to either satisfy the audience or to frustrate them (Abbott, 54). The events that take place sends out signals, and those signals give the audience certain expectations as to the outcome of such signals (54). When the expectation is met, closure is the result. Frustration is the result when the expectation is missed (54). These signals will be integrated into "*Interloper*" by tracking the player's actions and spawning the relevant "Monster Camera" events and environmental responses that frustrates the player's progress.

Richard A Doust posits a domain-independent model of suspense in his 2015 PhD thesis, *A domain-independent model of suspense in narrative*. He integrates William F. Brewer and Edward H. Lichtenstein 1982 model of suspense into his model, similar to what Lehne and Koelsch did to construct their model. According to Doust, Brewer and Lichtenstein's model states that suspense requires an "Initiating Event" and an "Outcome Event" that a protagonist strives to reach and several sub-events, designed to frustrate the desired outcome, which is placed in between (86). Doust expands on Brewer and Lichtenstein's model by adding "constructionist narrative comprehension theory". Doust also analyses suspense by comparing it to a simple action in the sport of curling that comprised several suspenseful elements (72). When the puck is shot towards its intended target, Doust suggests that the observers will do the following:

1. They will track the movement of the puck.
2. They will continuously check the puck's direction or if it is slowing down.
3. They will hope that the puck stops near its intended target.
4. They will try to force their will on the puck to try and stop it at its intended target.
5. They will be fully aware that the actual result will be imminent.

Further reading of Doust's paper suggests that these events can be referred to as "narrative threads" or limited story events that can be woven together to create a suspenseful story structure (84). The model requires background knowledge of some events that both conflicts and "disallows" each other and acts as a narrative frustration element (105). Finally, Doust points out four variables that can be tweaked to calculate the suspense factor for each "narrative thread". These four variables are the following:

- "Importance", which establishes the stakes involved in completing the story.
- "Foregroundedness", which determines the level of importance a narrative thread has in the minds of the audience.
- "Imminence" is the do-or-die moment where the danger is so close that it threatens to either complete the thread or interrupt it.
- "Confidence" determines whether the outcome of the narrative thread coincides with story events (106).

Doust's variables expand on Lehne and Koelsch's model in several ways, as shown above. The narrative threads are, for example, analogous to the events that will be used in the creative practice. The video game "*Interloper*", which was developed as the creative practice component of this paper, does not test the behavioural efficacy of these events, however. It merely modulates these events in minor ways based on several environmentally driven

variables, the narrative arc and the progression of the story. These features will be explored in-depth in the "Interloper Overview" chapter.

2.3 Effective Components of Suspense

In addition to the overall model of suspense, several will be introduced into "*Interloper*", which heightens the experience, and will be explored below. Only the elements that will assist in building negative suspense in conjunction with the "Monster Camera" events will be identified and listed.

Four elements will be included in "Interloper" as they were used extensively in the games analysed in section 1.1.

1. Lighting.
2. Colour.
3. Obscurity.
4. Sound.

2.3.1 Lighting

Dead Space, Alan Wake, Alien Isolation, and *Call of Cthulhu: Dark Corners of the Earth* all make use of dynamic lighting effects to present levels drenched in atmosphere. Including dynamic lights that the player can switch on and off or are affected by the player's action can have a powerful psychological effect on the reliability of light sources. In their 2007 article *Dynamic Lighting for Tension in Games*, Magy Seif El-Nasr, Simon Niedenthal, Igor Knez, Priya Almeida and Joseph Zupko explores how dynamic lighting is used in games to drive suspense (1). El-Nasr et al. point out that, without being physically accurate, all the algorithms used in "simulated lighting" to produce illumination in video games does an

excellent job of emulating "our experience" of actual illumination, which similarly affects our emotional reaction to it (2). In *Alan Wake*, for example, the brilliant light shafts of lone light sources in an otherwise hostile environment provide the player with the occasional respite. The game also subverts this reliance on light sources for protection. When a group of enemies pursues players, their first instinct would be to run to the nearest light source in the woods. When the player gets to some of these light sources, however, their light bulbs would pop, leaving players stranded. In *Alien Isolation,* the player's avatar is equipped with a flashlight so that it is easier to see in dimly lit areas. This tool is a double-edged sword; however, as enemies and the alien can spot players when they are using it.

"*Interloper*" uniquely makes use of dynamic lighting. The monster is afraid of light sources so that the player can use them to their advantage. Staying in the darkness will eventually lead to the player's premature death, so by continually shifting from darkness to lit areas can save the protagonist's life. Staying in lit areas is, similar to *Alan Wake* and *Alien Isolation*, a dangerous gamble, however. When players decide to stay in lit areas for too long, the monster will eventually get frustrated and disable one of the lights in the parking structure. Over relying on light sources to stay safe will push the monster to keep on disabling lights until all of them are disabled.

2.3.2 Colour

Colour and contrast effects are used to differentiate between the perspectives of the protagonist and antagonist. This effect was added due to the similarity between these two perspectives. There is also the potential to create additional suspense or tension using these colour effects. El-Nasr et al. identify several ways colour and contrast can cause tension, which adds to the overall suspense.

- When a “colour composition” changes in a specific way or order over time (4).
- Transitioning from high contrast lighting to low contrast lighting (5).
- Transitioning between cool lighting to warmer lighting during a “peak moment” (5).
- Changing the contrast between shots (5).
- Changing the “affinity” or colour saturation between shots (5).

El-Nasr et al. note that lighting designers use the contrast techniques in their shots to emphasise a transition in “dramatic intensity” which can evoke an emotional response from an audience (5).

Contrasting shots will be used in the following way in “*Interloper*” using post-processing effects.

- Colour affinity will change between the player’s perspective and the monster’s perspective. When the player is in control, the saturation is reduced. The monster’s perspective will, in turn, boost the saturation.
- Cool blues are the primary colours used in the level and are visible when the player is in control. The monster’s perspective transforms these colours into a harsh red colour.
- The scene seems washed out while the player is in control. When the monster is in control, the contrast is boosted substantially.
- Every time a “Monster Camera” event initiates, the colour composition mentioned above changes making each state very distinct looking.

2.3.3 Obscurity

El-Nasr et al. mention that obscurity is one way these patterns are used in games, especially survival horror games (6). Obscurity enhances "vulnerability" or "uncertainty" in players which, according to El-Nasr et al., means that "clear perception" and situational knowledge is either incomplete or reduced (6). The lack of knowledge created by certain types of obscurity including "darkness", "fog" or architectural "occlusion" can have an emotional impact on the audience (El-Nasr et al., 6). The element of obscurity noted by El-Nasr et al. harkens back to the influence uncertainty has on suspense explored previously. Obscurity will also be integrated into "*Interloper*" in two ways. Firstly, darkness will be the central gameplay mechanic. The protagonist can see better when the lights are on, so the objective is to stay in the lit areas, while the antagonist can see better when it is dark while being blinded by the lights. To compensate, the antagonist will occasionally attempt to disable them one by one to see. When all the lights are off, a swift end to the player's avatar is close. The antagonist will further make use of obscurity to hide from the player, so the player will never know what lurks out of sight. Even though the level is relatively open, with the player able to see the entire space at a single glance, space will be populated with pillars and procedurally placed cars. These cars and pillars will provide the obscuring elements that the antagonist can use to hide behind.

2.3.4 Sound

Alan Wake and *Call of Cthulhu: Dark Corners of the Earth* distorts the sound made by human characters to make them sound unnatural. Mark Grimshaw explores the use of uncanny sounds in horror games in his 2009 article *The audio Uncanny Valley: Sound, fear and the horror game*. Grimshaw notes that fear is the primary emotion linked to the Uncanny and that it can be exploited to make effective survival horror games (1). The "Monster

Camera" system will be combined with a predatory growling sound to make the monster sound intimidating. According to Grimshaw, sounds above five kilohertz and below two kilohertz can cause unease, especially when such sounds are run through a high-pass filter (2-3). These pitches will be combined to form the sound of the monster and will be boosted whenever the monster moves in for the game-ending attack.

A breathing system will be introduced that will track the player's progress on the suspense curve. This system could subconsciously drive the players to hasten their search for the key.

The uncanny will be introduced to make the breathing effect sound more unnatural after every "Monster Camera" event. Grimshaw notes that some Eastern films distort familiar sounds to evoke the uncanny (3). The effectiveness of such sound distortions does, however, require the sounds to be placed within a horror context where a threat is apparent (3).

Three breathing speeds will be introduced into "*Interloper*" to signal the desperation of the situation to the player. Grimshaw notes that by distorting and dynamically shifting the resolution of sounds, it will prevent players from getting used to these sounds (5). The distorted audio's resolution should also never match up with the visual quality that is presented (5). A low-pass filter will also be used to make breathing, and environmental audio sound progressively more muted. Doing this will bring more attention to the task of finding the key before the monster attacks.

3. Disempowerment

Disempowerment in horror is another major theme that will be explored below and is integrated into the video game "*Interloper*".

Tanya Krzywinska investigates the use of horror in several video games in her 2001 article *Hands on Horror* (12). Games are different from cinema in the way they focus on action (Krzywinska, 12). Krzywinska further states that this focus on action can take cinema's ability to extract human emotions that are both "kinematic and emotional" and enhance it (12).

3.1 Manichean moral duality

Both horror games and horror films make use of a "Manichean moral duality" to structure content. The model was invented by the third to fifth-century Manichean sect, who described the universe as a battle between good and evil (13). Krzywinska suggests that this duality is intrinsic to video games. The duality produces a pleasurable experience when events continuously swing between "good" and "evil", "action" and "in-action" or "self-determination" and "predetermination" (13). This "oscillation" produces a rhythm that, according to Krzywinska, works particularly well when placed within a horror context by presenting an evil force that ebbs and flows around the player as occasional threats (13). This rhythm also serves as a balancing force that prevents the experience from devolving into a chaotic mess (13). According to Krzywinska, the Manichaeistic dualism manifests as an overarching goal where the player must bring order to a chaotic game world (13). This model is implemented into "*Interloper*" by giving the player a simple, relatable goal cast in disarray by a chaotic force. The antagonist should frustrate the player's ability to achieve what should be a straightforward objective.

Krzywinska points out that even though players can interact with a predetermined surface level of the dualistic Manichaeistic substrate, they are unable to directly interfere with the fixed foundational composition of it (14). A notable example of the Manichaeistic dualism at play is where the primary camera, through which the world is viewed, oscillates between a state of control and lack of control (Krzywinska, 16). The player can have full control over the camera movement one moment before it cuts to a cinematic, wresting control away from the player. The ability for game cinematics to take control away from a player works exceptionally well when paired with the otherworldly "metaphysical" force that takes control of events, shaping how it unfolds (Krzywinska, 16). This disempowering fluctuation is, according to Krzywinska, a great way to build suspense (16).

Krzywinska takes it a step further by looking at how it compares to the "I-cam" camera technique used in several horror films (18). The camera technique mentioned by the author consists of an alternating cut between a shot of the victim trying to escape and a point-of-view shot from the killer's perspective approaching this hurried individual (18). Krzywinska notes that when this technique is employed in games, it differs slightly from its intended use in films (18). In films, a shot from the viewpoint of the killer or monster usually promotes an uneasy relationship between the audience and said antagonist and can also act as a foreshadowing or forewarning mechanic (18).

When the game world is viewed using a first-person or point-of-view shot that switches to a first-person shot from the perspective of the antagonist, it complicates the relationship between these two agents due to the similarity between the two shots and the motivations of each agent (Krzywinska,18). This complicated relationship is remedied when looking at the context of each character's shot and the audience's inherent compassion for the protagonist (Krzywinska,18). It seems ideal to use the alternating shot modality as a

foundational element of the suspense building tool. It will have to be modulated with each appearance and combined with additional elements to make it more effective, however.

3.2 Fear and Hope

Carrying on from their claim that the "paradox of suspense" does not occur in video games, Jonathan Frome and Aaron Smuts disagree with another one of Krzywinska's assertions that helplessness is hinged on an "occulted" or predetermined "fate" to generate suspense in a horror context (14). They explain that helplessness is not solely limited to horror games in that audience members are similarly unable to affect film narratives (14). They do, however, agree that helplessness is required to build suspense and tension, mainly when it is used in conjunction with three additional elements namely: "fear, hope and uncertainty" (14). By placing the player in the shoes of a relatable protagonist moving through the drudgery of life without any specific superpowers inherent in some modern video games, it could play exceptionally well into the notion of a helpless victim trying to escape. When players are not equipped with the latest weaponry, quick-wittedness becomes the only tool available to overcome proximate threats. When the player inhabits a kindred character, it should make the experience more frightening, but overcoming it even more satisfying.

According to Frome and Smuts, the uncertainties caused by fear and hope can be measured in several ways and lead to differing levels of suspense (15). The likelihood of a specific outcome is an essential component of uncertainty and a reliable driver of suspense (Frome and Smuts, 15). The authors note that a strong relationship exists between an uncertain outcome, the stakes involved with such an outcome and the suspense that results from that relationship (Frome and Smuts, 16). The example given by Frome and Smuts shows that a similar amount of suspense can be generated when the stakes are high, and the uncertainty is low or when the stakes are low, and the uncertainty is high (16).

Frome and Smuts also explore how suspense is induced in both film and video games. An emotional relationship between the characters and the audience and an unwanted approaching event that is more likely to occur is required to build suspense in cinema (17). Helplessness is also vital to a suspenseful experience in film. When an audience member is fed information of a potentially catastrophic event that awaits the protagonist, it usually becomes exceptionally suspenseful when given time to reflect on the outcome, unsure what will happen (Frome and Smuts, 17-18). In "*Interloper*" this catastrophe will be fed to the player by subtly subverting a simple objective. Players will be fed information early-on via an in-game intercom situated in the elevator. This information will establish the context in which the antagonist is located and will guide players to their primary objective, which at face value seems easy enough to achieve by following the simple instructions laid out by the official over the speaker system. This simple instruction will then be subverted by thwarting the player's ability to achieve it. This subversion will be achieved through overt environmental cues that hint to the approaching calamity. These cues can include the main gate leading to the street closing before the player can reach it, and the lit room that leads back to the elevators closing and locking behind the player.

The authors note that the "paradox of suspense" occurs in films due to the audience's ability to imagine different outcomes, even when the result is already known (19). They do, however, argue that this phenomenon does not affect games at all. This assertion is made because the outcome might differ slightly due to the player's actions each time the player plays the game (19). Frome and Smuts do, however, concede that suspense decreases when a set outcome is known after repeat playthroughs (19). This suppression of suspense becomes even more pronounced when the player's skill with each playthrough improves (19). Humans do not have the multi-tasking ability to reflect on narrative events while taking part in it; this quells suspense (Frome and Smut, 20). To combat this, the authors suggest several methods

(20). The first is to include “desirable and undesirable outcomes” or high stakes that are measured against player performance indicators such as level progression, endpoints, collectables, extra lives and other ways to push a player to their ideal goal (21). If the player fails, the game usually ends, or the player encounters an adversary (21). Another suspense enhancing technique mentioned by Frome and Smuts is a timing mechanic (22). This method reduces the player’s control over the situation due to the knowledge of an “imminent outcome” that increases the stakes (Frome and Smut, 22). The time a player invests into a game can be used to increase the stakes as well. A player will experience suspense when a high potential for failure can result in the loss of a lot of invested time (Frome and Smuts, 22). Games also reward players for completing sections or levels by providing them with powerful upgrades or showing them enticing cutscenes designed to expand the main narrative further (Frome and Smuts, 23). “Interloper” uses the timing method mentioned by Frome and Smuts will be used. Several timers were introduced that triggers when the player performs specific actions or stays in certain areas too long. The game’s total length is also pre-determined when the delays between the various “Monster Camera” events are combined. When this combined time runs out, the players will die if they are unable to find a way out. By limiting the time a player can escape, the stakes are raised considerably. Failing to escape before the clock stops will reset the entire experience, raising the stakes further.

The author asks whether a game maintains suspense through uncertainty when the game is played several times, increasing their mastery in the process. One such method could be to vary the experience sufficiently, including changing the difficulty of the foes, to allow for unique experiences each time and boosting uncertainty in the process (Frome and Smuts, 26). Researchers using the tool should be able to achieve suspense generating uncertainty by inputting various durations between each “Monster Camera” event. These controls are available to players and researchers on the main menu. They will, however, not be able to

control what events spawn. Such events are more context-sensitive and follow the length of the overall experience.

Another method that Frome and Smuts suggest could create a suspenseful experience is to use the third dimension more effectively. Three-dimensional games allow the threat to come from any direction, significantly increasing uncertainty (28). The antagonist in "*Interloper*" will be an implied threat with a single game-ending attack. The game will make use of three-dimensional space to increase the uncertainty of its location. By coupling a disembodied sound to the triggering of the "Monster Camera", an increased sense of uncertainty should result.

The authors mention that to build suspense effectively; reflection is required (29). This contemplative behaviour is better achieved in films when compared to video games because the audience is not actively engaged in the narrative (29). The authors suggest that the player should be removed from all activity, allowing them to reflect on the narrative and "possible outcomes" (29). This method can be achieved in various ways, according to Frome and Smuts. One method is to take control away from a player briefly (30). This break in control allows the player to ponder the outcome of their actions (Frome and Smuts, 30). This break in player control matches up with the point-of-view camera technique mentioned by Krzywinska in that a player can reflect on the situation when the player's camera view switches to that of the antagonist, unable to control its movements.

After reflecting on the ideas posited by Frome and Smuts, project-related objectives were extracted and added as components in "*Interloper*". When control is wrested away from the player, suspense should be the result. Occasionally the player will lose control of the camera perspective used to navigate the level, and it will transition to a point-of-view cinematic of the antagonist moving around the environment, stalking the player and hindering

the player's progress through specific actions. This moment of passivity provides the player time to reflect on the situation and formulate the best course of action to escape it within the allotted time. When the situation becomes grave, these camera events should appear closer to the player, ramping up the tension significantly.

4. Interloper Overview

4.1 Narrative Overview

Before exploring each aspect of the “Monster Camera” system integrated into the video game “Interloper”, a narrative overview of each important event in the life-cycle of the entire game will be examined. This exploration will be done to make it easier to comprehend how all the systems work together. If any of the terms mentioned in this design breakdown is unclear, the “clarification of terms” section should be consulted for additional clarity. The “clarification of terms” section contains a precise breakdown of every relevant term and concept that is used in the systems of “Interloper”.



Figure 8. The player starts in an elevator.

The game will be set in the 1980s and will take place in a single floor underground parking lot. Setting the game in that period rules out the use of certain modern technologies such as cell phones and mobile computers, which is a subtle way to tell players that outside help is unavailable and that they have found a way out on their own. The parking garage, which will be the only level in the game, should also serve as the ideal environment for the game, due to its ability to be both unassuming and eerie. The player will start in a moving

elevator where a small amount of stage setting exposition is delivered via the intercom by the resident security official (figure 8). The small movement limiting space, which is the starting point of the game, makes it possible to deliver an introductory exposition dump while foreshadowing the disempowerment that lies ahead. As the player waits for the digital display to count down to the "B" symbol, a voice over the intercom states the following: "*Please note that this is the last elevator ride for the night. We're busy locking the building for the weekend. The street is accessible through the basement parking exit. If the gate is closed, you can ask the resident security guard for the key. You can find the security office at the far end of the parking structure*." This message should provide players with just enough information to know exactly where to go and what to do when they get there. The building is going in lockdown, so the player needs to leave immediately. As soon as the elevator reaches the basement floor, the doors will open and reveal the elevator antechamber.



Figure 9. The emergency exit is inaccessible to the player.

It will become immediately apparent to players entering this room that a double door leading to another exit is blocked off. The boxes shown in figure 9 were added as a forewarning element and a subtle way to tell players that there is no escape. Note the blue colour scheme used on the walls. This colour scheme is used throughout the environment and

should introduce additional colour tension when paired off with the red colourisation provided by the “Monster Camera” post-processing effect.



Figure 10. Players will see an automatic sliding that leads to the parking garage.

At the end of the passage, players will notice a sliding door that leads directly to the parking garage and a ramp leading to the street exit (figure 10).



Figure 11. The automatic sliding door opens when the player gets close to it.



Figure 12. The metal gate closes before the player can exit.

The game environment will guide the player to see the main objective first. This primary objective requires the player to escape through a massive gate, which leads to the street that prematurely closes before the player can escape. Making the objective visually evident to the player upfront should etch the primary objective of the game in the player's mind throughout the experience.

These sliding doors will open as soon as the player approaches them, but will also trigger the massive gate to close, blocking the player's exit. These simultaneous events were added to subtly signal two things, a significant action the player can undertake and an essential objective that the player must achieve. The action is that doors can open automatically, allowing the player to pass through (figure 11). The gate closing in the distance is the real prize and signifies the objective or the portal that the player needs to pass through to beat the game (figure 12).



Figure 13. The red light on the gate control shows that the gate is locked.

If players decide to approach the control panel next to the main exit gate they will discover a keyhole right below the red light (figure 13). This keyhole should provide another clue that a key is required to open the gate. If the player decides to press the “use” key, a message will pop up that states that a “key is required”.



Figure 14. The sliding door locks behind the player.

The moment players turn around, they will discover a driveway to the rest of the parking area and that the sliding doors leading to the elevator antechamber are closed (figure 14). Players will not be able to re-enter this room which adds to the subtle messaging that safe well-lit areas are being locked off and that no convenient escape is available.

After this introductory phase, the only recourse for the player is to explore the rest of the environment. This exploration phase allows the player to walk through the entire space without any direct threats present and gain a grasp of the whole control scheme unencumbered.



Figure 15. The main play space is open-ended and excludes hiding spots.

As soon as players reach the main parking area, it will become apparent that a large percentage of the space is visible and that hiding spots are kept to a minimum (figure 15). Nothing significant will happen until players reach the security office, which means they can explore at their leisure while mastering the control scheme. Apart from the starting area, which is more linear, the main play space is mostly open-ended and will challenge the notion that suspense only occurs in a carefully controlled funnel. The environment will mostly act as a contextual background against which the narrative events of the game will be set.

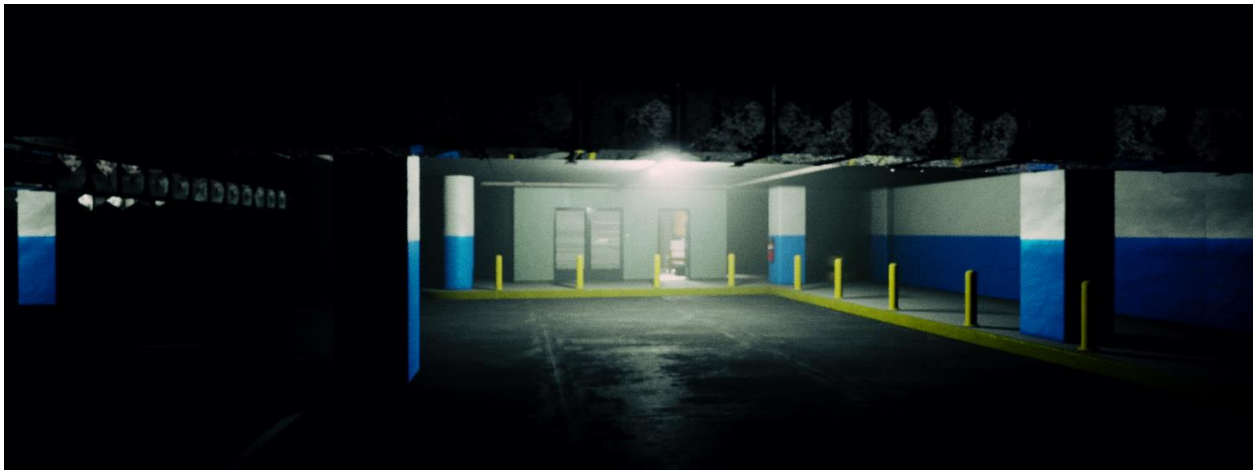


Figure 16. The security office is located in the opposite side of the parking garage.

The official that provides an exposition on the elevator ride down to the basement reveals that there is a security guard stationed at the security office in the parking space. Players need to visit this guard if they get stuck. Finding this officer, who holds the key to the main gate, will automatically become the next objective once the player realises that there is no escape.

The parking office is in the furthest most corner of the parking area from the point of ingress (figure 16). This distance will provide players enough time to explore the space, but it also has an additional function. The room will always stay lit. The player may stay in it while the monster roams around outside. While the player hides inside, the monster will keep on deactivating lights in the parking area. This in-action will make it extremely difficult for the player to escape. When the event tracker reaches the last spawn point on the suspense curve, the system usually spawns a “Monster Camera” that launches a deadly attack on the player roaming around in the parking area. If this event launches and the player is hiding in the office, nothing will happen until the player leaves.



Figure 17. Players will notice a problem at the security office when they approach.

As soon as players reach the office, they will immediately notice that something has gone awry (figure 17). This realisation should prompt them to enter.

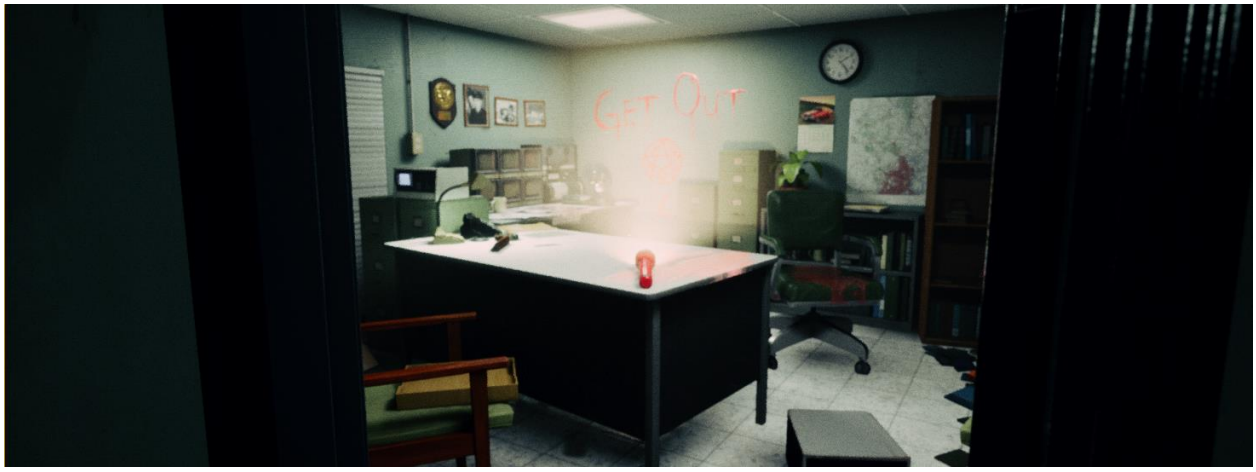


Figure 18. Players will discover a security office in disarray.

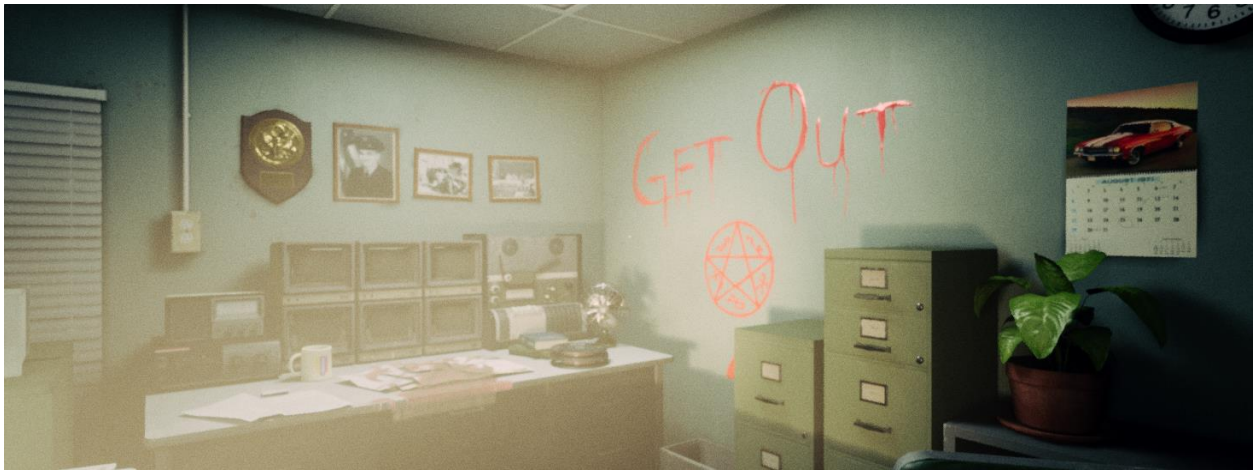


Figure 19. A flashlight on the desk illuminates ominous text scribbled on the wall.

Players will get a better idea of the nature of the problem the moment they turn the corner. The interior of the security office reveals a struggle that occurred recently (figure 18).

Strange symbols are scribbled in blood all over the walls, with the words “Get Out” featured prominently (figure 19). The office is in a state of disarray, and the security guard is missing. A flashlight, which the player needs to pick up and use, reinforces what the player needs to do by illuminating the bloody words on the wall. The delay to the very first “Monster Camera” event only triggers the moment the player retrieves the flashlight on the desk. This scene and the words written on the wall should make it clear to players what they need to do next.



Figure 20. Environmental story telling techniques reveal what happened to the security officer.

The office endeavours to make use of environmental storytelling to show how the monster snatched the security guard off the swivel chair (figure 20). The primary work desk was rotated outwards from its original position to show how the guard was pulled out of the office. Hand smear decals were also placed near the key holder on the wall next to the door to show how the security guard grabbed the gate key before being pulled out by the monster.



Figure 21. As soon as the player collects the flashlight on the desk, the monster activates.

Textual pop-ups that appear whenever they are required will provide the player with relevant information. This information includes when a player can interact with an object or what the player's immediate objectives are.

A hand symbol appears when the player is close to the flashlight while looking at it (figure 21). This symbol is the same hand symbol shown when the player looks at the control box next to the main gate. The symbol denotes that the player can interact with the object in question. As soon as the player presses the "interact" key, the player will absorb the flashlight. It was decided early on not to add unnecessary object handling animations to the game and make it function more like older games from a decade ago. The player is essentially a floating camera with the basic interactivity of a human-sized avatar.

The moment the player retrieves the flashlight, it signals to the system that the event tracker is now active and will start spawning "Monster Camera" events based on the values fed into the system. A message will also appear on the screen informing the player to retrieve the key.



Figure 22. The main gameplay loop is to find the gate key with the acquired flashlight.

Equipped with the red flashlight from the desk, the player will then start frantically searching for the key (figure 22). The key gets dropped in a random location in the playable space, and the system can be set up to only drop the key after a specified amount of time.



Figure 23. The monster camera takes over and seeks the player at set times.

Every once and a while, the player will lose all control. At this moment, the screen will fade to the perspective of the monster (figure 23). The system tracks the location of the player and the progress of the event array and spawns the most appropriate "Monster Camera" event. The starting location of the monster for the first event spawn will always be the same while subsequently spawning future events based on the last known location of this first spawn.

The game has a procedural suspense curve. This curve will be determined by a set of duration values that the researcher can enter into numeric fields on the main menu before starting the game. These values are linked to event spawns. As mentioned earlier in the paper, each spawn entered into the numeric fields on the main menu will trigger a "Monster Camera" event that is contextually linked to other values that activate when the player is in certain areas within the level.

Simon Egenfeldt-Nielsen, Jonas Heide Smith, and Susana Pajares Tosca explains, in their 2013 book *Understanding Video Games: The Essential Introduction*, that in many video games players usually progress through plots by satisfying a linear set of scripted objectives or events (Egenfeldt-Nielsen et al. 172). James Newman similarly argues, in his 2012 book *Videogames*, that narrative and gameplay works in tandem and that the former can only progress when the player interacts with sequential gameplay elements directly (Newman, 104). "*Interloper*" should allow for satisfactory narrative progression when spawning the "Monster Camera" events, which are tracked in the background. A procedural content generation technique, popularly referred to as PCG, can facilitate this requirement. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup explains, in their 2013 article *Procedural content generation for games: A survey* that this technique can judge the meaningfulness of content after being spawned.



Figure 24. The system drops an avatar that represents the player's last known location.

The system also drops a three-dimensional representation of the protagonist in the level whenever a “Monster Camera” event spawns (figure 24). This representation is used because the first-person controller that the player uses is merely a floating camera attached to a physics capsule, as mentioned previously. The monster also needs a physical representation of the protagonist to encounter in the level.



Figure 25. Pre-baked “stationary lights” are used for realism and interactivity.

The light emanating from the various light sources throughout the playable space is pre-baked into a lightmap (figure 25). A special “stationary light” technique is used that allows soft shadows to be baked into the geometry but also makes the light somewhat interactive. The colour and intensity of a “stationary” light source are alterable. The dynamic nature of these lights allows the antagonist to disable any of the fluorescent light sources illuminating the parking area.

“*Interloper*” uses colour in two ways. Firstly, how the physical assets are coloured and secondly by adjusting the post-processing effects and colour grading applied directly to the in-game camera. Light and colour contrast will be used extensively throughout the game. The use of contrast shows the difference between the experience of the protagonist and that of the monster. The antagonist operates in the dark, and its perspective needs to illustrate how it

can see better in dimly lit areas. An active “Monster Camera”, boosts the exposure and contrast. The system will also apply colour contrast similarly. When the player controls the protagonist, the viewed scene will be graded with an unsaturated blue tint. Alternatively, a saturated red tint will be applied whenever the Monster Camera is active. A slight visual distortion and a reddish highlight of the player’s avatar, whenever the antagonist sees it, are also added to make the antagonist’s prey stand out in the level.



Figure 26. When found, the player can pick up the gate key.

The key that the player needs to retrieve is quite small (figure 26). As soon as the player looks down at it, another “interact” hand symbol should appear, allowing the player to retrieve it when pressing the corresponding key.

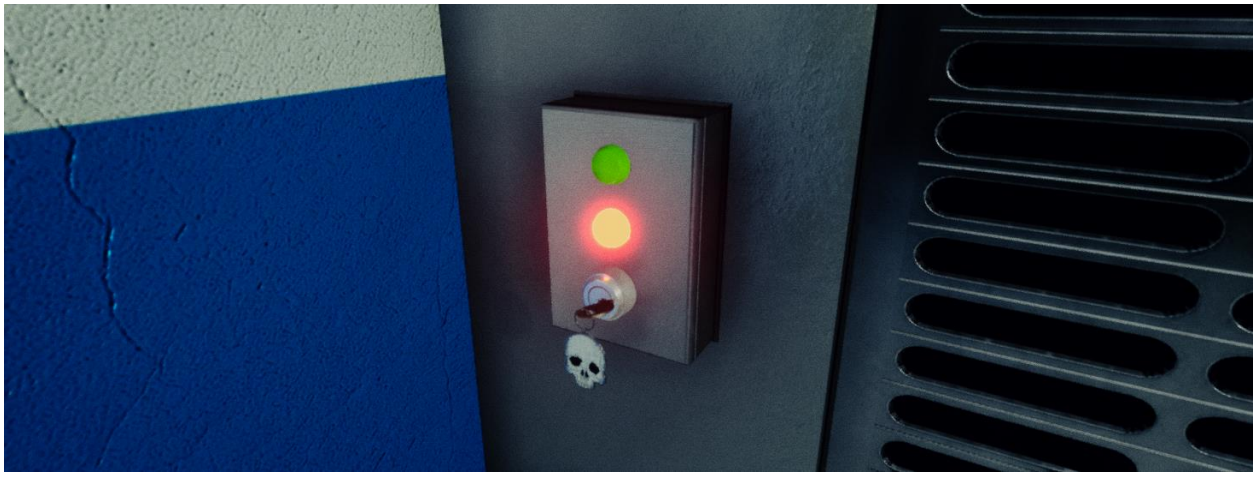


Figure 27. After collecting the key, the player can insert it into the gate control.

The same "interact" icon that the player saw at the control box previously will again become active when the player reaches the gate before the monster launches its final attack. As soon as the player presses the corresponding key, an animation will play showing how the key is inserted into the lock and turned (figure 27).



Figure 28. The player can exit the parking garage, as soon as the gate opens.

The main gate will start opening when the light turns green (figure 28). When the player passes through the front gate portal and walks up the ramp toward the street, the screen will fade to white – ending the game.

Monster Camera



Figure 29. Unreal Engine 4's editor.

This research paper accompanies a horror game developed to fulfil the requirements of a Master's Degree through a dissertation and a creative practice component. Epic Games "Unreal Engine 4" (figure 29) , which will be referred to as "UE4" in this paper, is a game engine toolset designed to simplify the development of industry quality video games and was used to develop "*Interloper*". The game code will contain a tool that enables researchers to measure suspense. This tool will be called the "Monster Camera". Researchers can customise variables in the game's menu system that allows them to procedurally spawn a predetermined amount of voyeuristic camera events throughout the experience.



Figure 30. A green navigation mesh.

Each event will flip between the first-person perspective of the protagonist, which is controlled by the player, to that of the antagonist while simultaneously stripping the player's control away. The system handles the types of camera events that spawn and the number of spawns separately. The event spawn types are linked to the player's location in the level while the player sets the number of spawns. An AI controller that makes use of a navigation mesh will control the camera movement of the antagonist. The navigation mesh updates dynamically, so when any changes are made to the environment, the antagonist will know precisely where movement is permissible. This navigation mesh will also facilitate the procedural placement of the key that allows the player to escape the level (figure 30).

The entity that stalks the player is sensitive to light, so the player's location relative to any light source has an impact on the type of event that will spawn. The events that spawn will always be contextual based on a player's distance to light sources. The antagonist will always attempt to strike players when they spend too much time in the darkness. The only way the player can break the antagonist's line of sight is to move through a light source. When a player spends too much time within the boundaries of a light source, the antagonist will always attempt to disable another light source within the level to limit the player's hiding spots.

Five possible event types can be spawned.

1. The antagonist walks around randomly, searching for the player's location.
2. The antagonist circles the player at a reasonable distance.
3. The antagonist moves closer to the player.
4. The antagonist disables a light.
5. The antagonist launches a deadly attack on the player.

As mentioned previously, the tool will track the player relative to the environment. The narrative arc is determined by the number of events that the player sets up when starting "*Interloper*" for the first time. This narrative arc also represents the "suspense curve". Due to the contextual nature of the event spawns, the narrative arc or suspense curve only controls two aspects of the experience:

- How intense the monster screams are between each "Monster Camera" event spawn.
- How rapid and disembodied the protagonist's breathing gets as the game progresses.

The system comes with three independent timers. The first-timer will determine how long the player will play after triggering the first "Monster Camera" event until the

antagonist's aggression starts increasing. The second and third timers will only activate once the first-timer concludes. Timer's two and three deal exclusively with how much time the player spends within lit and unlit areas. The researcher that makes use of this tool can specify on the main menu how long the player can stay in the various lit or unlit areas before a contextual "Monster Camera" event triggers. When the game detects that the player is moving through a darkened area, a timer will commence. The system will only spawn "Monster Camera" events that show the antagonist circling the players while they are still within this time-frame. When this time lapses, the fifth event spawns, which kills the player and ends the current play session. The only way the player can reset the darkness timer is to enter a lit area. Entering such an area disables the darkness timer, sets a Boolean value that indicates that the player is in a lit area and activates the light timer.

Resourceful players can use light sources in two ways. Firstly, they can be used to hide from the antagonist or, if the antagonist finds the player, break the antagonist's line of sight. Staying within the radius of a light source comes with risk, however. A timer activates as soon as the player enters a lit area. When the player is within the bounds of a light source, the antagonist receives a signal after a predetermined period.

It should be noted that this predetermined duration can also be specified when a researcher is setting up the parameters on the main menu. Once the timer exceeds the predetermined value for staying within the boundaries of a light source, the fourth "Monster Camera" event will spawn. The antagonist will proceed to disable a random light source within the level reducing the number of potential hiding spots by one. This continual reduction of hiding spots should add a puzzle element to the experience that will force the player to decide how long to stay within the comforting embrace of a light source and when to leave it. The antagonist will disable all the lights if the player refuses to venture out into

the darkened areas. The antagonist sees the light sources as obstacles that it mostly tries to avoid when pursuing the player. When the player passes through a light source, which resets the darkness timer, the antagonist will revert to event one. This first event will again upgrade to events two and then three if the player spends some time in dimly lit areas or spawn event four if the player stays within the confines of that light source.

The dynamic way that the event spawns are handled based on criteria such as distance to the player, its distance to a light source or the progression of the narrative arc shows that the antagonist has intent. Even though the way the antagonist's artificial intelligence is handled in a very rudimentary way, it could fool unaware players if researchers set these values cleverly.

The darkness is designed to obscure the antagonist, preventing the player from truly knowing its appearance. The lack of a visual representation might induce a hallucinatory reaction within players, especially when researchers inform players that there is indeed a physically embodied entity stalking the player in the darkness.

The fifth and final camera event takes place whenever the player spends too much time in the darkness or if the player reaches the final event spawn point established by a researcher on the main menu. This final event acts as an end game state and a narrative climax. Researchers will, unfortunately, not be able to set the length of this final event on the main menu. The "Monster Camera" event will be spawned in a random location within the level and proceed to the location of the player. When the antagonist reaches the player, the screen will fade to black, and the sounds of a violent struggle will ensue.

The player's controller consists of an empty capsule that assists with physical movement throughout the level and prevents the player from falling through the floor. The

player's controller does not contain a third-person representation of the player's avatar. Whenever the perspective changes to that of the antagonist, the system spawns a three-dimensional representation of the protagonist. The system spawns this avatar where the switch occurred. Placing an animated model will allow the "Monster Camera" to observe its prey.

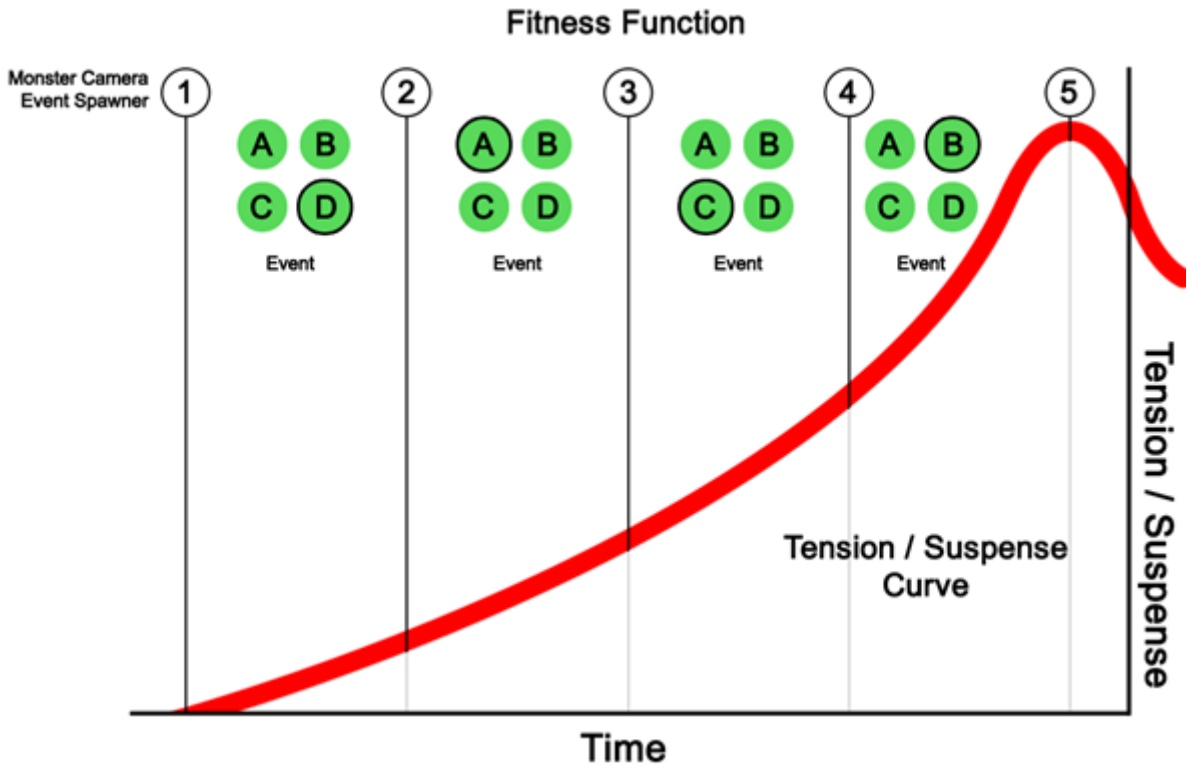


Figure 31. A suspense curve that shows how the PCG algorithm chooses the correct "Monster Camera" event based on the value of the fitness function.

The "Search-based Procedural Content Generation" technique advanced by Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley and Cameron Browne in their 2011 article *Search-based Procedural Content Generation: A Taxonomy and Survey* were chosen to facilitate the spawning of the "Monster Camera" events. It will be used to track a suspense curve in one direction and determine when to spawn the "Monster Camera" events (figure 31). The method proposed by Togelius et al. is an extension of the "generate-and-test"

method, which generates content based on a “fitness, evaluation” or “utility” function (3). The fitness function “grades” newly generated content, which informs the fitness value, and generates future content based on this value (3). The fitness function is a variable that dynamically changes based on the progress of the suspense curve and the player’s location within the level. When the variable of the fitness function corresponds with the value of an event spawn on the curve, a “Monster Camera” event will trigger irrespective of the player’s location and what the player is doing. When combined with logic gates that determine where the player is located in the level, this technique can be potent and spawn events that are more relevant to where the player is located within the level.

4.1.1 Monster Camera - Player Tracking

The system will dynamically track the player's position and update a corresponding variable. When a "Monster Camera" event spawns, the system notes the last known location of the player and spawns a stationary 3D character that denotes the appearance of the player's avatar.

4.1.2 Monster Camera - Placement and Movement

The "Monster Camera" will be placed randomly based on the navigation mesh generated from the environment (figure 30). The system will, however, keep track of the "Monster Camera" event's spawn location, so that each new spawn has a certain positional logic to it that keeps the player aware of the antagonist's last known location.

- Several AI characters will be created based off the base monster camera AI character.
- Each character will be set up with additional logic that will enable it to either track the location of the player or the location of a light it wants to disable.

4.2 Procedural Content Generation

The “Monster Camera” events will make use of a procedural content generation technique to spawn camera events at opportune moments.

Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley and Cameron Browne survey several notable procedural content generation (PCG) techniques in their 2011 article *Search-based Procedural Content Generation: A Taxonomy and Survey*. According to Togelius et al., PCG generates video game content “automatically” using “algorithms” (1). Content that can be generated includes “terrain, maps, levels, stories, dialogue, quests, characters, rulesets, dynamics and weapons” (1). They also propose a new PCG technique, coined “search-based procedural content generation” (1). Before Togelius et al. arrived at their proposed technique, they had to evaluate the most essential elements prevalent in PCG methodology, which would play a significant role in their new technique. These elements include “online versus offline” generation, “necessary versus optional content”, “random seeds versus parameter vectors”, “Stochastic versus deterministic” generation, and “constructive vs generate-and-test” methods (2-3). The method proposed by Togelius et al. is an extension of the “generate-and-test” method, which generates content based on a “fitness, evaluation” or “utility” function (3). The fitness function “grades” newly generated content, which informs the fitness value, and generates future content based on this value (3). Togelius et al. also discusses optimisation techniques and other sources from which the fitness function can be derived (4 -6).

The search-based procedural content generation technique will be used to spawn the “Monster Camera” events in “*Interloper*”. Researchers will be able to set the amount of “Monster Camera” events before launching the level. This amount will determine the shape of the suspense curve that controls a breathing mechanic that signals to the player how far the

suspense curve has progressed. Each event spawn increases the protagonist's breathing rate while increasing the strength of a low pass filter that is not only applied to the breathing sound but also the ambient sounds of the entire level. The amount of time between each event spawn can also be set when assigning the number of spawns. This single time variable will determine the amount of time between the current event and the next one. Each event brings the player closer to the bad ending unless the key is found that will unlock the exit and the good ending.

Augmenting a similar method

Various procedural methods were examined so that "*Interloper*" could be placed as a novel design within the nexus of similar works.

Phil Lopes, Antonios Liapis and Georgios Yannakakis's 2016 article *Framing Tension for Game Generation* is the one approach that resembles the method used in "*Interloper*" the most. The mentioned method augments another one that was advanced by Lopes et al. with several new improvements.

Their game dynamically constructs an environment and places tension inducing sounds so that the system can conform to a predetermined experiential frame. The system developed by Lopes et al. uses narrative structure framing to push "emotional progression" dynamically (206). It uses several "fitness functions" to drive a dynamic tension curve based on a "genetic algorithm" (Lopes et al., 207). The first thing the authors developed was a video game with a linear level structure designed to drive up suspense based on what they term an "intended tension curve" that effectively simulates the "fear, anxiety and stress" experienced when playing a suspenseful game (207). The results gained from this linear design then informed a procedural system that also makes use of the search-based procedural

content generation method coined by Togelius et al. in their 2011 article (207). What the system they developed does is make use of a tension curve that dynamically adjusts based on the genetic algorithm mentioned earlier and then measure a fitness function against this curve (208).

"Mutation seeds" are used to generate the rooms and place smaller elements, including doors and monsters (208). These rooms are directly linked to the tension curve and drive the playback of specific sound frequencies in each room based on the level of tension perceived by the player (208). The tension curve is scaled or adjusted based on the number of rooms spawned, the number of dead ends generated and the critical path (208). The amount of tension that a room is intended to represent determines what audio assets are played (208). A crowdsourcing method was used to determine how to effectively map the correct audio to each room (208). The crowdsourcing decided how a range of sounds was able to induce tension and determined that a loud, high-pitched sound above 5 kHz creates a "fearful" response (208). The procedural method introduced by Lopes et al. takes that set of sounds, ranging from a lower pitch to a higher tension inducing pitch, and places them in a particular sequence on the pre-generated critical path that the player traverses (208). This critical path is attached to the tension curve that is determined when the level is generated initially (208). According to Lopes et al. it is possible for the player to hear the pitch of the audio sample used in adjoining rooms, acting as a type of forewarning (209). The Lopes et al. method tries to model tension that rises and falls multiple times during an experience (211). The technique used in "*Interloper*" will add to the ideas advanced by Lopes et al. in the following ways:

- The suspense curve will be a single static curve formed by the number of events or trigger points set by the user at the start of the experience.

- The fitness function will track this suspense curve linearly and will not be allowed to reset or travel in reverse until the experience has concluded.
- When the fitness function travels down the suspense curve and hits one of the predetermined trigger points, an event will spawn in the level.
- Several internal and external conditions determine the types of events that will spawn. The player's location in the level will have the most significant impact on what events will spawn. The progress of the suspense curve will have a direct effect on the sound of the protagonist's breathing and how intense the applied low pass filter is.
- Each event trigger spawns a "Monster Camera" event that consists of an AI agent that navigates the level using a dynamically updating navigation mesh.

Hendrikx et al. distinguish between several discrete components that constitute a game that can be spawned using procedural content generation (4). Although numerous elements can be included, such as "game bits", "game space", "game systems", "game scenarios", "game design", and "derived content", the elements that will remain static will need to be distinguished from those that will be spawned procedurally (4-7).

The only elements that will be spawned procedurally include the following:

- Vehicles will be spawned and fall under the "game space" and "game bit" umbrellas. A pseudo-random number generation method will be used to spawn them and change several internal parameters, including their colour and wheel appearance. The dynamic placement of the vehicles will change the layout of the level slightly and add unique obstacles with each playthrough. The system will give a static spawn node to each parking bay, that will either allow one of the many cars to spawn or not.
- "Game systems" make the experience more believable (6). Light sources are dynamic and can be affected by the actions of the antagonist.

- “Game scenarios” that are either “abstract” or “concrete” will be spawned. These are the underlying systems that control how events are ordered (6).

Suspense is an implicit component of an explicit, concrete stimulus presented to the player in the world. The story-specific elements will remain static and provide the context that will add credence to the suspense-building elements.

5. Development Breakdown

5.1 Event Graph

The event graph is the main canvas used when developing blueprint scripts. Among the various blueprint types including level blueprints, widget blueprint, game instance blueprints, actor blueprints and others, two event types will always be available and immensely useful. These types are “Event Begin Play” and “Event Tick”. “Event Begin Play” simultaneously executes all the blueprints connected to it, the moment the level loads. This execution happens only once per session.

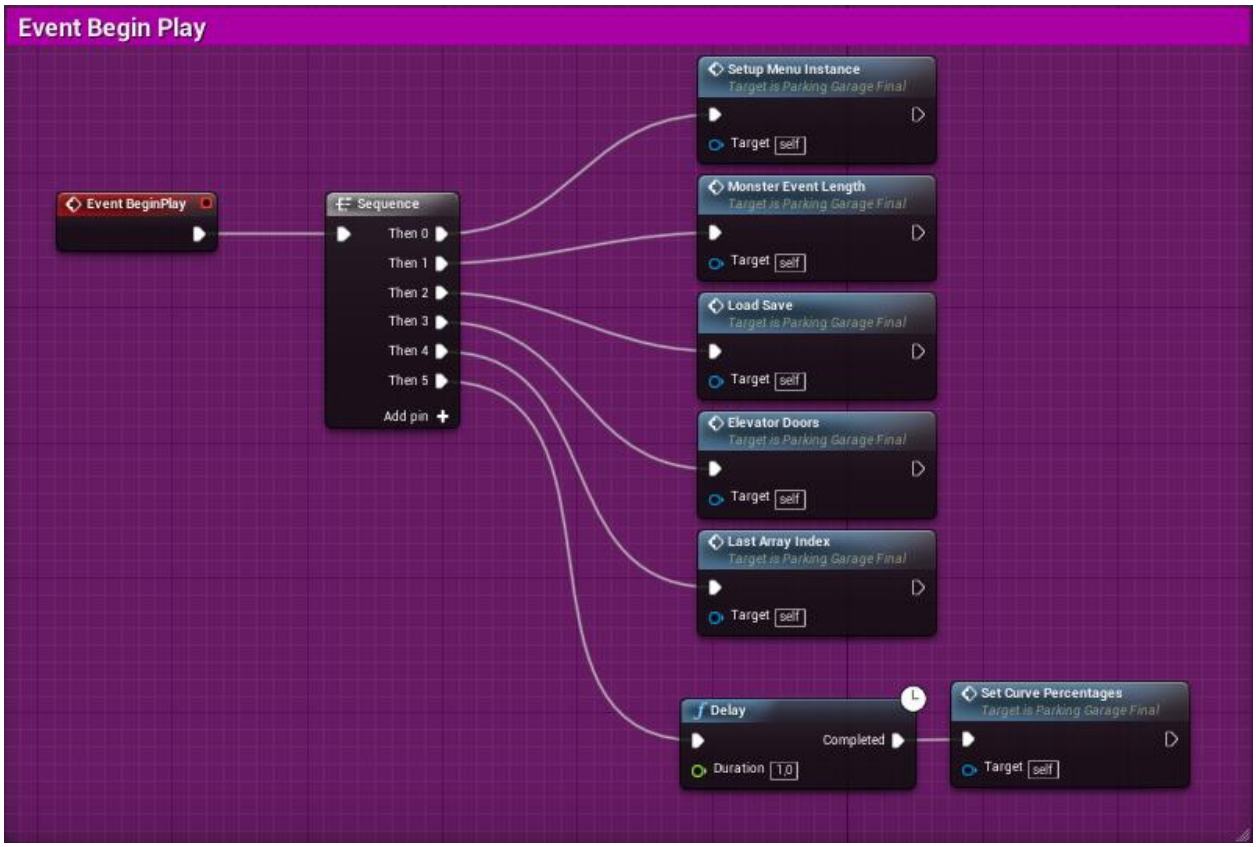


Figure 32. Event Begin Play graph.

Figure 32 shows an “Event Begin Play” and all the events that are executed the moment the level loads. Figure 32 is a handy reference when the origin of some of the blueprints explored below is unclear.

"Event Tick" executes every blueprint connected to it once every frame, which means that it will keep on executing until the level is unloaded. Depending on the frame-rate, this can occur every second. This execution node should not be abused due to its relatively high computational cost.

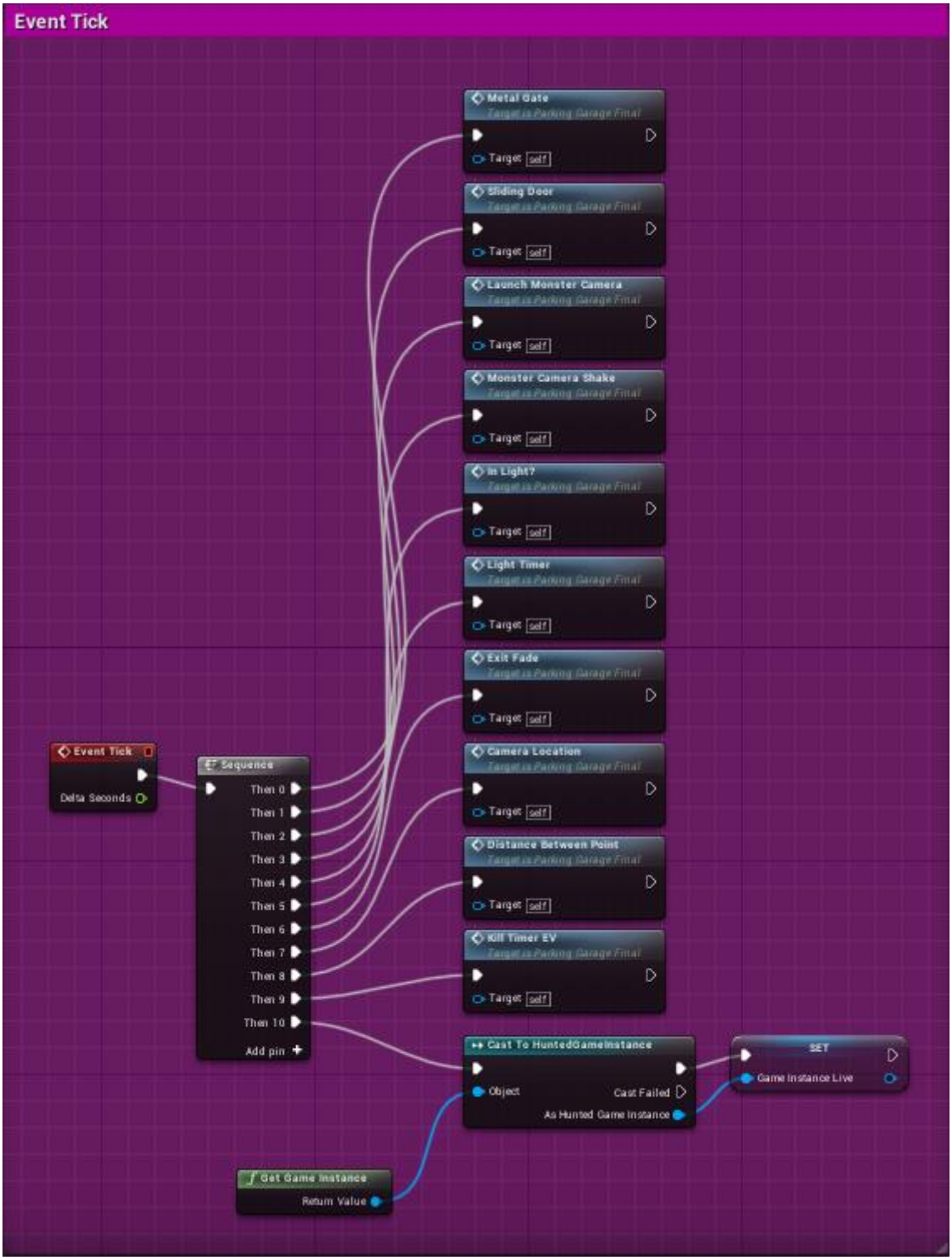


Figure 33. Event Tick Graph.

Figure 33 shows an “Event Begin Play” and all the events that are executed. Similar to the “Event Begin Play” reference, figure 33 can also be used as a reference when examining the blueprints below.

5.2 Main Menu



Figure 34. Intro Logo.

Figure 34 shows the first screen the player will see when “Interloper” starts up. As soon as the player presses any key, the “Event Manager” page will activate.



Figure 35. Monster Camera Event Manager.

The “Event Manager” page is shown in figure 35. The player can click on three buttons when landing on this page. There is the “Create New” button that lets the player set up a new sequence of events, the “Load Save” button that lets the player load the previous session, and the “Quit” button that exits the application once clicked.

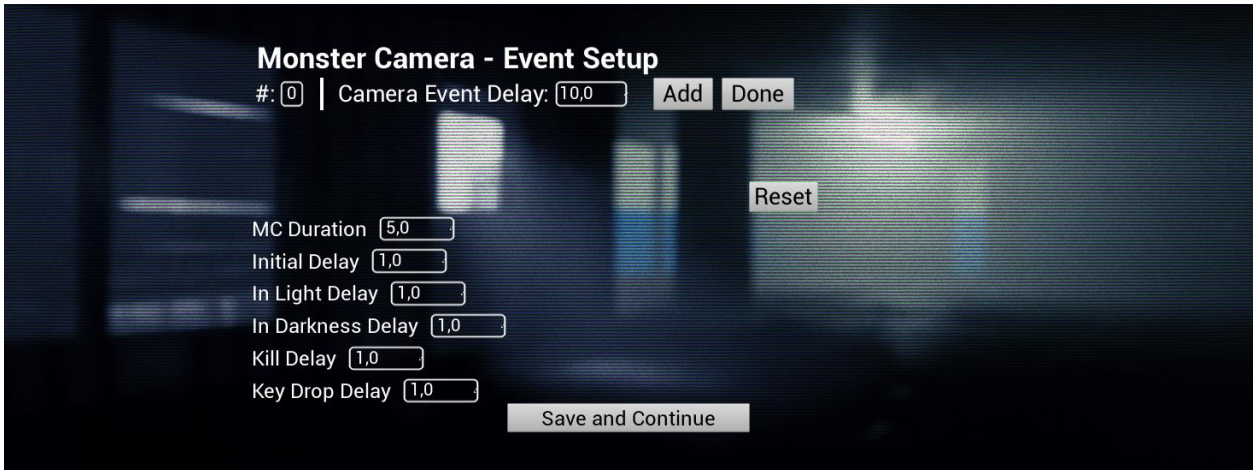


Figure 36. Monster Camera Event Setup.

Once the player clicks on the "Create New" button, the "Monster Camera - Event Setup" page appears, as shown in figure 36. This page allows the player or a researcher to enter several values that will affect the game.

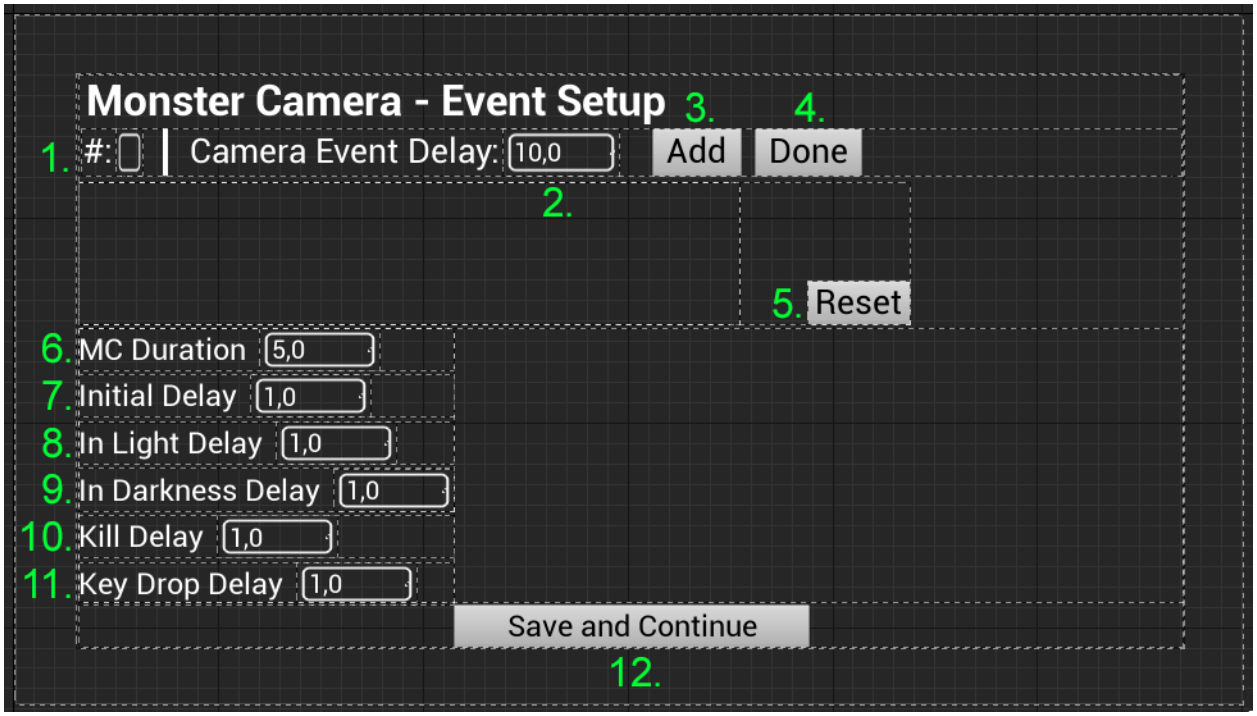


Figure 37. Event Setup Menu.

In figure 37, a breakdown of the “Event Setup” page is shown. The player or researcher is unable to alter the “#” input (1). This text box shows the event that is currently being edited. The “Camera Event Delay” input (2) is where the length of the event is entered. The delay takes place before the event triggers, so if a delay value of “10” is registered for the first event, ten seconds will elapse before the event is executed. The “Add” button (3) registers the value that was entered into the array. The “Done” button (4) is a test button that was used to check to see if the values genuinely registered in the array, so will be removed in the final build. The “Reset” button (5), resets the entire list so that a fresh collection of events can be entered.

The “MC Duration” input (6) allows the length of “Monster Camera” events to be specified universally. This duration is the amount of time that the antagonist is in control. The “Initial Delay” input (7) provides an amount of time to the “Initial Timer” as explored on page 202. Within this period, the antagonist merely wanders around the environment, seemingly unable to find the player’s location. The “In Light Delay” input (8) fuels the duration of the “In Light Timer” as discussed on page 202. This timer activates as soon as the player enters a brightly lit area. If the player fails to leave the lit area before the timer runs out, the next event will show the monster disabling a light source. The “In Darkness Delay” input (9) provides a value for the “In Darkness Timer”, covered on page 203. The “Kill Delay” input (10) sends its value to the “Kill Timer”, which is defined on page 203. The “Key Drop Delay” input (11) is the duration between the moment the player collects the flashlight and the moment when the key is dropped in the level that can be specified here.

"Monster Camera" events require several variables to function correctly. These variables can be viewed and edited on the "Event Setup Menu". Saved values are retrieved from the hard drive and mapped to their corresponding local variables. The system saves the values as individual variable types, such as "floats", or an array that could contain a list of similar variable types. To correctly display the values stored in the array, while readying them for the in-game "Monster Camera" system, a "For Each Loop" scans through it while incrementally extracting and mapping each value in order. The blueprint interacts with UE4's built-in save system that saves permanent copies of these variables to the hard drive for later use. Afterwards, these variables are converted to text strings and then displayed on the "Event Setup Menu".

The system can reset the saved variables used by the "Monster Camera". Hardcoded values stored in a "default" variable and array set allows the system to easily swap out the previously saved values with the default ones. The system further combines theses "default" values with several other techniques to reset the values displayed on the "Event Setup Menu".

The "Monster Camera" system requires the values retrieved from the save slot on the hard drive to be transferred from the local variables used by the "Event Setup Menu" to a corresponding global set of variables that are used by the "Monster Camera" system in the playable level. A "Save Game Instance" is used to facilitate the above requirement in that local variables can be loaded into equivalent global variables that can then be used in any level, while "Interloper" is running. The variables stored in the "Save Game Instance" are wiped as soon as the player quits the "Interloper" application. A detailed breakdown of blueprints used to achieve the systems mentioned above is available in Appendix 3 on page 130.

5.3 Level Scripting

Every important narrative event explored above has a technical layer that needs to be discussed to make the functionality of each clear within the broader context.

5.3.1 Office

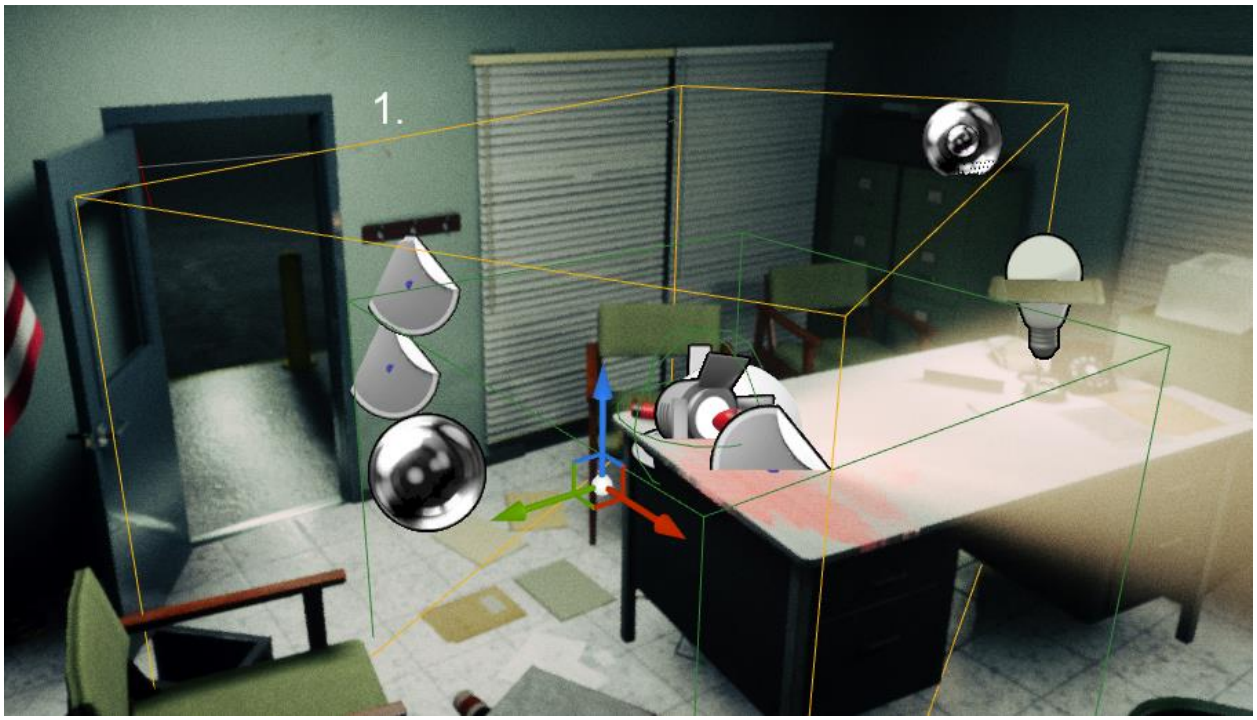


Figure 38. Office environment.

The office is a crucial area within the narrative arc (figure 38). The stakes of the game are established here, and it is the area where the player triggers the activation of the "Monster Camera" event system. In figure 38, the moment the player turns the corner, a trigger box (1) plays a sound that marks the surprise discovery of the scene. A technical breakdown of how the system plays this discovery sound can be view in Appendix 4 on page 140.

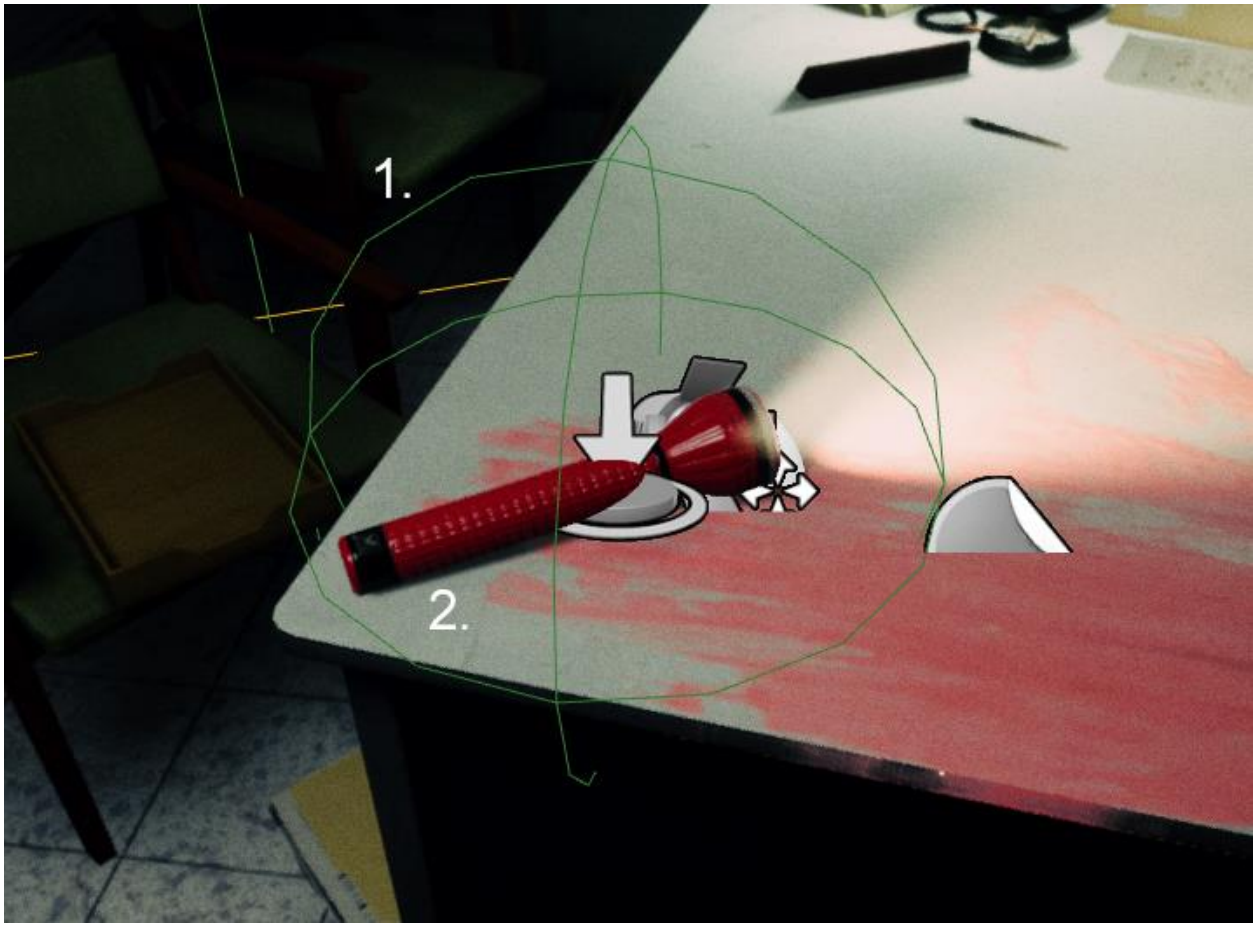


Figure 39. Flashlight pickup.

Figure 39 shows the flashlight (2) located on the desk in the office, which makes use of a collision sphere (1). This large collision sphere was used to make it easy for the system to detect that the flashlight on the desk is an interactive object. Initially, the system had difficulty picking up that a flashlight was on the table, due to its size.

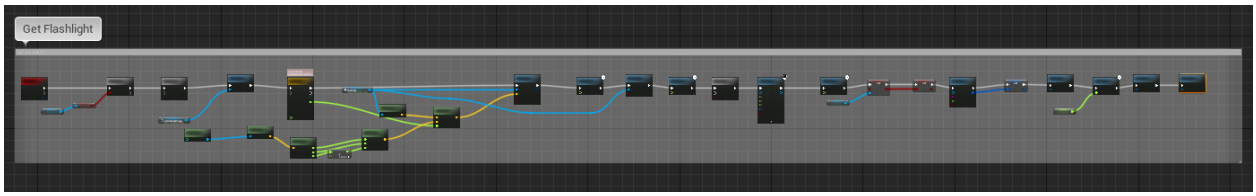


Figure 40. Blueprint that allows the player to collect the flashlight.

A level blueprint was used to enable the player to retrieve the flashlight (Figure 40).

The blueprint runs through several actions when the player's avatar overlaps the collision sphere that activates the "interaction" icon that initiates the retrieval animation when the corresponding key is pressed. The activation of the retrieval animation informs the system of the player's intention and immediately disables the collision sphere, which also disables the "interaction" icon for possible future key presses. It also samples the position of the player so that when the player 'absorbs' the flashlight mesh, it moves it in the right direction. The system hides the flashlight mesh as soon as the player absorbs it, which signifies the successful retrieval of the said flashlight. The beam of light that illuminates the text on the wall is also disabled when the player retrieves the flashlight, while a flashlight switch-off sound plays concurrently. The flashlight retrieval blueprint also triggers the first monster camera event, the randomised placement of the gate key in the playable area, and the first suspense curve incrementation. An in-depth breakdown of the flashlight retrieval process and how this action activates the "Monster Camera" system is detail under Appendix 5 on page 141.

5.3.2 Car Spawner

A procedural car generation and placement system were also created to make the placement of the cars in the parking structure different each time a player plays the game.



Figure 41. Procedural car.

Figure 41 shows the appearance of one of the procedural cars. The car mesh uses pink as the default colour when placing it. This default colour will dynamically change to one of the predetermined colours, each time the system spawns a car in the level.

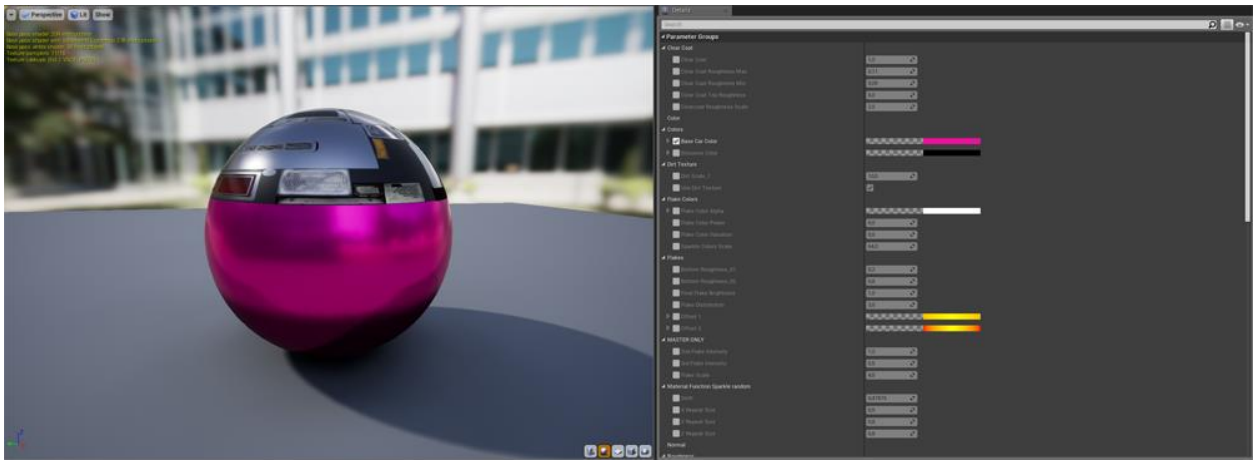


Figure 42. Procedural car material instance.

Figure 42 shows the inner workings of the material that is applied to this car. It contains various parameters, one of which will be changed dynamically.



Figure 43. Base colour of procedural car.

The “Base Car Color” parameter, as shown in figure 43, is the colour parameter that will be changed dynamically.



Figure 44. Procedural car blueprint hierarchy.

Figure 44 depicts the hierarchy of the car’s blueprint. The “Default Scene Root” component (1) is the container that holds the remaining items. The “Static Mesh” component (2) contains the car body mesh. This mesh can be changed depending on the car model chosen. The “WheelBL”, “WheelFR”, “WheelBR”, and the “WheelFL” components (3) are unique blueprint components that dynamically change the appearance of the wheel, each time the car spawns.



Figure 45. Procedural wheel.

This wheel blueprint is depicted in Figure 45. It is a standalone blueprint that can swap out the tyre and hubcap meshes to one of the similar-looking library meshes.



Figure 47. Procedural wheel blueprint hierarchy.

Taking a closer look at the wheel blueprint's hierarchy, as shown in figure 47, it is clear that its construction is relatively simple. It also has a "Default Scene Root" component (1) that holds everything else. Embedded within this root component are a "Wheel" component that contains a mesh of the tyre, and a "Rim" component (3), which includes a mesh of the hub cap. Both these meshes can be swapped out.



Figure 48. Procedural wheel rim selection.

The “Wheel” blueprint, as shown in figure 48, contains a set of nodes that seems somewhat complicated. The blueprint achieves this by selecting a random number within a predetermined number range, as represented by the number of meshes available, and then selecting the corresponding meshes when it reaches the said number. A detailed explanation of how the blueprint procedurally swaps out the wheel meshes is available in Appendix 6 on page 146.

Further to this, the car blueprint executes two additional pieces of code.

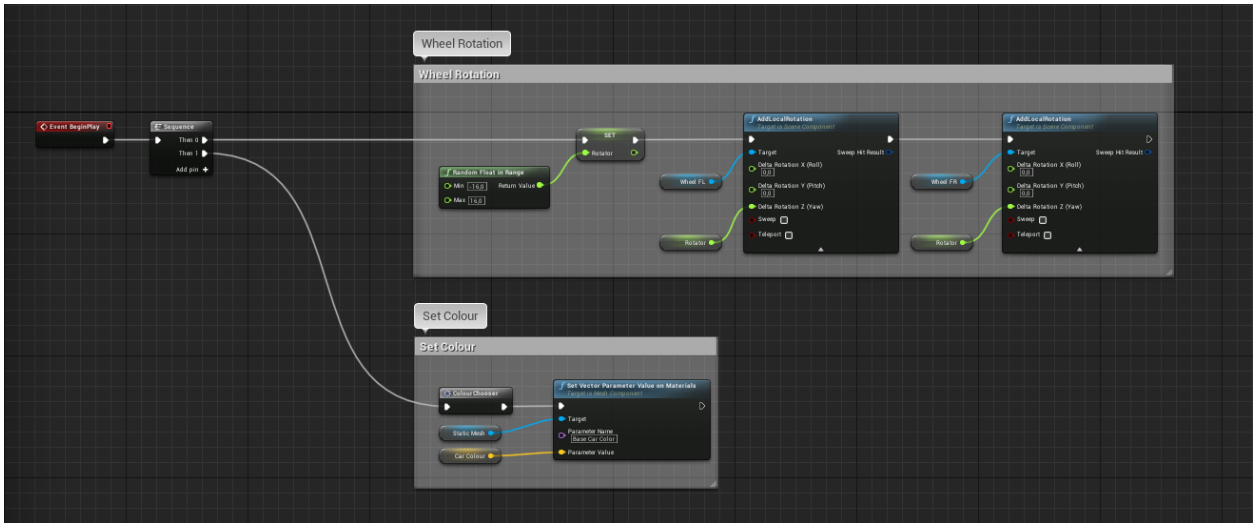


Figure 49. Procedural car wheel rotation and random colour selection system.

These two pieces of code, namely “Wheel Rotation” and “Set Colour”, are depicted in figure 49. As soon as a new car spawns, the “Wheel Rotation” script slightly rotates the front wheels to provide the player with the illusion that the car turned during its parking manoeuvre. The “Set Colour” script randomly selects a colour value for the car from a predetermined palette the moment that car spawns. A detailed breakdown of the “Wheel Rotation” and “Set Colour” scripts is available under Appendix 7 on page 148.

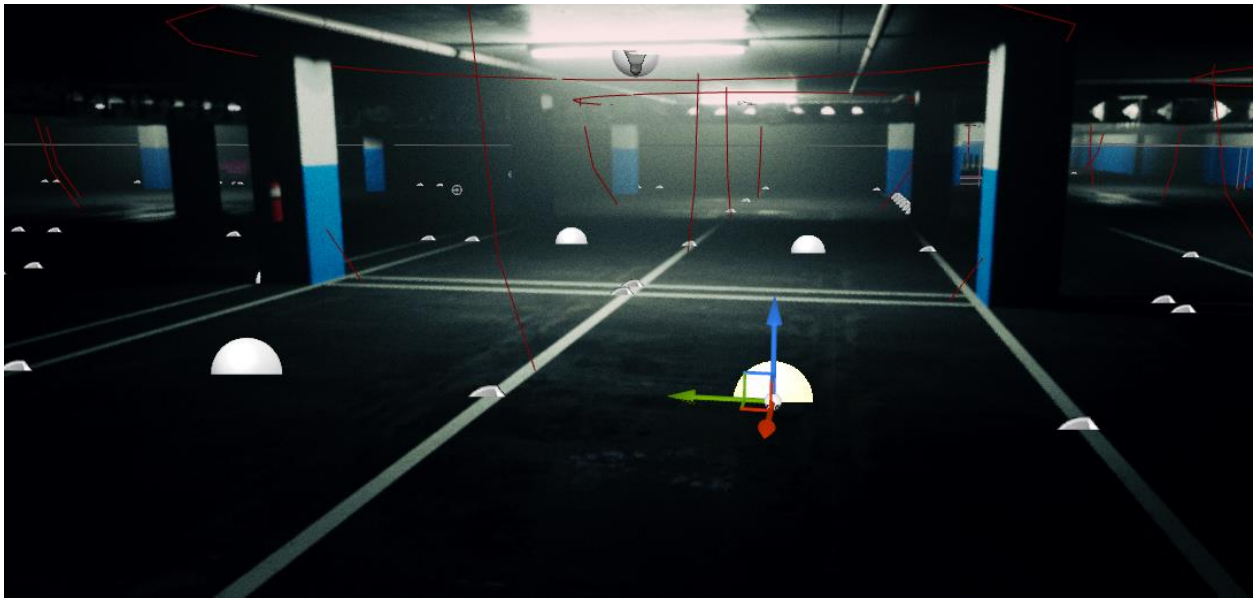


Figure 50. Procedural car placement nodes.

Placing the various procedural cars within the level requires additional coding to determine how many cars should be spawned and where they should be located within the level. Figure 50 shows the empty actor components that are used to place procedural car actors.



Figure 51. Procedural car placement node rotation.

As shown in figure 51, the "Z" rotation is also set to "90" degrees, so that the cars are correctly placed within each bay and not across the white lines.

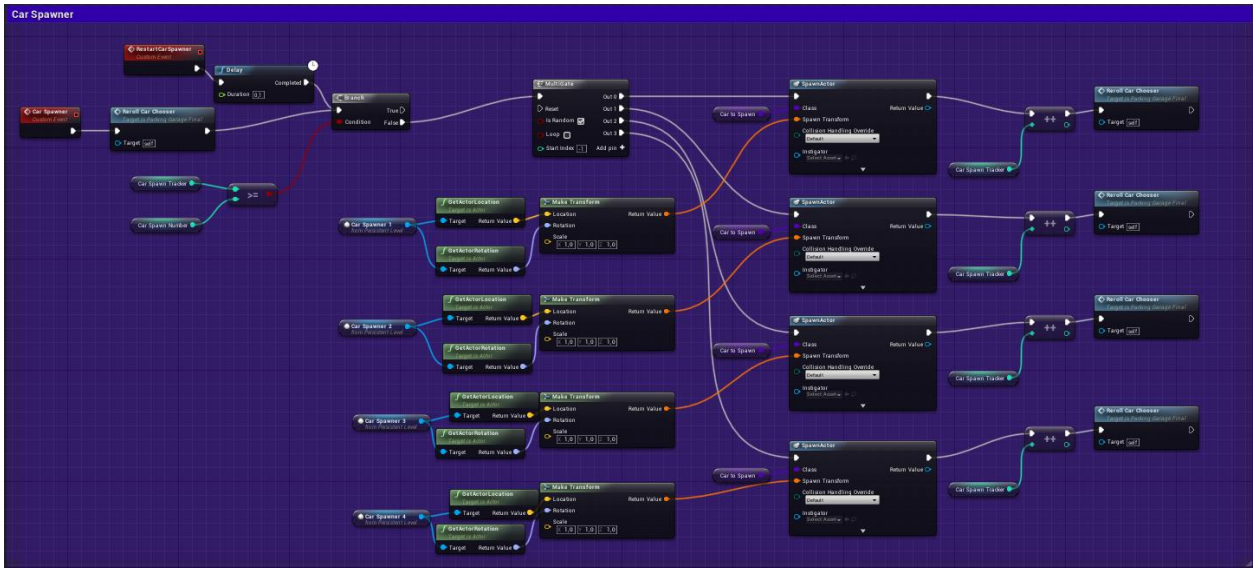


Figure 52. Procedural car placement blueprint.

Figure 52 shows a demonstration blueprint script that was created to place four cars. The "Car Spawner" blueprint makes use of the pre-placed empty actor components as spawn points. A predetermined maximum spawn value determines how many cars to allocate to spawn points in the parking area. The system then takes this value, runs through the available spawn points and then places vehicles on these randomly picked points. A detailed breakdown of how this blueprint procedurally places vehicles in the level is available in Appendix 8 on page 151.

5.4 Monster Camera Spawn System

The "Monster Camera" spawn system is quite robust as it can simultaneously track several elements that are happening in the level at once while spawning the correct "Monster Camera" event as soon as the suspense curve triggers one.

The "Monster Camera" event requires several variables to function correctly. The most significant variables are the time durations that determines when "Monster Camera" events spawn. The system saves these time duration values in an array that gets extracted in

the order required by the spawning system. The “Event Launcher” function is used to launch the correct “Monster Camera” event that makes use of an index value to keep track of the time duration values kept in the “Monster Cam Event Array”. The “Spawn AI” blueprint spawns the correct “Monster Camera” event based on several tests that it performs internally. The blueprint makes use of the “Monster Camera” event durations stored within the array explored above. The current index value is compared with the “Last Index” value to track the progress of the array.

It also activates and deactivates a “skylight” component within the level itself. This “skylight” component, when active, illuminates the entire level with a uniform ambient fill light that does not cast any shadows. This light activates in conjunction with the post-processing effect that is enabled when a “Monster Camera” event is active. This light is a quick-and-dirty technique to make it easier for the antagonist to see in the dark.

The system continually checks whether it has reached the last index in the array or not. This test is used to track the progress of the entire suspense curve. As soon as the system hits the last index, a special “Monster Camera” event spawns that kills the player.

When the player is still within the confines of the suspense curve, as represented in the array, the system uses a Boolean value to check whether the initial “Monster Camera” event needs to be activated or not. This initial event always starts from the same location and shows the player, through the opposing perspective, where the antagonist enters into the playable area.

As soon as this initial “Monster Camera” event activates, the system gets primed to initiate the following “Monster Camera” event type. The “Player Location – Monster Cam Active” function tracks the location of the player for use by the spawning system, while a Boolean test checks to see if the initial timer has expired. This initial timer provides the

antagonist with some time to wander around, making it appear as if the antagonist is searching for the player.

When this initial time lapses the system automatically activates the part of the blueprint that checks to see if the player is near any light sources. The variable saved by the "Player Location – Monster Cam Active" function determines whether to activate the "In Light" or "In Darkness" timers or not. These independent timers use CPU resources sparingly, unlike the "Event Tick" nodes.

The system spawns "Monster Camera" events that randomly roam about until the "In Light" timer runs out. When the "In Light" timer runs out, the "Restart Light Test" event launches. The "Restart Light Test" event checks what light sources are active and picks out a random one for the antagonist to deactivate.

When the player remains in a dimly lit area for a long time, the "In Darkness" timer starts up. Another Boolean test then checks to see when it runs out. A "Monster Camera" event that locks on and circles around the player's avatar gets activated when the "In Darkness" timer is still within its countdown. The "Kill Timer" initiates as soon as the "In Darkness" timer runs out and the player does not move to the protective confines of a light source. The "Kill Timer" similarly only counts down as long as the player remains in darkness. The system spawns a "Monster Camera" event that shows the antagonist approaching the player during this countdown. The system spawns a player-killing event as soon as the countdown ends. A detailed breakdown of the blueprints used to achieve the above is available in Appendix 9 on page 153.

5.5 Monster Camera Events

The blueprint responsible for spawning the “Monster Camera” events reveals that there are five camera event types that behave differently. “Monster Camera” events consist of two components that work together to create the final effect. The first component is a “Character Pawn” actor. A “Character Pawn” can be classified as an AI entity with scripted logic that allows it to move around the level while making use of the navigation mesh. The “Monster Camera” macros are self-enclosed pieces of logic that can spawn the “Character Pawn” entities and define the destination of their movement. These macros can be summoned at any point in the main blueprint code, making them extremely versatile. The “Character Pawn” entities will be explored first, as they are simple with minor differences between their variations. The “Monster Camera” macros will be explored next. All of the “Monster Camera” macros are similar.

One of these macros will be explored in-depth, while the differences of the other macros will be pointed out afterwards.

5.5.1 Monster Avatars (Character Pawns)

The monster avatars are “Character Pawns” that consists of a collision capsule that allows them to navigate the level and prevent them from falling through the floor. These avatars also contain a camera that allows the perspective to shift between the player and the monster. Three separate pawn entities were created with slightly different properties to allow them to either look at the direction of the player or walk faster. Several parameters are needed for the monster avatars to accelerate, move, turn and stop smoothly. The “Monster Cam Seek”, “Monster Cam Look” and “Monster Cam Kill” avatars are very similar to each other, apart from tiny differences that the paper will explore below.

5.5.1.1 *"Monster Cam Seek"*

The "Monster Cam Seek" avatar enables the monster to look straight ahead while walking through the level. It is used in conjunction with the "RandomMonsterMovement_Initial" and the "RandomMonsterMovement_Next" macros.

The paper covers the avatar's appearance, the components used to enable its functionality, and the various settings used to move it around in Appendix 10 on page 167.

5.5.1.2 *"Monster Cam Look"*

The "Monster Cam Look" avatar can look at the player while moving through the environment. It is meant to be used with the "MonsterLookAt_Next", "MonsterApproach_Next" and "MonsterDisableLight" macros. It is nearly identical to the "Monster Cam Seek" avatar. The paper explores a detailed breakdown of the "Monster Cam Look" avatar in Appendix 11 on page 173.

5.5.1.3 *"Monster Cam Kill"*

The "Monster Cam Kill" avatar is also nearly identical to the two avatars above. The camera component is also able to lock in on the location of the player's avatar. The only difference is the speed at which this avatar moves and the amount of "head bob" affecting the camera. The paper explores the settings used by "Monster Cam Kill" avatar in Appendix 12 on page 174.

5.5.1.4 Head Bob

The game makes use of a "head bob" effect to jostle the camera depending on how fast an avatar is moving. It is intended to simulate a character's head movement while walking. The head bob used in UE4 is a universal effect that affects all cameras equally. It,

therefore, needs to be enabled or disabled depending on the camera that is currently active. The “head bob” effect is disabled when the player is in control and enabled whenever the antagonist is moving through the level. The system uses the head-bob in conjunction with a post-processing effect to distinguish between the protagonist and the antagonist. The blueprint makes use of the protagonist’s velocity to modulate the intensity of the camera’s shake speed. The camera shake effect is a component placed in the scene that, when active, has a maximum radius that affects all cameras present within this radius. The set radius encompasses the whole level. The paper explores this blueprint in Appendix 13 on page 176.

5.5.2 Monster Camera Macros

The “Monster Camera” system makes use of four core macros to spawn the correct avatars. These modular macros work the same way, for the most part, bar some small changes between them.

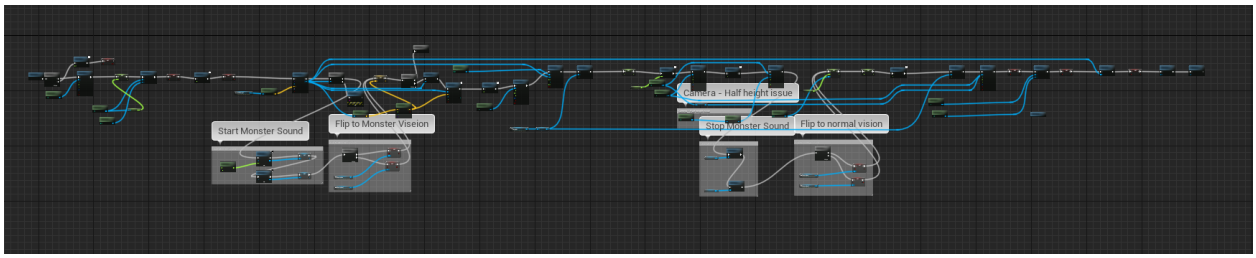


Figure 53. “Monster Camera” macro.

Figure 53 shows the appearance of one of these macros. Only the “RandomMonsterMovement_Initial” macro will be covered in-depth, while the differences will be covered for the rest. These breakdowns are explored in Appendix 14 through 17 and will cover several essential aspects of the system. These aspects include how the system spawns the antagonist AI, how the player loses control and how the player’s camera

perspective fades to the viewpoint of the antagonist. These sections also cover the activation and deactivation of sounds and visual effects originating from the antagonist.

Four main “macro” types control the various “Monster Camera” avatar spawns. The “Initial” macro launches the “Monster Camera” from a fixed location when the player first activates the system; the paper explores these details in Appendix 14 on page 177. The “_Next” macros make use of the antagonist’s last known location after switching back to the player so that the next AI spawn happens in the same vicinity. Making use of the previously known location provides the AI with some believable continuity. The paper expands on the “_Next” macros in Appendix 15 on page 188. The “MonsterKillPlayer” macro spawns the last “Monster Camera” event that is responsible for delivering the killing blow and the player’s fail state. The paper analyses the “MonsterKillPlayer” macro in Appendix 16 on page 194. The “Monster Disable Light” macro considers the location of the player when spawning an event. When the player spends too much time in a lit area, this macro spawns a “Monster Camera” event that disables a randomly selected light source in the playable area. Fewer light sources make it harder for the player to avoid the antagonist’s killing blow. The paper unpacks the “Monster Disable Light” macro in Appendix 17 on page 200.

5.5.3 Timers

The “Monster Camera” system makes use of several timers that determine when each one of the various “Monster Camera” macros is launched. Timers execute custom functions every second. These functions contain code that increments the timer value until a predetermined endpoint is reached. The timers activate and deactivate independently and only consume processing power when enabled, unlike an “Event Tick” process. The system contains four timers, namely the “Initial Timer”, the “In Light Timer”, the “In Darkness Timer”, and the “Kill Timer”.

The “Initial Timer” is activated when the player picks up the flashlight. This single-use timer counts the duration between the initial pick-up action and when the first “Monster Camera” event spawns. The system spawns “Monster Camera” events that merely roam around looking for the player, while this timer is active.

The “In Light Timer” activates whenever the player is near a light source. The system will trigger a “Monster Camera” event that deactivates a random light source when the player remains in an illuminated area after the “In Light Timer” expires.

Similar to the above timer, the “In Darkness Timer” checks to see how long the player stays within a dimly lit area. How the “In Darkness Timer” differs from the “In Light Timer” is what happens afterwards. When the “In Darkness Timer” runs out, the “Kill Timer” activates. Unless the player quickly returns to a lit area, the “Monster Camera” event associated with killing the player’s avatar activates. The paper analyses the timer blueprint in Appendix 18 on page 201.

5.6 Player Avatar

The player's avatar is a "Character Pawn" that represents the protagonist, which the player can control.

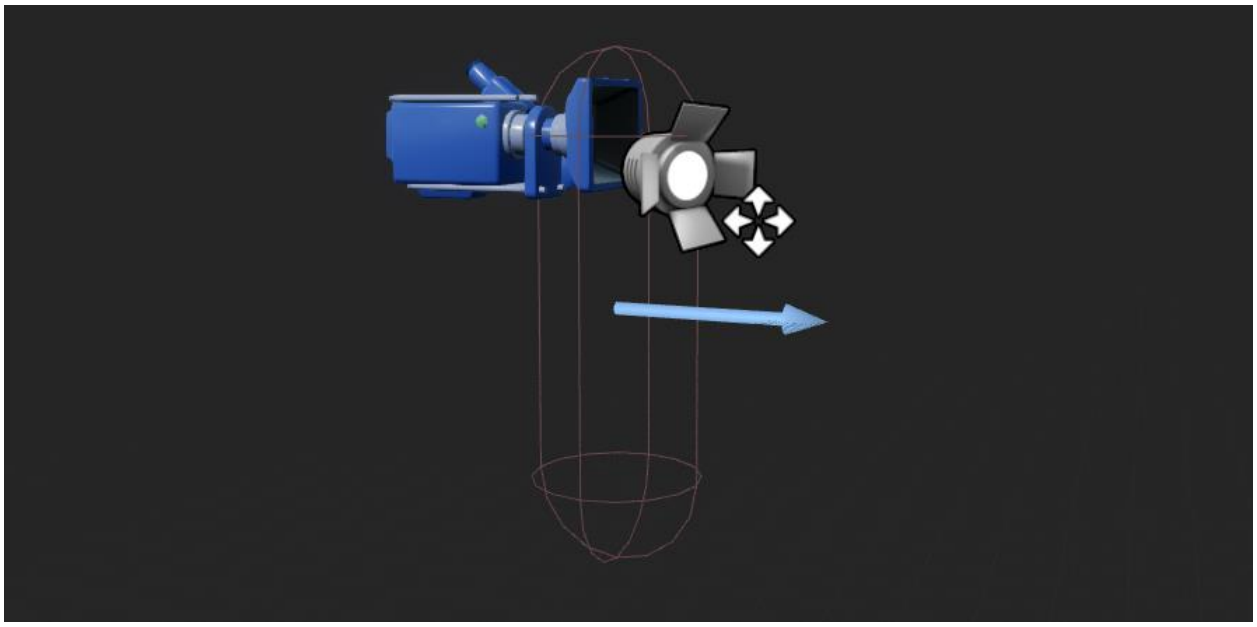


Figure 54. Player "Character Pawn".

Figure 54 shows that the physical construction of the player's avatar is straightforward.

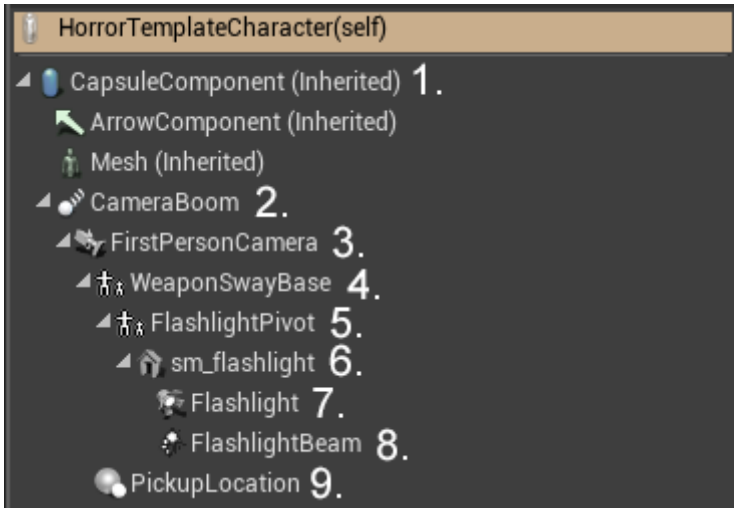


Figure 55. Player "Character Pawn" hierarchy.

The hierarchy of the player's avatar depicts the various elements that are embedded together within the player's avatar (figure 55). The "Capsule Component" (1) is a physically enabled capsule that allows the player to move through the level without passing through other physically enabled meshes such as walls, floors, pillars, and doors. The "Camera Boom" component (2), which is a "Spring Arm" component, refers to the camera booms used in cinematography and has a similar function in that it controls the movement of the "First Person Camera" component (3). The "Weapon Sway Base" component is a pivot that was originally used to house a weapon model for first-person shooter-style games. For "*Interloper*" it will be used to house another pivot, which is aptly called the "Flashlight Pivot". This pivot stores the mesh of the flashlight (6), its light source (7), and a particle system that represents the beam of the flashlight (8). This assembly of items will be raised whenever the player presses the button that enables the flashlight. The "Pickup Location" component (9) was originally used as a destination location for a key pickup animation that was meant to work in the same way as the flashlight collection animation. Unfortunately, the key collection animation never worked as intended, so it was left out of the current build of "*Interloper*".

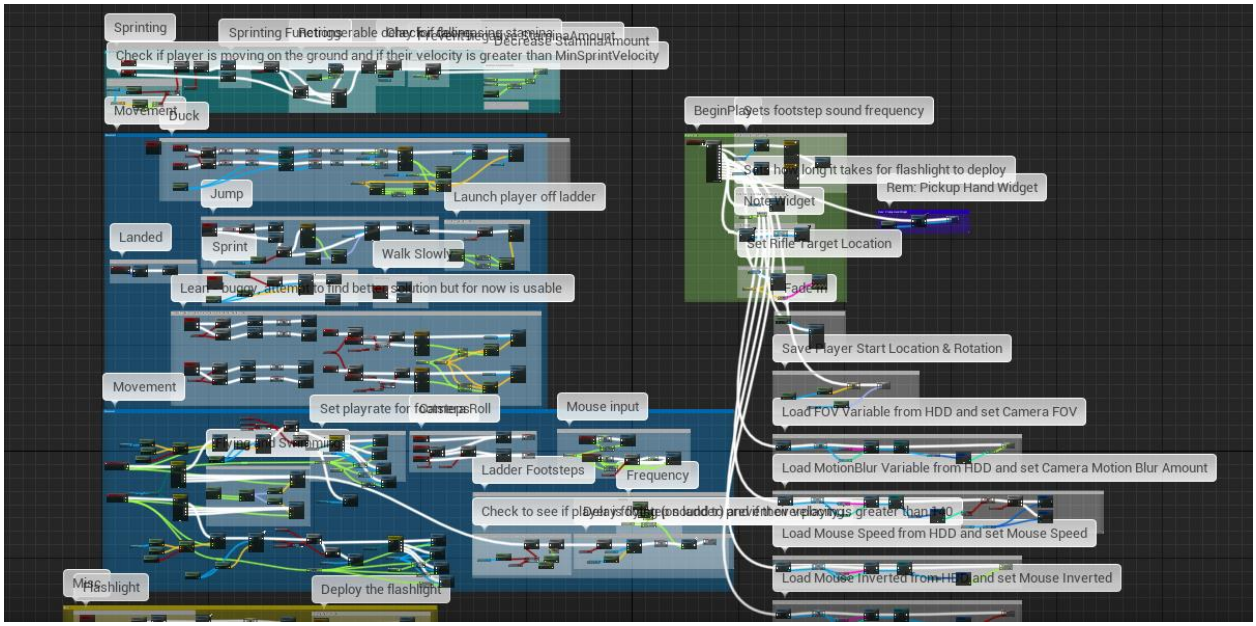


Figure 56. Player controller blueprint.

Figure 56 shows the complexity of the character controller. The controller was purchased on the Epic Games Marketplace and is heavily modified. It provides the necessary functionality for a smooth play experience. Most of the nodes within this controller does not fall within the scope of this paper and will not be covered. There are some code snippets embedded within this controller that does link to the core mechanics of "*Interloper*", however.

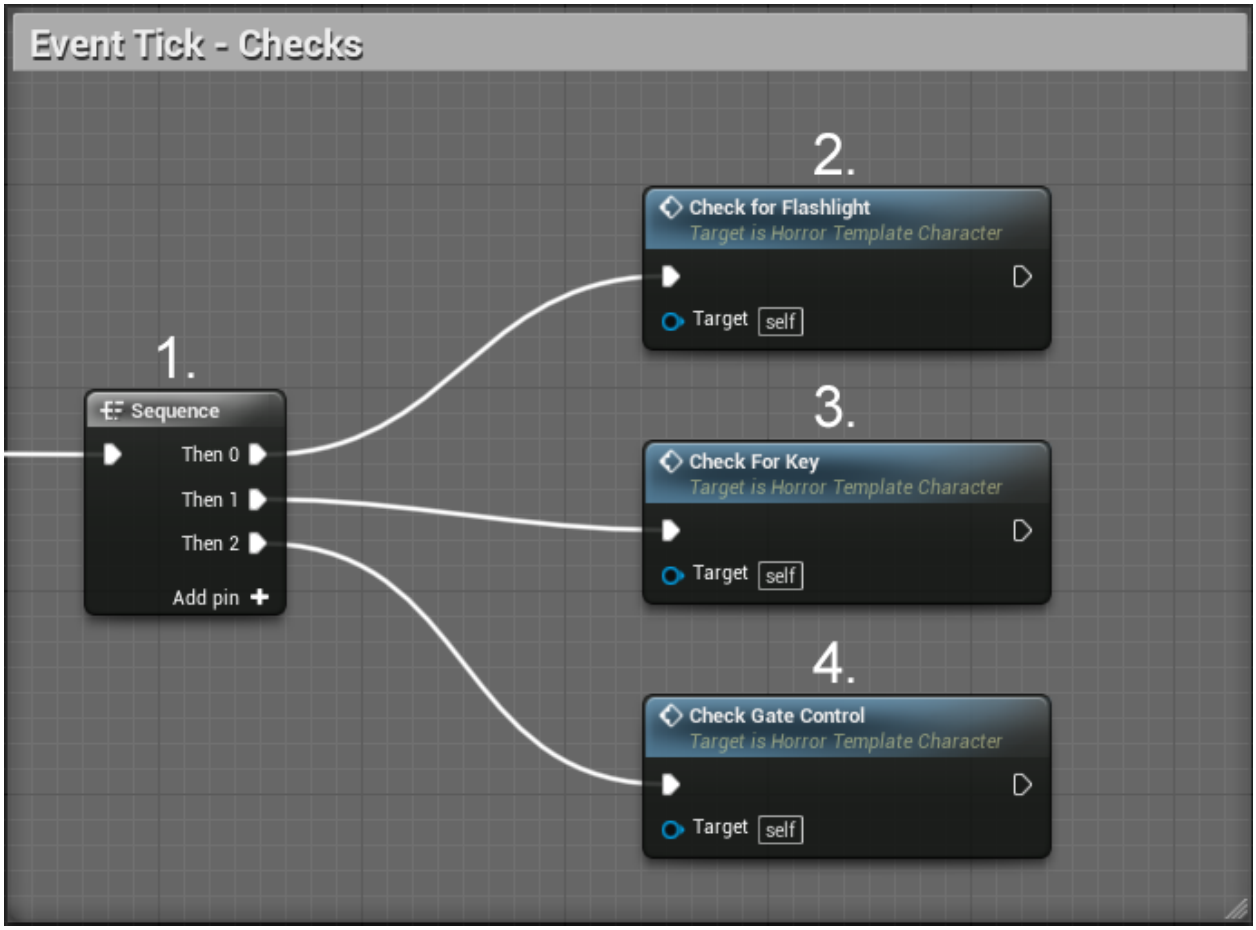


Figure 57. Player controller event checker functions.

Figure 57 shows three custom "check" functions (2, 3, 4) that run sequentially (1). Please take note that a tick operation executes these three functions every frame. All three functions are duplicates of each other, barring some of the variables used in each. These functions incorporate a conditional operation that determines whether the player can interact with the item before displaying the "use" icon.

The "Check for Flashlight" function, for example, checks to see whether the player has collected the flashlight before displaying the "use" icon.

The "Check Gate Control" function displays a use icon when the player is close to the control box. The message that displays after the player presses the "use" key changes based on the availability of the key.

The “Check for Key” function is straight forward in that it only displays a “use” icon when the player looks at the key mesh that is procedurally placed in the level. When the key is visible, and the player presses the “Use” key, the key is immediately absorbed and enables the player to activate the gate control.

The paper examines the functions mentioned above in Appendix 19 on page 205.

5.6.1 Breathing System

A breathing system that works alongside the other blueprints explored above tracks the progress of the suspense curve and plays an increasingly frantic breathing sound as the suspense heightens. It uses the resultant values to map the correct breathing sound and low pass filter value to the said breathing sound and the ambient audio found in the level, resulting in an increasingly disembodied audio mix. The paper explores how the blueprint achieves the abovementioned outcome in Appendix 20 on page 209.

6. Design Improvements

"*Interloper*" was developed with the primary goal of using disempowerment and reflection to cause suspense within a non-linear open-ended environment, while steering clear of scripted funnels. After several iterations and level layout changes, the final layout was chosen to combine some aspects of the funnel with an expansive section that can be explored freely. This method is in keeping with the way *Alan Wake,* Microsoft Studios (2010), used sizeable arenas for its enemy encounters. The main difference between *Alan Wake* and "*Interloper*" is the fact that the player will never see what the antagonist looks like, so will never face it directly. Players merely see through the eyes of the antagonist for brief moments, while losing total control. Several major stumbling blocks were faced during the development of "*Interloper*". One of these issues was finding the best way to show the actions of the antagonist. Initially, several canned animations were developed and placed randomly in the level. There were limitations with these animations, however. The "Monster Camera" events were replaced with AI-controlled characters, which can freely move around when permitted by the navigation mesh, to make their actions unique each time. There is one thing that can be added to "*Interloper*" to make the ending more frightening. Adding a three-dimensional representation of the antagonist pouncing at the player could make the impending dread even more suspenseful for players. This addition could defeat the purpose of this research, however. Making the threat known could make the suspense rely on this concluding event only and could detract from measuring the effectiveness of the "Monster Camera" events by themselves. By only using the "Monster Camera" events, it should be possible to test the efficacy of using disempowerment and reflection to evoke suspense. Other features that can still be added to the tool include the following:

- A dedicated button to create control groups when a control group sample is necessary during an experiment. Currently, the user needs to reset the “Monster Camera” event list and save this empty list so that no event spawns during a play session.
- Additional controls to manipulation the in-game colours. These controls can include a method that controls the post-processing colour schemes for both the protagonist and the antagonist. A control that can adjust specific in-level colours, such as the blue colour scheme that is used in the parking area.
- A menu that visualises the suspense curve after inserting the required event durations.
- Additional controls that can disallow certain “Monster Camera” events from playing.
- A dedicated control that enables a “jump scare” moment when the last event activates.
- A technique for the players to track the gate key down could be a welcome addition. A heartbeat sound that increases based on the proximity to the key could provide some additional situational awareness.
- Make the monster sound more agitated every time a new event spawns, by integrating a system that works in the same way as the breathing system.

Fluid hypotheses on how best to use this tool could hinge on three properties. These three properties include the amount of time before the first “Monster Camera” event spawns, the duration between each spawn and the duration of each self-contained “Monster Camera” event. The actions primarily control the event that gets chosen and the location of the player in the level, so it should not have a significant bearing on the system unless the individual location-specific timers are set to work in conjunction with the event spawns. A possible configuration that could provide researchers with the best results is by gradually decreasing the duration between each “Monster Camera” spawn. The duration between each spawn has a direct impact on the appearance of the curve. Reducing the time between each subsequent

spawn would ramp up the suspense curve exponentially. To confuse players, the duration of the last event, which is the killing blow, can be made longer than expected by the player, making for a tense pause.

As mentioned above, not having a "jump scare" at the end could have a detrimental effect on the overall experience. The last event, which is the killing attack, is very similar to the other "Monster Camera" events in that it moves through the level in the same way. The only difference is that it moves faster and moves directly to the protagonist, fading the screen to black and playing an attack sound, when it reaches the player's location.

7. Conclusion

The development of "Interloper" shows that it is possible to develop a game that can generate potentially suspenseful events in an open-ended space, using the search-based procedural content generation technique. Researchers can use this game as a tool to create disempowering events. Testing how the events interrelate and how such dynamics contribute to suspenseful moments could provide researchers with invaluable information regarding how humans perceive suspense.

This game needed to be able to portray a horror experience convincingly. Using physically based materials, I was able to create near photorealistic scenes. These scenes by themselves could potentially immerse the players of "*Interloper*" deep within the atmosphere of this virtual world and potentially cause suspense that way. Adding procedural mechanics that strips the control away from players when they so desperately need to find an escape provides an additional layer of suspense to this experience. Menu systems were added to allow researchers the ability to customise every play session. This system will provide them

with the unique ability to structure several unique experiments. "*Interloper*" rests on the primary suspense causing pillars that are made up of disempowerment and reflection in a seemingly open space. Achieving this was a challenging feat in and of itself. A deep understanding of the game engine was needed and subsequently achieved by completing this research project. One of the core theories that were used to develop the narrative arc in "*Interloper*" was Moritz Lehne and Stefan Koelsch's general model of suspense and tension. It provides the game with both a good and a bad end state. The bad end state is usually more common in "*Interloper*" as it is rather challenging to find a way to escape the parking structure. Fortunately, hope springs eternal for the players of "*Interloper*" as a means to escape is literally at their fingertips in the form of a key lying on the ground somewhere. Players have to find it. Various environmental clues are also scattered around the level, giving players some insight into what happened and what they need to achieve.

Developing this game was challenging. Various novel methods had to be invented for everything to work according to plan. Many problems arose, including but not limited to lighting issues, issues with the AI controllers, the monster not spawning correctly and many more.

There are both "firm" and "loose" factualities that arose during the development of "Interloper". Several firm facts emerged and mainly dealt with procedural technology. The first fact is that procedural technology can be used in an effective way to create relatively suspenseful scenarios in a video game format. The second fact is that procedural technology can be used in conjunction with relatively open play spaces. The last firm fact is that static events that are supposed to be suspenseful can be converted to dynamic events using the above-mentioned procedural technology.

The loose facts that emerged from the development of “Interloper” deal with the effectiveness of the events and environmental elements that are supposed to evoke suspense. The first loose fact is the amount and effectiveness of the “Monster Camera” events, which need to be tested to see what frequency delivers the best results. The second loose fact deals with the use of contrasting colour and lighting that were used in conjunction with the camera events. Whether the inclusion of these events has any real impact is a big question that requires additional research. The last loose fact is the effectiveness of the breathing system. It was added to both increase immersion and as a subtle measuring tool. Players can track their progress on the suspense curve by listening to the speed of the protagonist’s breathing and the intensity of the low-pass effect. Whether this mechanic hampers the overall suspense, also requires more research to be done.

The video game “Interloper” will provide researchers with a strong foundation to assist them with additional research. The stimuli that “Interloper” generates can be customised and combined in various ways and, when tested in conjunction with physiological measurement tools, could deliver compelling results. The results of such tests will assist researchers in finding the answers to several hypotheses regarding disempowerment and reflection when trying to build suspense in an open space without engaging enemies directly.

8. References

Alan Wake. Remedy Entertainment, 2010.

Alien: Isolation. Creative Assembly, 2014.

Call of Cthulhu: Dark Corners of the Earth. Headfirst Productions, 2005.

Dead Space. EA Redwood Shores, 2008.

Dead Space 2. Visceral Games, 2011.

Dead Space 3. Visceral Games, 2013.

Abbot, Porter H. *The Cambridge Introduction to Narrative*. Cambridge University Press, 2001.

Benford, Steve, et al. "Uncomfortable user experience." *Communications of the ACM*, vol. 56 no. 9, Sept. 2013, pp. 66-73.

Brewer, William F., and Edward H. Lichtenstein. “Stories are to entertain: a structural-affect theory of stories.” *Center for the Study of Reading Technical Report*; no. 265, 1982, pp. 14

Cheong, Yun-Gyung. *A Computational Model of Narrative Generation for Suspense*. 2007. Department of Computer Science, North Carolina State University, Raleigh, NC, PhD Dissertation.

Doust, Richard. A. *A Domain-Independent Model of Suspense in Narrative*. 2015. The Open University, PhD Dissertation.

Dresch, Aline, et al. *Design Science Research: a Method for Science and Technology Advancement*. Springer, 2015.

Egenfeldt-Nielsen, Simon, Jonas Heide Smith, and Susana Pajares Tosca. *Understanding Video Games: the Essential Introduction*. Routledge/Taylor & Francis Group, 2013.

Frome, Jonathan, and Aaron Smuts. "Helpless spectators: Generating suspense in videogames and film." *TEXT technology,* vol. 13, 2004, pp. 13-34.

Grimshaw, Mark. "The Audio Uncanny Valley: Sound, Fear and the Horror Game," *Proceedings of the Audio Mostly 4th Conference on Interaction with Sound*, 2009, pp. 21-26.

Hendrikx, Mark, et al. "Procedural Content Generation for Games." *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, Jan. 2013, pp. 1–22., doi:10.1145/2422956.2422957.

Krzywinska, Tanya. "Hands-on horror." *ScreenPlay: cinema/videogames/interfaces.* Wallflower Press, London, 2002. pp. 206-223.

Lehne, Moritz, and Stefan Koelsch. "Toward a General Psychological Model of Tension and Suspense." *Frontiers in Psychology*, vol. 6, Nov. 2015, doi:10.3389/fpsyg.2015.00079.

Newman, James. *Videogames*. Routledge, 2012.

Midnight Garage. Directed by Zhou Yaowu,
Performance by Alex Fong, Yoon So-yi and Gordon Lam,
3 April 2015

P2. Directed by Franck Khalfoun,
Performance by Rachel Nichols and Wes Bentley,
Summit Entertainment (US) and Palisades Tartan (UK), 9 Nov. 2007

Rouse III, Richard. "Match Made in Hell: The inevitable success of the horror genre in video games." *Horror Video Games: Essays on the Fusion of Fear and Play*. McFarland & Company, Oct. 2009, pp. 15–25.

Scream 4. Directed by Wes Craven,
Performance by Neve Campbell, Courteney Cox and David Arquette,
Dimension Films (US), 15 April 2011

Seif El-Nasr, Magy, et al. "Dynamic lighting for tension in games." *Game Studies Journal*, vol. 7, no. 1, 2006.

Takeda, H. et al. (1990) "Modeling Design Processes," *AI Magazine*, vol. 11, no. 4, Winter, 1990, pp. 37-48.

Togelius, Julian, et al. "Search-based procedural content generation: A taxonomy and survey." *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, Sept. 2011, pp. 172–186.

Vaishnavi, Vijay, et al. "Design Science Research in Information Systems*." Association for Information Systems*, 20 Jan. 2004, http://desrist.org/desrist/content/design-science-research-in-information-systems.pdf.

Vorderer, Peter, et al. *Suspense: Conceptualizations, Theoretical Analyses, and Empirical Explorations*. Routledge, 1996.

9. Additional Credits

Third party assets were used in the development of “Interloper”. The authors of these third party assets are listed below:

Arewoz. *Ultimate air ducts modular system 60 elements lowpoly VR / AR / low-poly 3d model*. 2017. CGTrader. https://www.cgtrader.com/3d-models/industrial/other/ultimate-air-ducts-modular-system-60-elements-3-with-animation-l

Bradovic. *Elevator*. 2012. Freesound. https://freesound.org/people/Bradovic/sounds/166698/

Chapman, Pete. *Autoslidingdoors01*. 2013. Freesound. https://freesound.org/people/klangfabrik/sounds/195905/

Dekogon Studios. *Retro Office Environment*. 2015. Unreal Marketplace. https://www.unrealengine.com/marketplace/en-US/product/retro-office-environment

Epic Games. *Showdown VR Demo*. 2014. Epic Games. https://www.unrealengine.com/marketplace/en-US/product/showdown-demo

Facetious Creations. *Customizable Key Pack*. 2017. Unreal Marketplace. https://www.unrealengine.com/marketplace/en-US/product/customizable-key-pack

Firelight.Studio. *Fluorescent Ceiling Light*. 2017. Turbosquid. https://www.turbosquid.com/3d-models/fluorescent-ceiling-light-max/1122170

Flourescent Ceiling Lightby Firelight.Studio

Kelley, Andrew. *Turning Key*. 2018. Freesound. https://freesound.org/people/akelley6/sounds/453050/

Klankbeeld. *Open rolling door 150328_0543 MS*. 2015. Freesound. https://freesound.org/people/klankbeeld/sounds/315943/

Lahoud, Mattias. *Elevator_Ping*. 2019. Freesound. https://freesound.org/people/MATTIX/sounds/459349/

Lazymonk. *New York Street Ambience*. 2014. Freesound. https://freesound.org/people/lazymonk/sounds/214319/

Meiners, Jasper and Isabel Paehr. *Concrete Walls – Japan*. 2016. Unity Asset Store. https://assetstore.unity.com/packages/2d/textures-materials/concrete/concrete-walls-japan-58774

Omnisis. *Gate-Heavy-OpenClose-WAV*. 2016. Freesound. https://freesound.org/people/Omnisis/sounds/336888/

Paulo, João. Metal-grill-003. 2019. 3D Textures. https://3dtextures.me/2019/05/15/metal-grill-003/

Porto, Thiago. *UAA - Belroy Sedan + Liberty Sedan + Taxi*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-belroy-sedan-liberty-sedan-taxi-118665

Porto, Thiago. *UAA - Bertrand Coupe*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-bertrand-coupe-137387 Porto, Thiago. *UAA - Grandstar Coupe + Sedan + Limousine + Hearse*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-grandstar-coupe-sedan-limousine-hearse-119126

Porto, Thiago. *UAA - Melbick Sedan + Coupe + Wagon*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-melbick-sedan-coupe-wagon-118407

Porto, Thiago. *UAA - Razor Coupe + Sedan*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-razor-coupe-sedan-120359

Porto, Thiago. *UAA - Route Sedan and Coupe*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-route-sedan-and-coupe-129944

Porto, Thiago. *UAA - Typhoon Coupe*. 2020. Unity Asset Store. https://assetstore.unity.com/packages/3d/vehicles/land/uaa-typhoon-coupe-126374

Purkiss, Allen. *Aircon Loop*. 2018.

Purkiss, Allen. *Breathing*. 2020.

Purkiss, Allen. *Footstep 1 - 8*. 2020.

Purkiss, Allen. *General Ambience*. 2018.

Purkiss, Allen. *Horror Game Menu Music*. 2020.

Purkiss, Allen. *Monster Breathing Loop*. 2020.

Purkiss, Allen. *Monster Kill 1 Short*. 2020.

Purkiss, Allen. *Monster Kill 2 Short*. 2020.

Roa, Eduardo. *Modular Industrial Pipes Pack 01*. 2017. Unreal Marketplace https://www.unrealengine.com/marketplace/en-US/product/modular-industrial-pipe-pack

Roach, Ian. *Public Hallway Pack*. 2018. Unreal Marketplace.
https://www.unrealengine.com/marketplace/en-US/product/public-hallway-pack

Scream Studio. *Moving Low Frequencies*. 2017. Freesound.
https://freesound.org/people/ScreamStudio/sounds/392698/

Shaw, Jonathan. *Piano, String Glissando, Low, A*. 2016. Freesound.
https://freesound.org/people/InspectorJ/sounds/339671/

Shirk, MacKenzie. *First Person Horror Template*. 2016. Unreal Marketplace.
https://www.unrealengine.com/marketplace/en-US/product/first-person-horror-template

Silverpak. *The ceiling is concrete with seams Texture*. 2018. CGTrader.
https://www.cgtrader.com/3d-models/textures/architectural-textures/the-ceiling-is-concrete-with-seams

Spik.AI. *Elevator Voice*. 2020. https://spik.ai.

SUMFX. *Chameleon Post Process*. 2015. Unreal Marketplace.
https://www.unrealengine.com/marketplace/en-US/product/chameleon

Sychov, Alexander. *Storage House Set*. 2017. Unreal Marketplace.
https://www.unrealengine.com/marketplace/en-US/product/storage-house-set

Vinnerminner. *Metal keys pickup sound 1*. 2018. Freesound.
https://freesound.org/people/vinnerminner/sounds/437600/

10. Appendices

Appendix 1: UE4 Technical Reference

Technical Reference	Description
Actor	It is an object, which contains several adjustable properties unique to it that can be placed inside a level. Actors can also contain internal blueprints that can dictate custom logic.
Alpha input	It is a linear interpolation node input that controls the transition from the value plugged into the “A” input to the value plugged into the “B” input.
Arrays	It is a variable that contains a list of embedded variables that are of the same type.
Audio cue actor	It contains logic that determines how embedded sounds clips play.
Avatar	It is the digital representation of a controlled character within a game level. It can be controlled by the player or an artificial intelligence algorithm.

Blueprint	It is known as a “script” or “scripting language” that is native to UE4. There are various blueprint types, from actor or object blueprints, level blueprints, UI widget blueprints and more.
Blueprint nodes	Scripting in UE4 happens when several nodes are strung together to form a logical sequence. This sequence, when executed, performs the game logic determined by the developer.
Boolean	It is a variable that consists of only two possible states, true and false
Camera	It is an actor that virtually simulates the characteristics of cameras that are used in cinematography. It can be controlled by the player or used for cinematic sequences.
Character	It is an entity that represents a living person or organism within a virtual world. It is similar to an avatar in that it can be controlled by the player or an artificial intelligence algorithm.
Character controller	It is an interface embedded in character pawns that allow the player to control them.
Collision box	It is an actor in the shape of a box that can be used to check for collisions with characters, line traces or other actors in the level. The size of this box can be manipulated to suit the requirements of the developer.

Collision spheres	It is similar to the collision box in that it checks for collisions with characters, line traces and any other actor specified in the level. The size and shape of the collision sphere can be adjusted to suit the requirements of the developer.
Default scene root - blueprint	It is the root of a hierarchy found in both the pawn and actor blueprint templates. It acts as the central pivot that dictates how actors attached to it moves and scales within the level.
Emissive value	It is a feature that is found on most materials and can be manipulated in the material editor. It prescribes how much a predetermined surface of a material glows. It does not emit light merely acts as the visual representation of a light-emitting glow. Emissive surface effects work well on glowing instruments and light bulbs.
Event begin play	It is an event that is commonly found in most blueprint templates, albeit with some variations of it. It executes all of the succeeding nodes once.
Event graph	It is the primary blueprint canvas that is used to construct node graph scripts.

Event tick	It is similar to the “event begin play” node in that is exists in most blueprint templates. It continuously executes all of the succeeding nodes until the level is unloaded. The rate at which this node executes is linked to the frame rate.
Float	It is a variable type that contains a numeric decimal value. It is usually more accurate to use a float value when doing calculations that contain fractions.
Flow control	It provides blueprints with the ability to choose what nodes or self-contained scripts to launches using conditional testing.
Framerate	It is the rate at which the game engine and a computer's graphics card produces frames that represent the actions taken by the player in a virtual game world.
Function	It is a simple self-contained script that can be created within any blueprint. It allows the blueprint to execute this script numerous times within the containing blueprint. Any changes and additions to the function will cascade down to the instances used within the blueprint.

Game instance	A game instance is a universal blueprint that can be used across multiple levels in a game. There exists only one of these and are usually used to store variables and scripts that can be used independently of the level that is currently loaded.
Graphics cards	It is a piece of hardware that is installed in most modern PCs and laptops. It is designed to calculate complex three-dimensional scenes at interactive speeds.
Head bob	It is the simulated head movement of in-game characters with a first-person perspective of the level.
Input events	It is an event that uses the input key designated to it to execute any succeeding nodes.
Integer	It is a variable type that contains a numeric value without decimals. It is less accurate than the float variables, but is useful when decimal values need to be truncated.
Interpolates	It is used when a smooth transition is required between a start point and an endpoint. Interpolation is widely used including variable interpolation, texture interpolation and even in-game object interpolation.

Level Blueprint	It is similar to the game instance in that it is a universal blueprint that can contain several other blueprints and scripts. If an actor blueprint is a pawn on a chessboard, then the level blueprint is the chessboard. Unlike game instances, level blueprints are unable to communicate with other level blueprints.
Lighting	Game engines, such as UE4, can simulate real-word lighting in digital spaces. These light sources are usually modelled to mimic their real-world counters in various ways.
Line trace	It is an operation that shoots an invisible line between two points. The line trace is usually instantaneous and provides the instigator important information about the in-game object that it hits.
Linear interpolation	It is an extension of the “interpolation” concept. It calculates the positions between two specified points, so that movement between these two points can occur smoothly.
Macro	It is a self-contained script that operates in nearly the same way as functions. The main difference is in the complexity of the scripts contained within a macro relative to the scripts contained within functions. It is possible to use several nodes in macros that are forbidden in functions. The “Delay” node is an example.

Master materials	Master materials are constructed using nodes similar to the nodes found within blueprints. These nodes can be used to define a wide variety of surfaces effects. Parameter nodes act as interfaces that blueprint scripts external to these master materials can manipulate.
Material instance	Material instances are derived from master materials. They save on video memory by only needing to reference the master material once even when applying several variations of this material instance to different meshes. Parameters that were set up in master materials are accessible in these instances and can be manipulated by other blueprint scripts.
Navigation mesh	A navigation mesh is calculated within a “nav mesh bounds” volume that is placed in a level. It calculates the navigation mesh using all of the traversable surfaces within the volume. The navigation mesh allows AI characters to see where they can travel and how they can get to their intended targets.
Object	Objects are the same as actors. They can be placed within a level and can have blueprint scripts associated with them.

Off-line light renders	Game engines are able to calculated complex lighting effects using an offline renderer to speed up the performance of games. These lighting effects are then stored within unique textures that are applied to the static meshes placed in a level.
Pawn	A pawn is the same as a character, and the terms can be used interchangeably. These actors can sometimes be referred to as character pawns. Both the player and AI algorithms can control these pawns.
Physically based materials	It is the material shading standard used by UE4. It makes use of additional texture maps to simulate the roughness or metallic characteristics of real-world materials.
Physically enabled	Physically enabled objects or actors make use of the built-in physics engine to calculate the movements of objects as they collide with each other and the world. The physics engine can simulate real-world physical properties such as gravity, friction, air drag and more.
Point light	A point light is one of the simulated light types that can be placed in a level. This particular type can be used to simulate omnidirectional light sources such as bulbs or fluorescent tubes. Each light source can cast shadows.

Post screen effect	It is a unique dynamic material effect that is used in conjunction with the camera actor by compositing the effect over the player's perspective.
Save game object	It is an object that contains all of the variables that can be filled and then saved into a file onto the player's hard drive.
Spawn	It is an operation that summons a particular actor, character or effect to a specified location within the level that is currently loaded.
Spring arm	It is used in conjunction with the camera actor that is placed within the hierarchy of a character pawn. The camera actor is usually attached to the spring arm component, which allows it to move in various ways relative to its character pawn.
Static mesh	It contains a three-dimensional model file that was uploaded to the engine. These model files represent the various in-game objects that are placed within a level.

Temporal processing	The volumetric effects available in UE4 are expensive to render. Temporal processing can be applied to these effects to speed up these effects. Volumetric effects are calculated every other frame. Temporal processing averages between these frames, providing these volumetric effects with relatively smooth motion.
Textures	Textures are the bitmap representations of surfaces applied to static meshes within an in-game level. These bitmaps, which are two-dimensional image files, are imported into UE4 and used in conjunction with materials to create unique surface effects.
Timeline	A timeline or timeline node provides a unique way to animated within blueprint scripts. Keyframes can be placed on a two-dimensional graph that represents a change in value over time.
Unreal units	These are the units of measurement used within UE4. One unreal unit equals one centimetre.
Variable	It is a container that blueprints use to store values. Various types can be stored, including floats, integers, Boolean values, arrays, objects, text string, and many more.

Vector	It is a variable type that contains vector values, which are location values in three-dimensional space. Vectors consist of an “X”, “Y” and “Z” value, which indicate the three axes in space.
Vector length	It is the resulting float value when the distance between two vector values are calculated.

Appendix 2: Blueprint Node Reference

Blueprint Node	Description
? Is Valid	Checks whether the targeted element exists and is not scheduled to be destroyed.
+	Adds two values together, which can include floats or integers.
+ +	Increments an integer value by one.
< =	Checks to see if value A is smaller than or equal to value B. The output contains a boolean value.
> =	Checks to see if value A is equal to value B. The output contains a Boolean value.
= =	Checks to see if value A is equal to value B. A Boolean value is the output.

Add	It adds the input value to the selected array into the next available slot.
Add Audio Component	Adds an audio component to the player's avatar that can be used to play audio.
Add Child	Adds a child component to a menu widget.
Add Local Rotation	Adds rotation to a component within a blueprint actor. An example would be a static mesh that is rotated.
AI Move To	Moves an AI actor to the "Destination" specified.
And	It takes multiple Boolean inputs with a single Boolean output. The Boolean output will only be true when all of the input Boolean conditions are met.
Append	This node can take several text string inputs and put them together to create a single continuous text string sentence.
Branch	It is a "flow control" node that has a true and false execution output. A single Boolean input determines the output.
Break Vector	This node takes a single vector value and breaks it into the resulting X, Y, and Z components.

Cast to	This node interacts with an actor or blueprint external to the blueprint that makes the call.
Clear and Invalidate Timer by Handle	It uses an independent timer's "handle" or name to stop and clears the timer from using CPU resources to calculate time.
Create Dynamic Material Instance	It creates an instance of a dynamic material so that its properties can be accessed and changed.
Create Dynamic Text Box Widget	It creates a new widget with the name "Dynamic Text Box". It dynamically adds it into another widget.
Decal Actor	It is an actor that can be placed in a level that can project a decal texture on a nearby surface. Text painted on a wall is an example of a decal actor.
Delay	It adds a delay within a blueprint sequence. It is used to sync certain scripted blueprint events or provide the blueprint with enough time to complete tedious operations.
Destroy Actor	It destroys a referenced actor that was placed or spawned into a level.
Disable Input	It disables the player's controls.

Do Once	It executes a string of blueprint nodes following this one once. It is useful when an "Event Tick" node executes the whole blueprint sequence every frame.
Enable Input	It enables the player's controls.
Event Begin Play	It is a universal node that exists in every blueprint. It activates the moment the level starts up and executes once only.
Event Construct	It is like the "Event Begin Play" node in that it operates once only. It is exclusively used within menu actors the moment the menu actor activates.
Event Pre-Construct	It is like the "Event Begin Play" node in that it operates once only. It is exclusively used within menu actors before the menu actor activates. It is an expensive node that should be used sparingly.
Event Tick	It is a universal node that exists in every blueprint. Once the level starts, this node fires continuously until the level is unloaded. The execution speed is linked to the framerate.
Flip Flop	It is a switch node that works together with input events. Once an input event activates it, the input event must trigger again before this node deactivates.

For Each Loop	It is a special loop node designed to extract the contents out of an array.
For Loop	It is a flow control node that indefinitely loops through a blueprint script attached to it until the loop is stopped.
Get Actor Location	It retrieves the location of the player and provides a vector value that can plug into another node.
Get Component Velocity	It retrieves the velocity of the linked actor and outputs a vector value.
Get Display Name	It retrieves the name of the object a ray trace hits and outputs a text string value.
Get Game Instance	It retrieves the game instance actor.
Get HUD	It retrieves the HUD or “Heads Up Display” component of the selected player actor.
Get Player Camera Manager	It retrieves the camera manager used by the current player pawn.
Get Player Character	It retrieves the player character that is currently in control.
Get Player Controller	It retrieves the player input controller that is currently in use.

Get Player Pawn	It retrieves the player pawn in its entirety. It can be used to test overlaps on trigger volumes.
Get Random Reachable Point in Radius	It takes a vector location and finds a destination location within the radius specified. It can only return a location that falls within a reachable point on the current navigation mesh.
Inputs	These are execution inputs that are used inside macros and functions.
Is Overlapping Actor	It checks whether two actors are overlapping and return a Boolean value.
Last Index	It retrieves the last index within a populated array.
Lerp (vector)	It is a linear interpolation node that smoothly blends between two vector values using an alpha input.
Line trace by channel	It shoots out a line trace using a start and end vector that hits items based on the selected channel.
Load Game from Slot	It loads a save file based on the name that is typed into the “Slot Name” box. If the file does not exist, nothing loads.

Make Transform	It takes individual transform inputs and combines them into a single transform output. A transform includes location, rotation and scale values.
Make Vector	It takes individual vector values and combines them into a single vector output. A vector includes an X, Y and Z value.
Map Range Clamped	It converts the input range of two float values to an output range of two newly specified float values.
Material Interface	It is an actor that allows a blueprint to communicate with the variables inside a material instance.
Multi Gate	It is a flow control node that allows several separate scripts to execute sequentially or randomly. It can execute once or loop indefinitely.
Normalize to Range	It takes a single float value and normalises the value within the range of a minimum value and a maximum value.
On Released	It is a button function that executes the blueprint script the moment the button, which was already pressed, is released.

Or	It is a Boolean test node that tests whether any of the input Boolean nodes are true. If any of the inputs are true, the resulting output of this node will also be true.
Outputs	These are execution outputs that are used inside macros and functions.
Play	It plays an audio component that was spawned into the level.
Play Sound 2D	It plays a sound that blends straight into the audio mix without having to be spawned into the level first.
Play Sound at Location	It plays a sound at a location in the level using a vector value as an input.
Play World Camera Shake	It plays a camera shake effect at the specified vector location in the level. It is a positional effect, so an inner and outer radius values need to be specified as well.
Print String	It is a development node that prints various variables to the screen. It takes in various inputs that are converted into a string value that can be displayed on the screen.
Random (Audio Cue)	It is a node that works exclusively inside an audio cue actor.

Random Float in Range	It outputs a float value that exists within the range between the “min” and “max” input values.
Random Integer in Range	It outputs an integer value that exists within the range between the “min” and “max” input values.
Remove all Widgets	Removes all the menu widgets that are currently loaded to show on the screen.
Save Game to Slot	It takes all the variables that are saved in the save game object and saves them into the save game file specified in the “Slot Name” text box.
Sequence	It executes all the blueprint nodes that are attached to the output plugs simultaneously.
Set	It reads an input value into a local or global variable. Set nodes work for most variable types.
Set Actor Enable Collision	It enables the collision of the actor references within the current level.
Set Actor Hidden in Game	It hides, without completely removing, an actor within the current level.
Set Actor Relative Location	It sets the location of an actor, relative to its current vector location. It does not take the world vector location into account.

Set Collision Enabled	It enables the collision of a shape or trigger volume placed in the level.
Set Decal Material	It replaces the current material applied to a decal with a new one.
Set Intensity	It sets the luminance level of a light component.
Set Low Pass Filter Enabled	It enables the low pass filter attached to an audio component.
Set Low Pass Filter Frequency	It sets the frequency level of the low pass filter attached to an audio component.
Set Relative Location	It is like the “Set Actor Relative Location” in that it sets the referenced object’s vector location relative to its previous vector location. It does not take the world vector location into account.
Set Scalar Parameter Value	It sets the scalar parameter value on the referenced actor. It is usually used to adjust specific parameter values within a material instance.
Set Scalar Parameter Value on Materials	It is like the “Set Scalar Parameter Value” node, in that it adjusts specific parameter values, but targets the scalar parameter values in a material instance directly.

Set Static Mesh	It replaces the referenced input static mesh with the one specified in the “New Mesh” drop-down.
Set Timer by Function Name	It creates a timer that executes a custom function. The name of the function is specified in the “Function Name” text box. The duration of the timer can be specified in seconds, and the timer can be set to loop until it is invalidated and cleared.
Set Vector Parameter Value on Materials	It is like the “Set Scalar Parameter Value on Materials” node, in that it adjusts certain parameter values, but targets the vector parameter values in a material instance directly.
Set View Target with Blend	It smoothly blends the perspective of the current in-game camera with the new camera that is specified, without any noticeable cuts or hitches. Several values, including the “Blend Time”, can be specified.
Set Volume Multiplier	It sets the volume level of the linked audio component.
Spawn Actor	It spawns the specified blueprint actor to a specified transform in the level. The selected blueprint to spawn is specified under “Class”.

Spawn AI from Class	It spawns the specified AI actor to a specified vector location in the level. The selected blueprint to spawn is specified under “Pawn Class”.
Spawn Default Controller	It spawns a default controller to the camera that is currently active. This is a required action, even when the player is unable to control the currently active camera.
Spawn Sound 2D	It works in the same way as the “Play Sound 2D” node. It plays a sound that blends straight into the audio mix after being spawned into the level.
Spawn Sound at Location	It works in the same way as the “Play Sound at Location” node. It spawns a sound actor at a location in the level using a transform input value to determine its location and rotation.
Start Camera Fade	It fades the currently active camera to or from a solid colour. The colour, “From Alpha”, “To Alpha” and “Duration” values can be altered for different effects.
Stop	It stops playback of the linked audio component.
Switch on Int	This is a flow control node that chooses an execution path based on the integer value that is fed into the node.

Target	It retrieves specific components from actors in the level so that their parameters can be adjusted.
Timeline	A timeline node consists of a curve that can be set up to translate in various ways depending on how the node structure was set up.
To Text (Integer)	It converts an integer input to text. This text can be used in menu widgets.
To Text (String)	It converts a string input to text. This text can be used in menu widgets.
Vector Length	It calculates the distance between two vector values and outputs this distance as a float value.

Appendix 3: Event Setup Menu

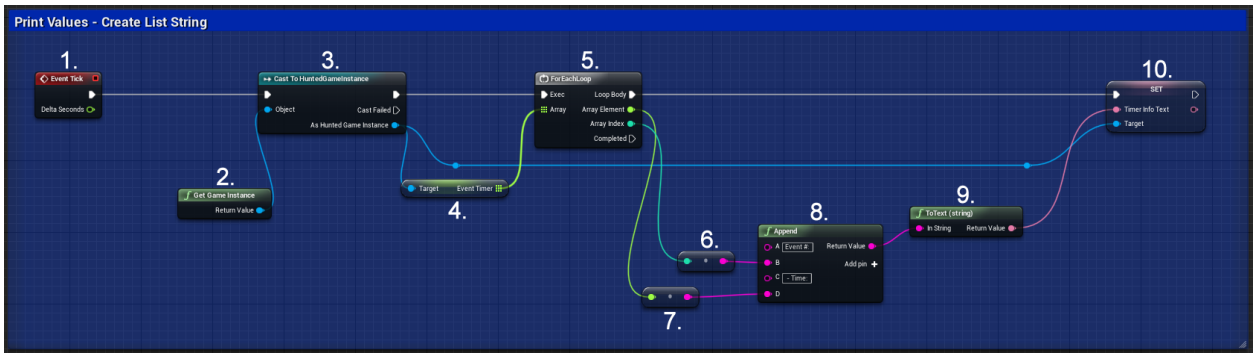


Figure 58. Event string creation.

Figure 58 shows the "Print Values" that is in the level blueprint. This blueprint takes the durations that are fed into the event setup menu and converts them into plain text strings that can be fed into a scroll box element on the menu. The "Event Tick" node executes this blueprint every frame, with nodes (2) and (3) calling up a reference of the game instance. The "Event Timer" node (4) is the array, which is resident within the game instance, that contains all of the event durations that will be converted to text. The "For Each Loop" node (5) loops through the entire array while extracting each element one by one. Each "Array Index" and "Array Element" are converted to text by nodes (6) and (7) respectively. These two nodes then plug into the "Append" node (8) which combines the values extracted out of the array with predetermined text. "Event #: 3 - Time: 30.0" is an example of how the text can appear. The "To Text (string)" node converts this text into a string that can be used as a text string in a scroll box menu element. This text string is then dynamically fed into the "Timer Info Text" variable using a "Set" node. Due to this blueprint updating every frame, the "Timer Info Text" value is continually changing. It is only when the player or researchers presses the "Add" button on the "Event Setup" menu that the text string gets fed as a separate value into the scroll box.

The event setup menu, as shown in figure 37, also has blueprints associated with it.

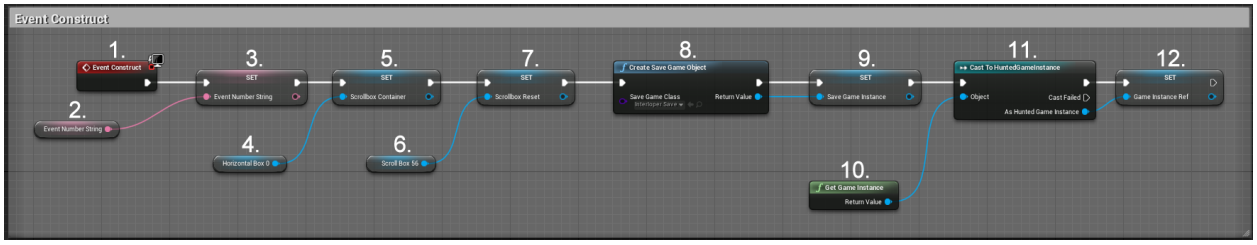


Figure 59. Event Construct.

Figure 59 shows the "Event Construct" blueprint sequence. This script is executed the moment the menu system is launched with the "Event Construct" node (1). The "Event Number String" node (2) resets itself by feeding a value of "0" back into itself using the "Set" node (3). The "Horizontal Box 0" node (4), which is a reference of this element in the menu, is fed into a variable node called "Scroll Box Container" (5). "Scroll Box 56" is also fed into its variable called "Scroll box Reset" like the menu element above. Loading these elements into variables will allow the system to reference them later or even reset them.

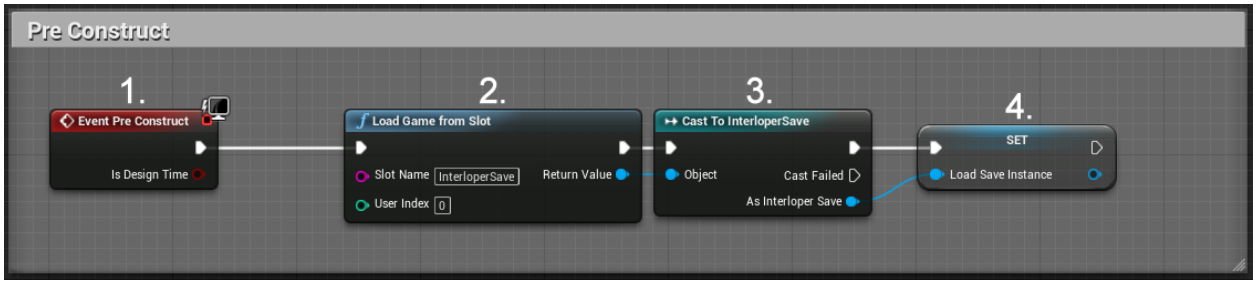


Figure 60. Event Pre Construct.

Figure 60 shows the "Pre Construct" blueprint. The "Event Pre Construct" node (1) executes these nodes when the scene launches and before the menu gets called. These nodes can crash the editor, so its intended use is for simple cosmetic changes using local variables. Interloper uses it to load the save game file and save a reference variable of it. The "Load Game from Slot" node (2) loads the save game file, which is named "InterloperSave", while

the “Cast to InterloperSave” node (3) interacts with the contents of this file. The “Set” node (4) takes a reference of everything within the save file and loads it into a variable called “Load Save Instance”.

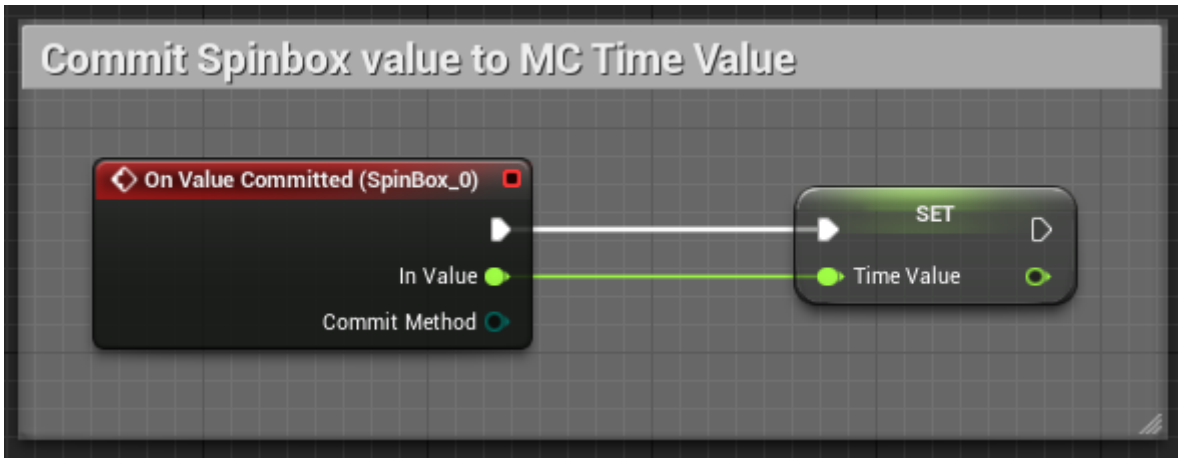


Figure 61. Commit Spinbox Value.

The value that is entered into the “Camera Event Delay” spin box needs to be saved into a variable. Figure 61 shows how this value is saved into a variable called “Time Value” with a “Set” node.

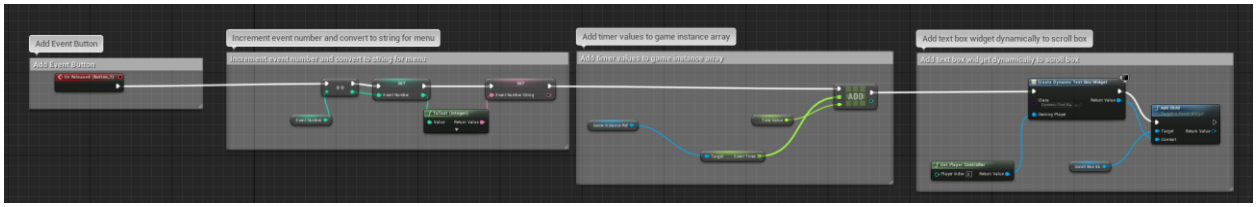


Figure 62. Blueprint that loads the event text into scroll box.

Figure 62 shows the set of blueprints that loads the event text into the scroll box.

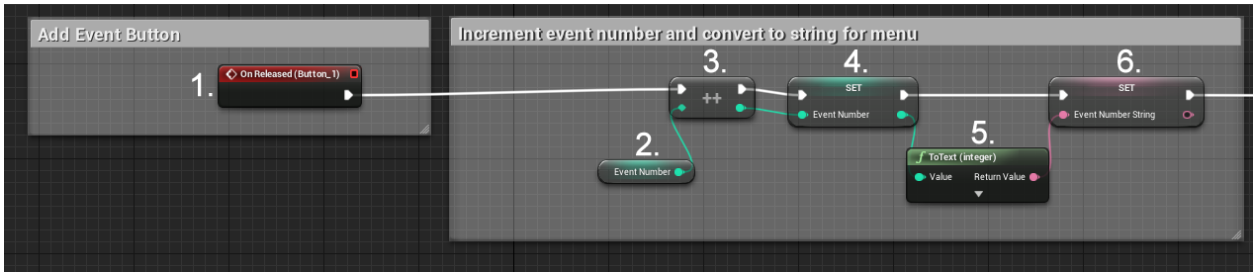


Figure 63. Blueprint that increments event number.

A closeup of the first part of this blueprint sequence is shown in figure 63. The "On Released (Button_1)" node (1) listens for the "Add Event" button to be pressed on the "Event Setup" menu. Once pressed, the "Event Number" variable node (2), which holds a default value of "0", is incremented by the "+ +" node (3). This new value is then saved back into the "Event Number" variable with the "Set" node (4). The "To Text (integer)" node (5) converts this value into text. The resulting text is then saved into the "Event Number String" variable with a "Set" node (6).

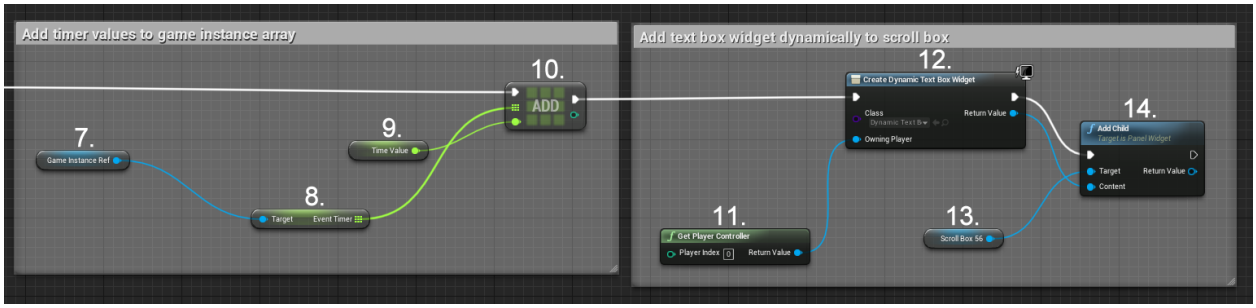


Figure 64. Blueprint that ads timer value to array.

Figure 64 shows the second part of the blueprint. Adding these "Monster Camera" timer values to "Event Timer" array is the next step. The game instance is referenced with the "Game Instance Ref" node (7). Out of this reference, the "Event Timer" variable node (8) is retrieved. The "Time Value" variable node (9) references the value that was entered earlier using the spin box commit node. This value is piped into the array at the first available location using the "Add" node (10). The next part dynamically adds the text to a text box

widget created for this purpose. The “Get Player Controller” node (11) is required to create the text box widget with the “Create Dynamic Text Box Widget” node (12). Under “Class” the text box widget in question is selected. The “Scroll Box 56” node references the scroll box that will be the destination for the text box that will be added. The “Add Child” node adds the dynamic text box into the scroll box.



Figure 65. Dynamic text box hierarchy.

The “Dynamic Text Box” is an entirely separate menu widget that gets embedded into the primary “Event Setup” menu widget. Figure 65 shows the hierarchy of this widget. It is clear that the only element within the “Dynamic Text Box” widget is a “Text Block”. Every time the “Add” button is pressed in the “Event Setup” menu, a copy of this widget is added to the scroll box with the relevant information.

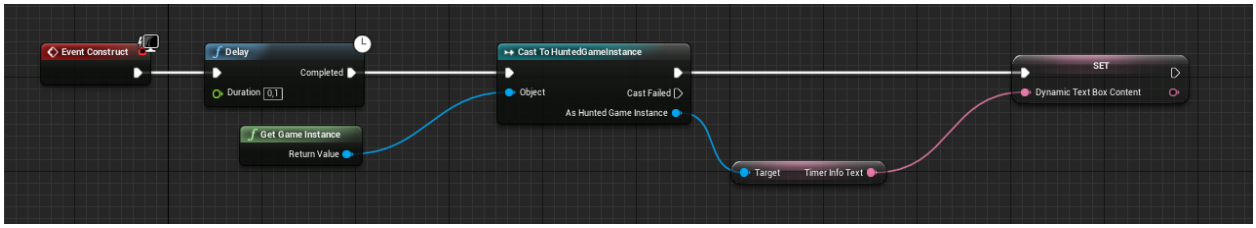


Figure 66. Blueprint that loads dynamic text into game instance.

Figure 66 shows that within the blueprints of this “Dynamic Text Box” widget, the “Timer Info Text” is extracted out of the game instance and saved into the “Dynamic Text Box Content” variable that can be referenced and used locally.

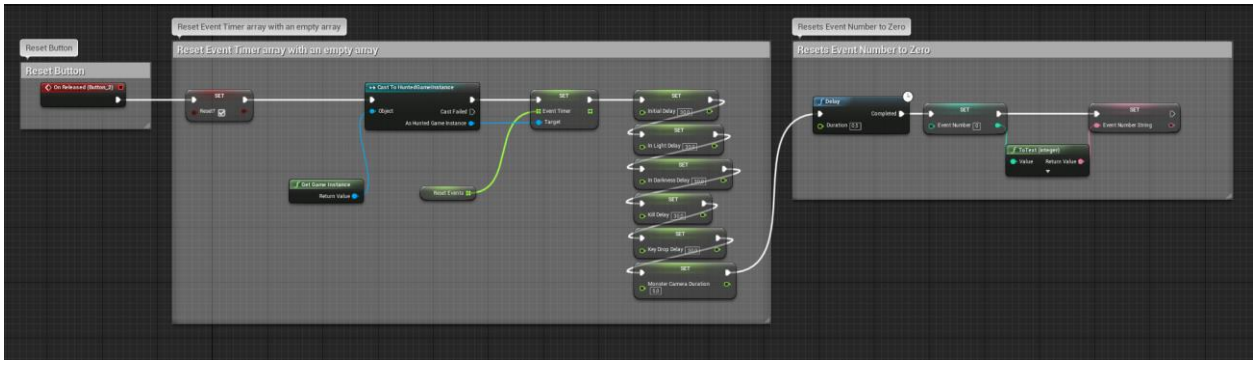


Figure 67. Blueprint that resets events.

Figure 67 shows the blueprint sequence that occurs when the “Reset” button is pressed in the “Event Setup” menu.

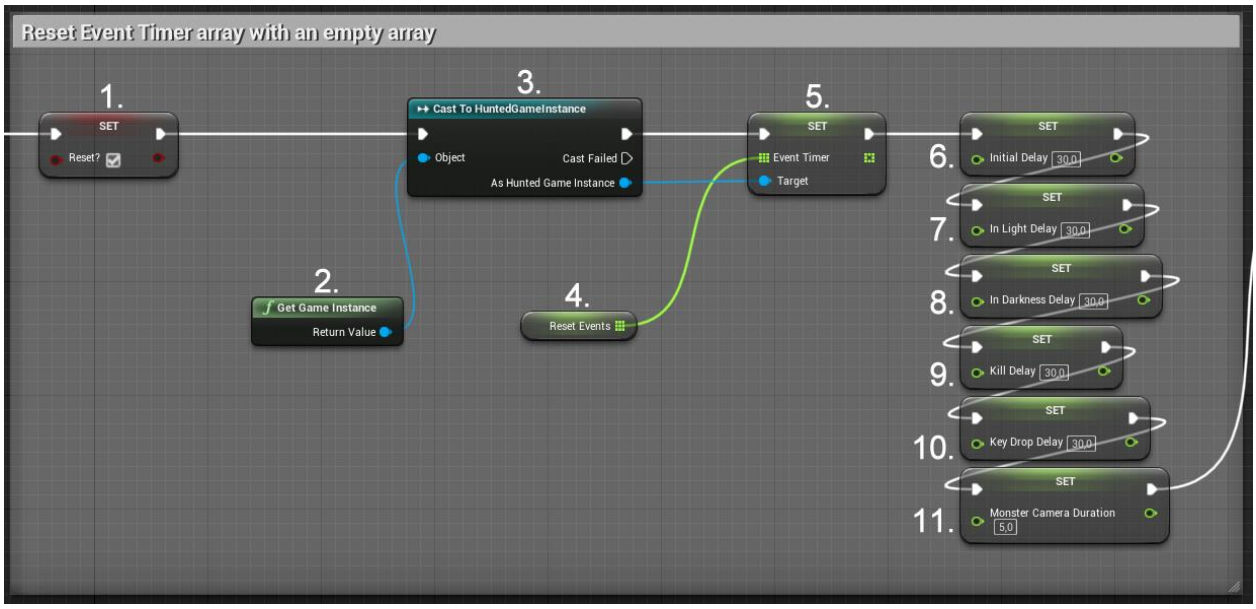


Figure 68. Blueprint that resets values.

The first part of this “Reset” blueprint is depicted in figure 68. The “Reset?” Boolean variable is set to true using a “Set” node (1). This action allows other blueprints to track whether the events that were present have been reset or not. Beyond these initial nodes, default values are fed into the main variables used to spawn events, so that they can be reset to their original states. Nodes (2) and (3) retrieves a reference of the game instance and

actuates it for use. The “Event Timer” variable is taken straight from the game instance so that the “Reset Events” variable node (4), which is an empty array that is held locally, can be fed into it using a “Set” node (5). “Set” nodes (6) through (11) are used to feed the default values into “Initial Delay”, “In Light Delay”, “In Darkness Delay”, “Kill Delay”, “Key Drop Delay”, and “Monster Camera Duration”.

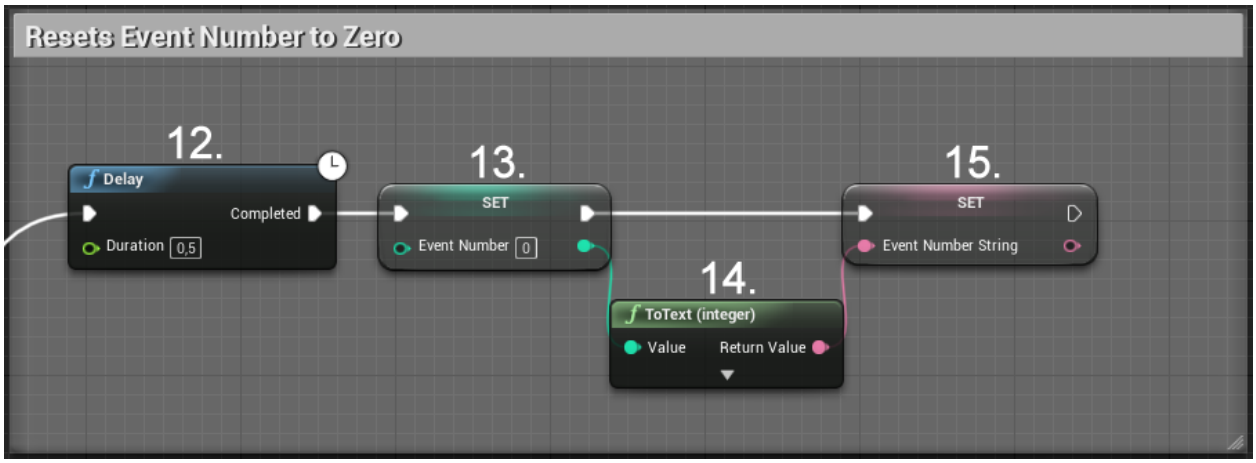


Figure 69. Blueprint that resets event number to zero.

The next string of blueprints resets the “Event Number” variable to zero, as shown in figure 69. A “Delay” node was used because the system did not delete the “Event Number” while it was busy deleting the items out of the scroll box, leaving some residual items. The Delay gives the system a slight respite so that it can adequately clear out everything. The zero value is also converted to a text string by the “To Text (integer)” node (14) so that this value can be inserted into the “Event Number String” variable node (15) and displayed it in the menu system.

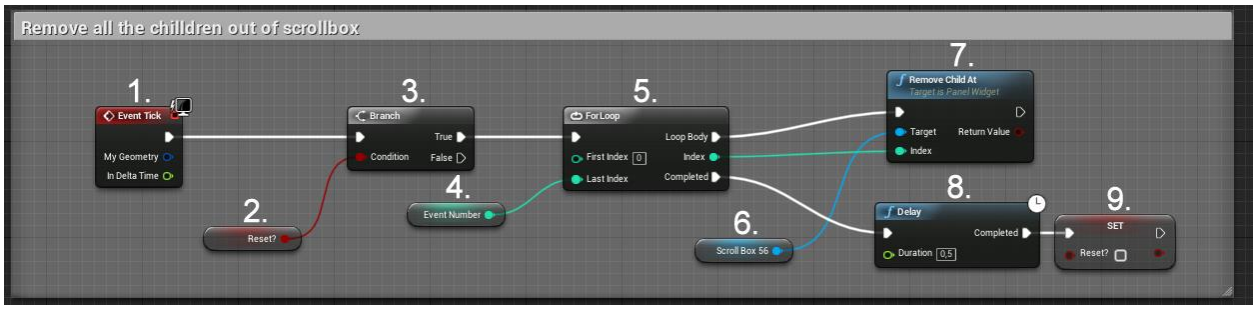


Figure 70. Blueprint that removes text out of scroll box.

Figure 70 shows the "Remove all the children out of scroll box" blueprint that uses an "Event Tick" node to execute the blueprint every frame. It is designed to remove the elements out of the scroll box when the "Branch" node (4) tests true using the "Reset?" variable node (2) as an input. When true, a "For Loop" node takes a reference of the "Scroll Box 56" variable (6) and removes the child text box widget at the index indicated by the for loop and stops when the last index, represented by the "Event Number" variable (4) is reached. When the "For Loop" completes a slight "Delay" (8) precedes the "Reset?" variable being "Set" (9) to false.

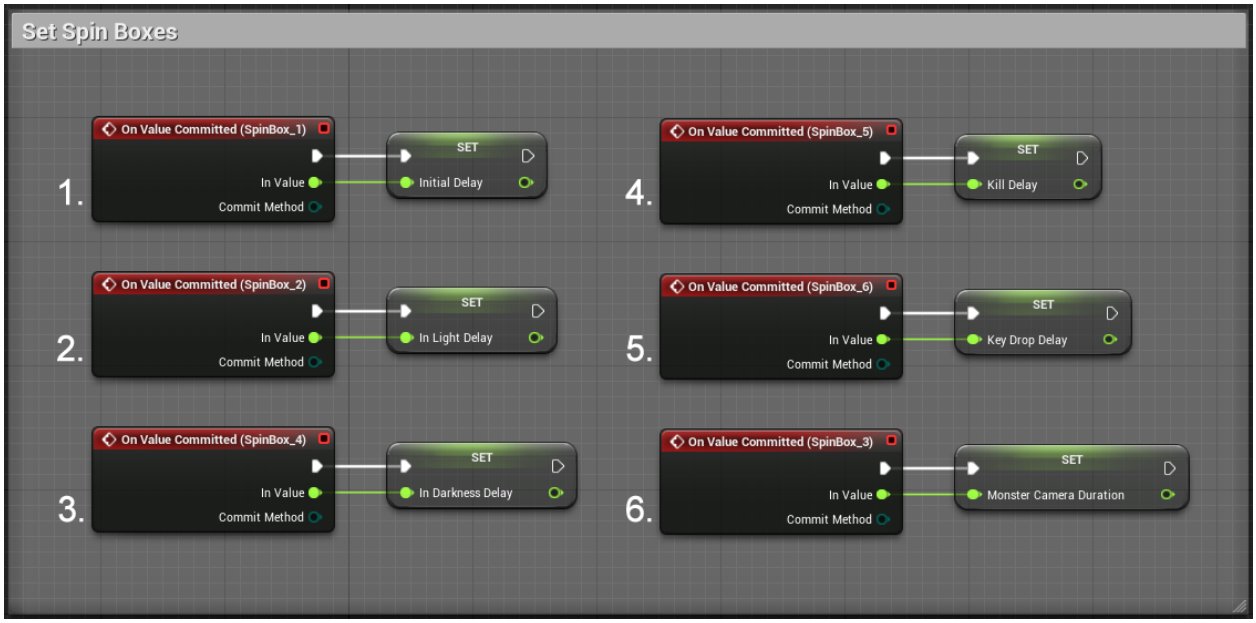


Figure 71. Blueprint that commits spin box values to variables.

The values of several spin boxes on the "Event Setup" menu need to be fed into their respective variable so that they can be saved and used. Figure 71 shows the "On Value Committed" commands that retrieve the value committed on each spin box and the variables that they are fed into using a "Set" node on each.

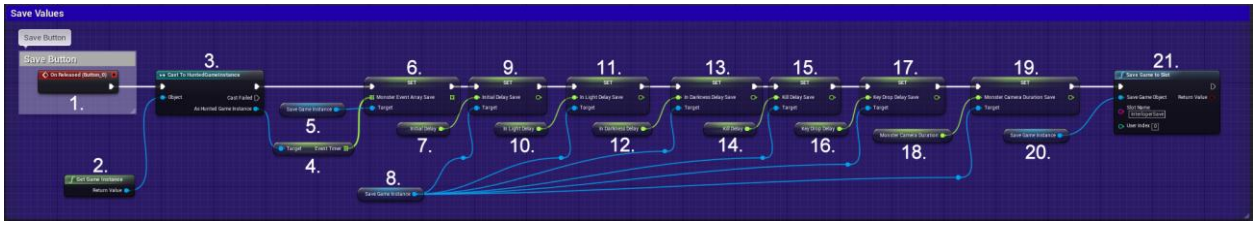


Figure 72. Blueprint that saves spin box values to file on the computer.

Figure 72 shows how the "Save" button operates. The "On Released (Button_0)" node listens for a save button press. Once pressed, nodes (2) and (3) retrieve and ready the game instance so that the "Event Timer" array variable (4) can be pulled out and saved. The "Save Game Instance" variable nodes (5 & 8) are used to quickly reference the save game object that was created earlier (figure 60). Nodes (7), (10), (12), (14), (16), and (18) are

variable nodes that are filled when their respective spin box values are committed on the "Event Setup" menu. Using the "Save Game Instance" variable node, these variables are then loaded into their corresponding save game variables using "Set" nodes (9), (11), (13), (15), (17), and (19). Once these variables are loaded another "Save Game Instance" node is used to save the game to a slot using the "Save Game to Slot" node (21). The name of the file saved to the computer must be specified under the "Slot Name" input for this to work.

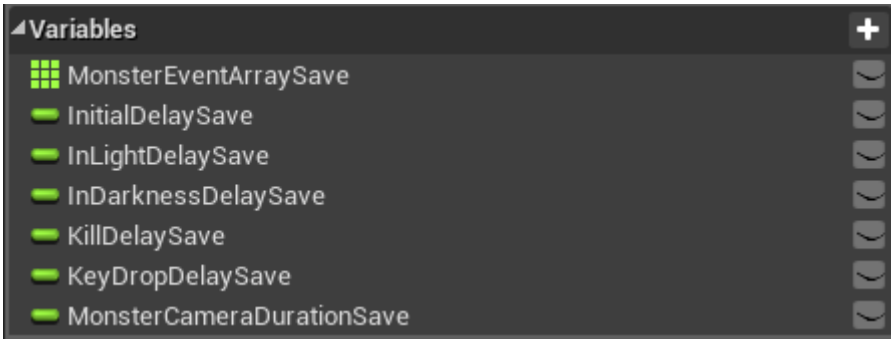


Figure 73. Save game variables.

Figure 73 shows the variables housed within the save game object called "Interloper Save", which corresponds with the variables that are loaded into them when the save button is pressed in the "Event Setup" menu.

Appendix 4: Level Scripting – Office

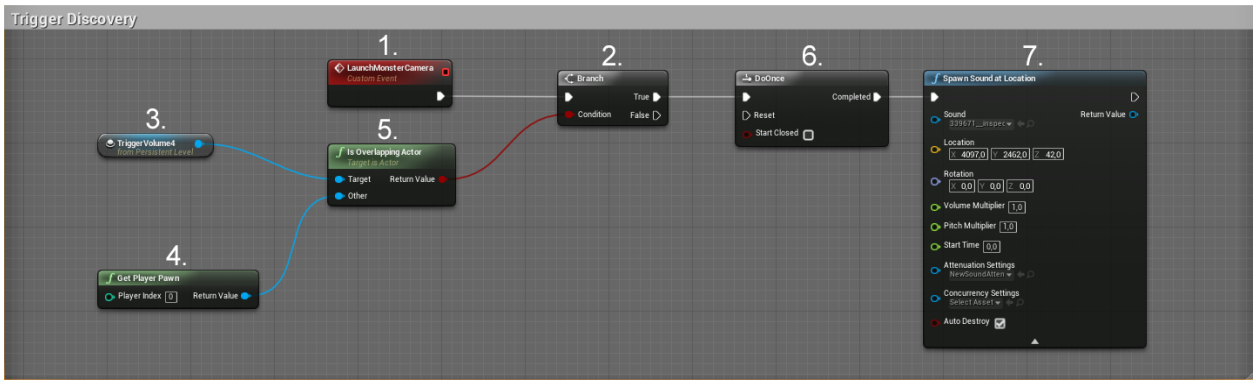


Figure 74. Trigger that plays the discovery sound.

Figure 74 shows how this sound effect is played. “Launch Monster Camera” is a custom tick event node (1) that feeds into a “Branch” node (2). An overlap test will determine whether this “Branch” node (2) executes the next node (6). The “TriggerVolume4” node (3) references the trigger box in the scene, while the “Get Player Pawn” node (4) retrieves a reference of the player’s character. The “Is Overlapping Actor” node (5) determines whether the player or the trigger box is overlapping at any point in the game. When they do overlap, a Boolean signal is sent to the “Branch” node (2), setting its value to true. When false, the “Branch” node (2) executes nothing, but when true, the “Branch” node (2) executes a “Do Once” node (6). The “Do Once” node (6) executes a single instance of the “Spawn Sound at Location” node (7). This node spawns the sounds at a custom vector location specified under the “Location” property.

Appendix 5: Level Scripting – Flashlight

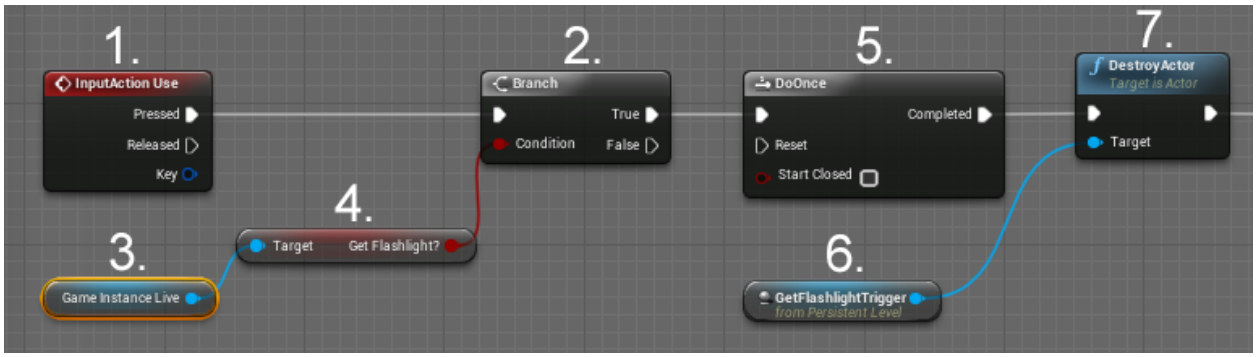


Figure 75. The input action that triggers the flashlight collection.

In figure 75, the "Input Action Use" node (1) listens for a player to press the "Use" key. When the player presses this key, a "Branch" node (2) is executed. A "Game Instance Live" node (3) is used to extract a variable out of the game instance using the "Target" node (4). The "Get Flashlight?" variable then determines whether the "Branch" node (2) is either true or false. The "Get Flashlight?" variable contains a Boolean value that tells the "Branch" node (2) whether the player is within range of the flashlight and looking at it. When true, the "Do Once" node (5) executes the next series of nodes once. The "Get Flashlight Trigger" node (6) references the collision sphere in the level. It is the collision sphere (1) visible in figure 39. The "Get Flashlight Trigger" node feeds into the "Destroy Actor" node (7). When node (7) is executed, the collision sphere is removed from the level, disallowing the player from interacting with the trigger area again. The destruction of the collision sphere happens at practically the same time as the flashlight retrieval animation.

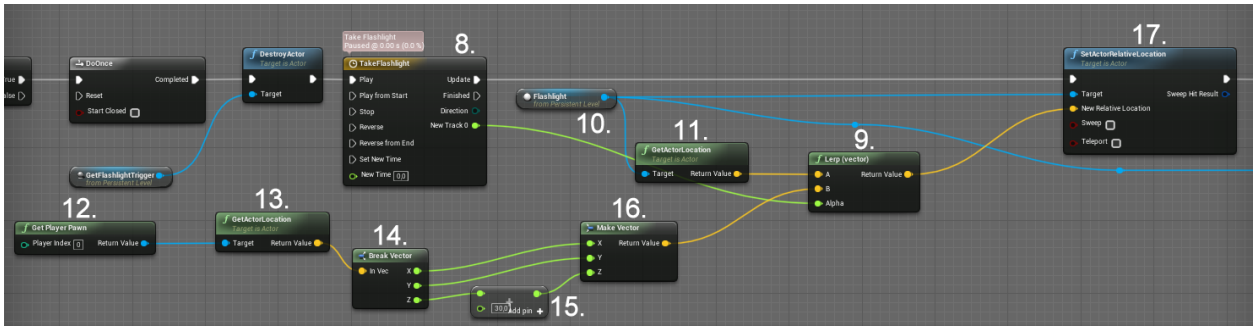


Figure 76. The animation system that moves the flashlight to the player's location.

After the collision sphere is destroyed, a timeline node is executed. Figure 76 shows this timeline node (8) called "Take Flashlight" and is used to translate the flashlight from the table to the player's location. Although an animation showing the representation of a hand picking up the flashlight might have been a more realistic depiction, the goal is to show the player retrieving the flashlight in the most efficient way possible. This method should effectively communicate to the player that the flashlight was retrieved for use. A message will pop up on the screen that instructs the player in the use of the flashlight the moment it is retrieved.

Moving the flashlight to the player is similar to the technique used to move the elevator doors, the sliding doors and the metal gate. A "Lerp" node (9), linearly interpolates the flashlight to the player. The "Flashlight" node (10) is a reference of the flashlight actor in the level.

The “Get Actor Location” node (11) extracts the location of the “Flashlight” node (10) and plugs it into “A” on the “Lerp” node (9). The “Get Player Pawn” node (12) references the player, and the “Get Actor Location” node (13) extracts its location. The “Break Vector” node (14) expands the “X”, “Y”, and “Z” coordinates into separate outputs. A “+” node (15) adds “30.0” to the “Z” value extracted out of the player’s location. The addition of this value moves the retrieval point down to the centre of the player, preventing any viewpoint obstructions when the player absorbs the flashlight.

The “Make Vector” node (16) combines these three values into a single vector value. This single value then plugs into “B” on the “Lerp” node (9). The “Take Flashlight” timeline node (8) is plugged into the “alpha” of the “Lerp” node (9). Combining these nodes drives the translation between points “A” and “B”. Plugging the “Lerp” node (9) into the “Set Actor Relative Location” node (17) will dynamically drive the movement of the mesh over time, resulting in the flashlight being absorbed by the player when pressing the “Use” key.

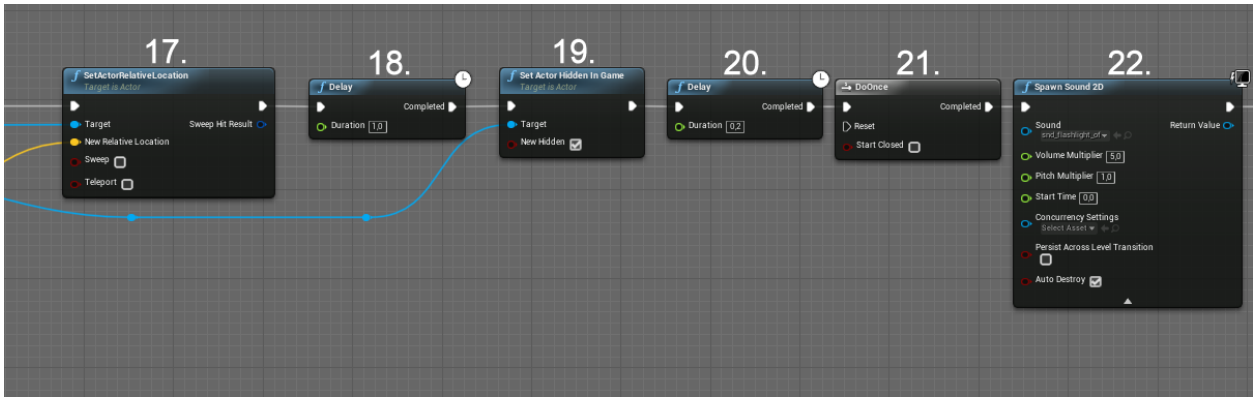


Figure 77. Blueprint that hides the flashlight and triggers the flashlight button sound.

Figure 77 carries on from the “Set Actor Relative Location” node (17) mentioned above. After the flashlight moves to the camera, a “Delay” node (18) provides the system with enough time to fully move the flashlight to the player before the “Set Actor Hidden in Game” node (19) activates. Hiding the flashlight when it reaches the centre point of the

player makes it appear as if the player collects it. Another “Delay” node (20) is executed before the “Do Once” node (21) launches the “Spawn Sound 2D” node (22). This node spawns a flashlight switch-off sound. The delay was added because the volumetric light shaft effect computes only every other frame and interpolates the effect temporally. The temporal processing produces some lag between a change on the light’s position, intensity or colour before the frames catch up. A delay was therefore required for the sound between the moment the light is switched off in-game and when it visibly goes off on screen.

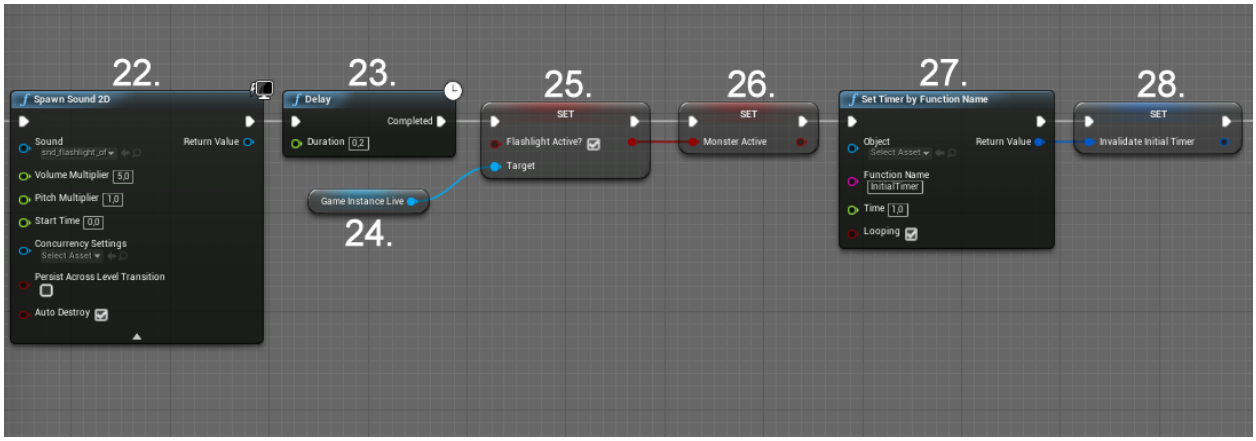


Figure 78. Blueprint that initiates the “Initial Timer”.

In figure 78, the “Spawn Sound 2D” node (22) that plays the flashlight switch-off sound is followed up by another “Delay” node (23). Even though most blueprint nodes follow sequentially, they execute at the same time due to the speed of the CPU. “Delay” nodes are added to force some parts of the sequence not to execute until the timer has run out. These timing elements are used to sync certain events, such as the playing of individual sounds, with the events that happen in the level.

After the “Delay” node (23) a reference of the game instance (24) is used to set a game instance variable “Flashlight Active?” to be true. The local variable “Monster Active” is also set to true.

The “Set Timer by Function Name” node (27) launches one of several timers, which is called the “Initial Timer”. The “Initial Timer” provides a period for the event system to spawn the “Monster Camera” events that merely roam around, looking for the player. The “Set” node (28) saves a reference of the “Initial Timer” into a variable called “Invalidate Initial Timer”. This variable can be referenced whenever the timer needs to be deactivated.

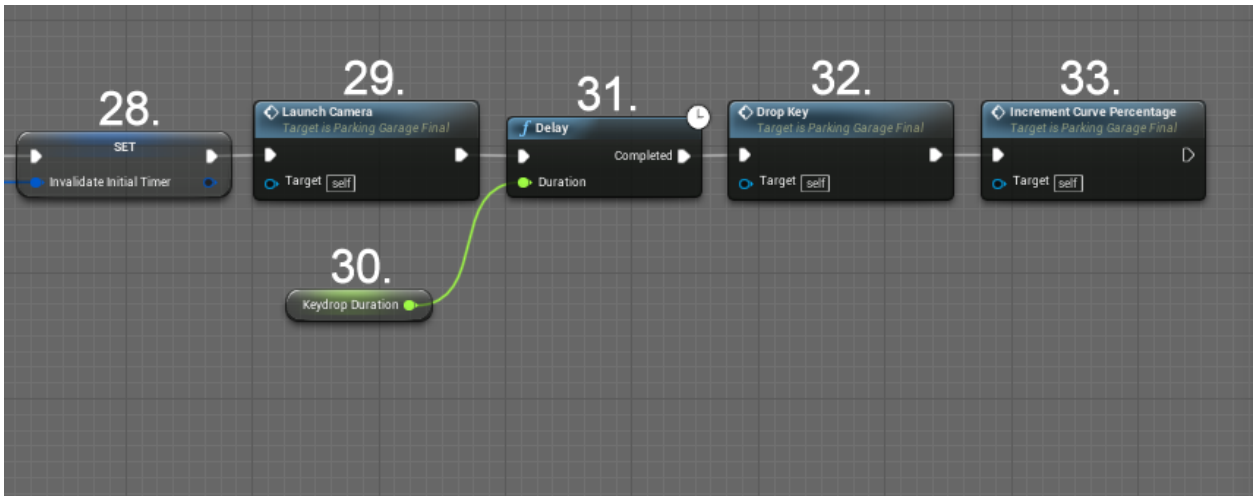


Figure 79. Blueprint that launches the first “Monster Camera”.

Figure 79 shows that after the reference of the “Initial Timer” was saved as a variable called “Invalidate Initial Timer” (28), the “Launch Camera” node (29) is executed. This node is a function that launches additional code to spawn the “Monster Camera”. The “Launch Camera” function will be explored later. The next node is a float variable node called “Keydrop Duration” (30). This node stores the amount of time between the moment the player collects the flashlight and the moment the gate key is dropped in the level. The duration of this variable node drives the duration of the next “Delay” node (31). After this delay lapses, two more functions are executed. The “Drop Key” node (32) facilitates the random placement of the gate key in the level while the “Increment Curve Percentage” node (33) keeps track of the game progression relative to the overall length of the suspense curve.

Appendix 6: Level Scripting – Procedural Car Wheel

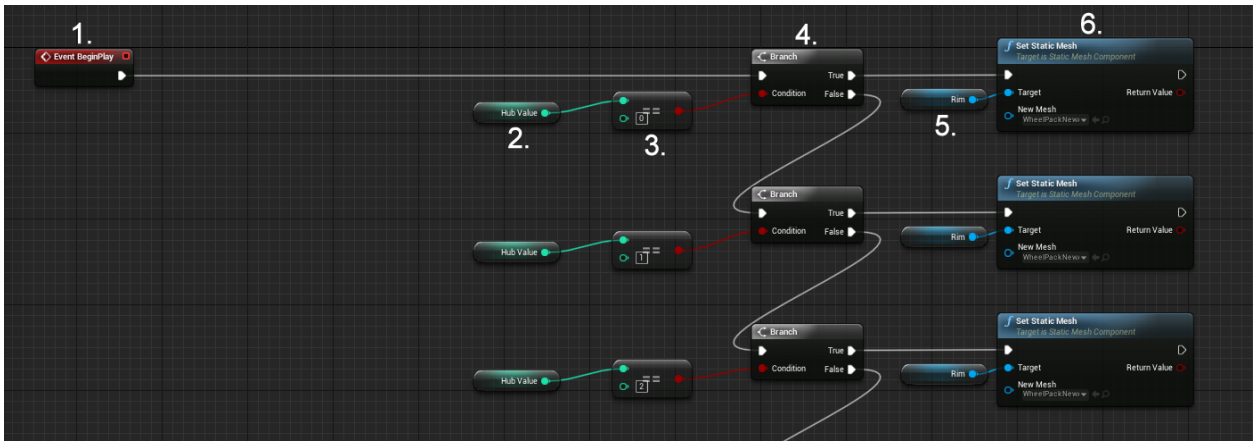


Figure 80. Procedural wheel rim selection closeup.

Looking closer at the blueprint (figure 80) it seems relatively benign. The moment the "Event Begin Play" node (1) launches the blueprint, a "Branch" node (4) conducts a test. It makes use of the value stored within the "Hub Value" variable to see if it is equal to the value stored within the "= =" node (3). When true the static mesh is set to the one specified. The "Rim" node (5), which contains a reference to the default rim mesh that is currently placed within the blueprint and replaces it with the one specified in the "Set Static Mesh" node (6). When the "Branch" node (4) tests false, the next "Hub Value" test is conducted until the corresponding "Hub Value" is found.

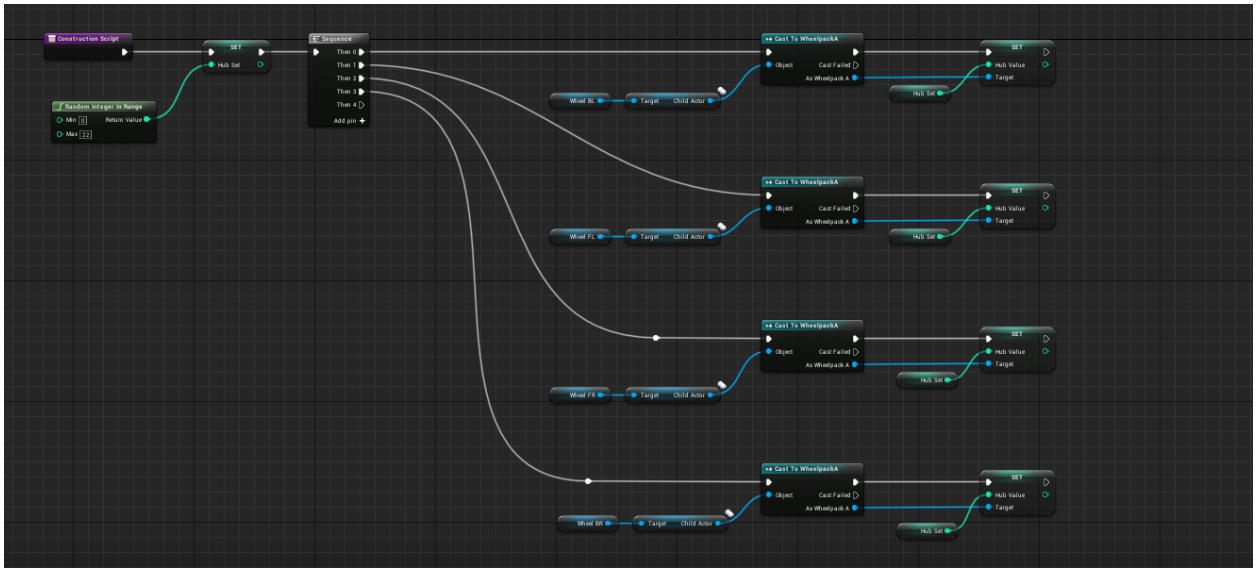


Figure 81. Procedural car "Construction Script" blueprint.

Figure 81 depicts the code that is set up within the "Construction Script" part of the blueprint, which can be used to script the mesh construction components within the blueprint actor.

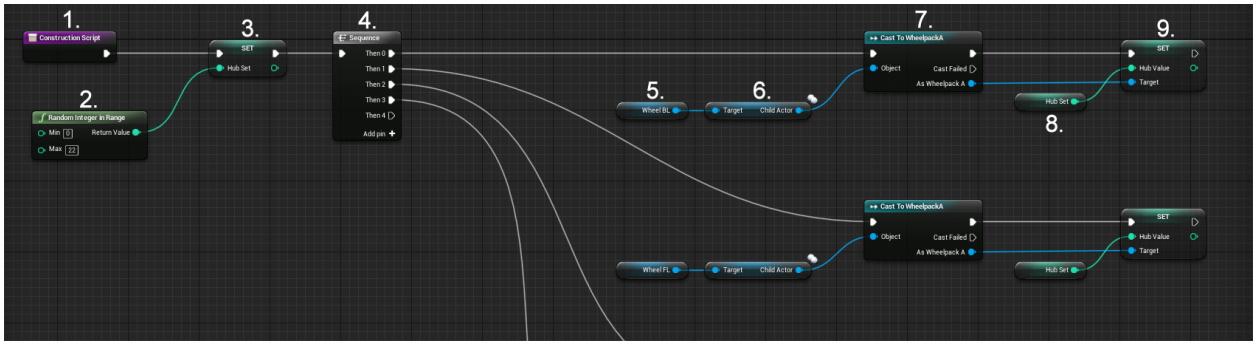


Figure 82. Procedural car "Construction Script" blueprint closeup.

Taking a closer look at the code, as depicted in figure 82, it is possible to see how it functions. Node (1) launches the construction script, after which the "Hub Set" value is determined. A "Random Integer in Range" node (2) choose a random number within the range of "0" and "22", which correspond with the amount of hub cab meshes available in the library. This randomly chosen number is then red into the "Hub Set" variable using a "Set" node (3). A "Sequence" node is used to apply the chosen hubcap to every wheel attached to the car.

Nodes (5) and (6) retrieve the wheel in question and ready it for use. The "Cast to Wheelpack A" node (7) communicates with the wheel blueprint so that its internal variables can be adjusted. The variable value within the "Hub Set" node (8) is determined by the "Random Integer in Range" node (2). It feeds it into a similarly worded variable resident within the wheel blueprint using the "Set" node (9).

Appendix 7: Level Scripting – Car Wheel Rotation & Colour Selection

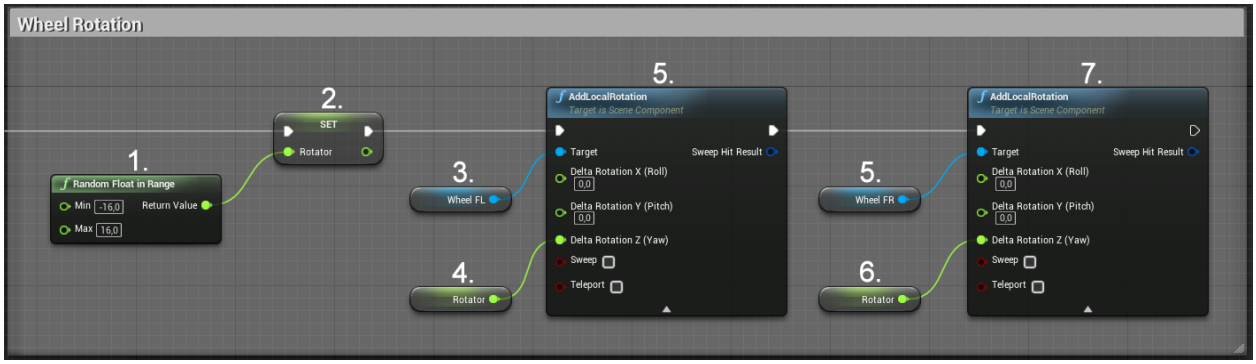


Figure 83. Procedural car wheel rotation closeup.

Figure 83 shows a detailed view of the "Wheel Rotation" script. This script gives the front wheels of the car a slight rotation so that it simulates the rotation the owner left his steering wheel in after turning into the parking space. The "Random Float in Range" node (1) takes the rotational range values entered into the node and generates a random rotational

value within this limit. This value is then fed into the “Rotator” variable using a “Set” node (2). The “Wheel FL” node (3) is a reference of the front left wheel attached to the car, while the “Rotator” node (4) references the rotational value that was set up previously. Plugging these two nodes into the “Add Local Rotation” node (5), allows the front left wheel to rotate with the amount that was set up in the “Rotator” variable. The “Wheel FR” node references the front right wheel. The same process is followed for this wheel with the “Rotator” variable node (6) driving the rotation of the wheel with the node (7).

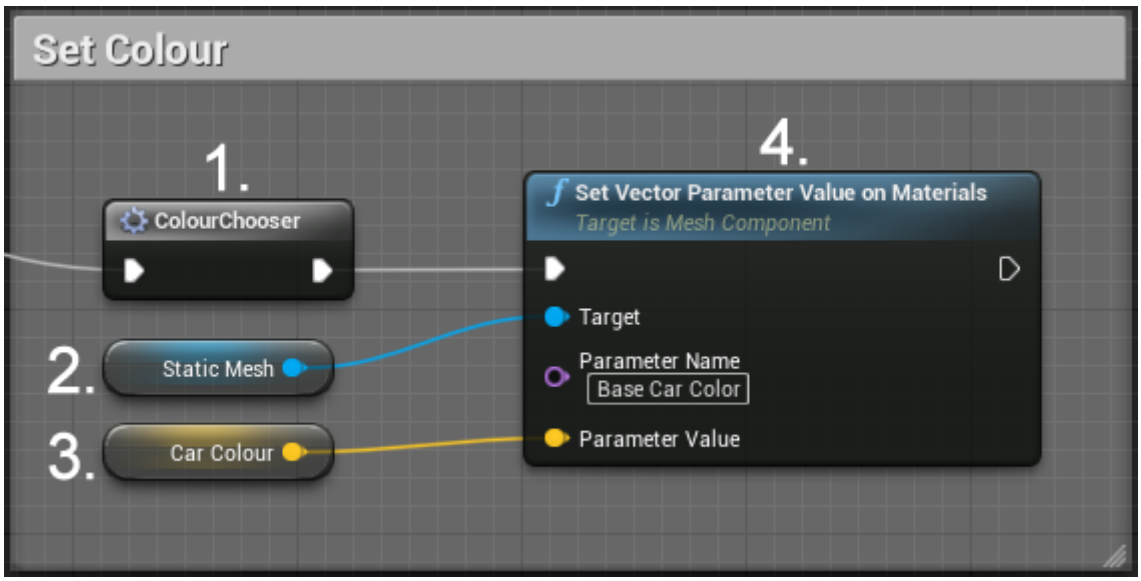


Figure 84. Procedural car colour selection system closeup.

Figure 84 shows the next script that changes the colour of the car. The “Colour Chooser” node (1) launches a separate macro that manages the various colours that can be chosen from and will be explored below. The “Static Mesh” node (2) references the body mesh of the car, while the “Car Colour” node (3), which is defined in the “Colour Chooser” macro, set the colour of the car. These two nodes are plugged into the “Set Vector Parameter Value on Materials” node (4) and change the parameter specified in the “Parameter Name” text box.

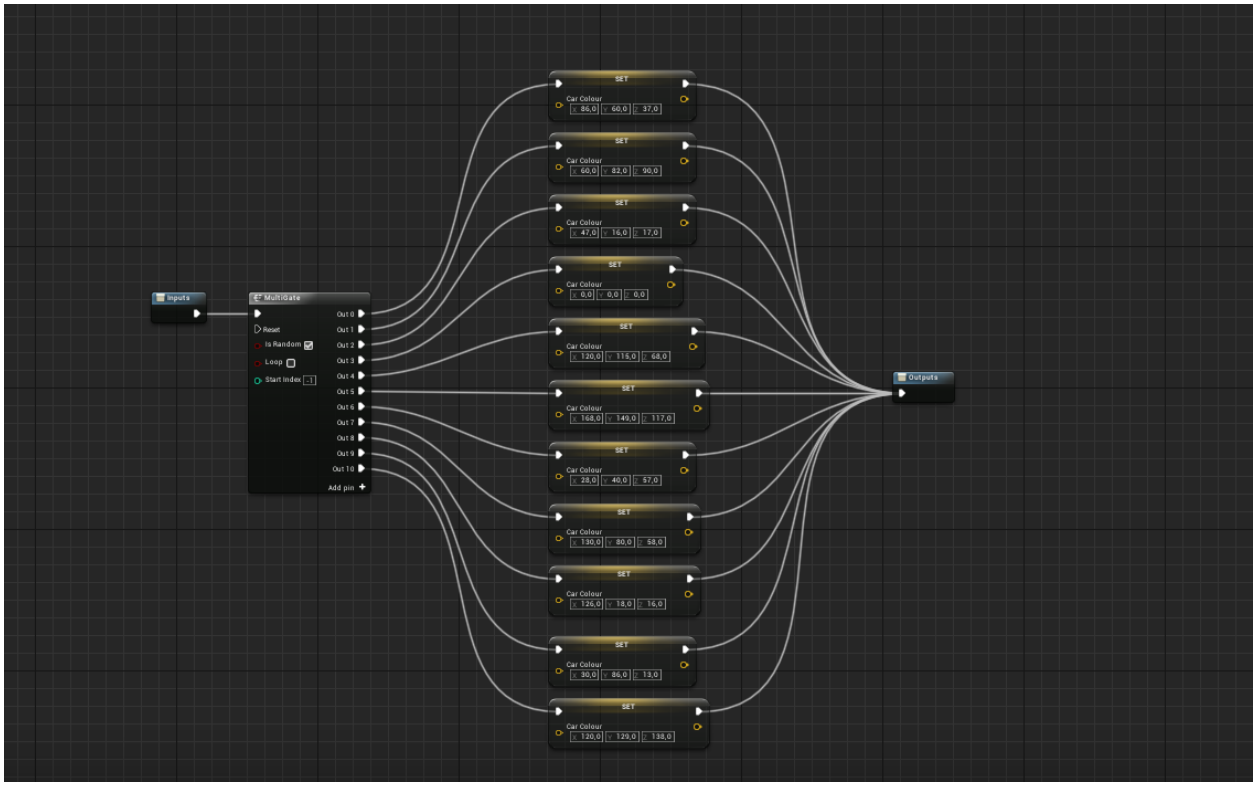


Figure 85. Procedural "Colour Chooser" macro.

Figure 85 depicts the overall appearance of the "Colour Chooser" macro.

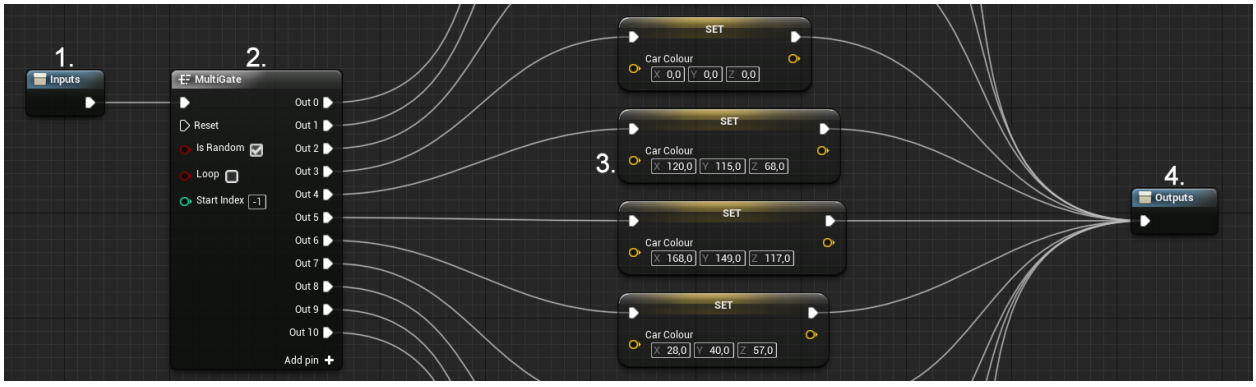


Figure 86. Procedural "Colour Chooser" macro closeup.

Figure 86 shows how this blueprint functions, which becomes apparent when taking a closer look at the macro. The "Input" node (1) launches the script with the "Multi Gate" node (2) randomly choosing what colour to pick. The various car colours are stored as vector values, which correspond with predetermined RGB colours. Several "Set" nodes (3) save

theses vector values into the “Car Colour” variable. The “Outputs” node constitutes the end of the macro.

Appendix 8: Level Scripting – Procedural Car Placement

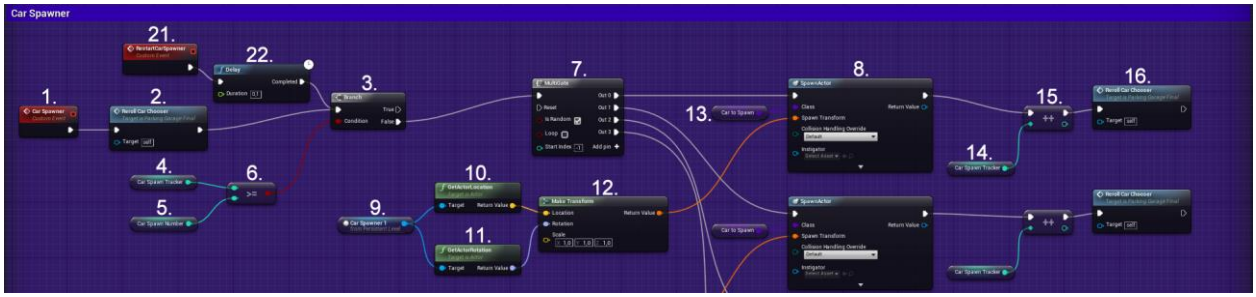


Figure 87. Procedural car placement blueprint closeup.

Figure 87 shows a close-up of this blueprint. The “Car Spawner” node (1) is the custom event node that launches as soon as the level starts up. This node immediately plays the “Reroll Car Chooser” node, which is another custom event script that randomly chooses between varieties of procedural car types. After a random car is selected, a “Branch” node (3) conducts a test. The system allows a predetermined number of cars to be spawned every time the level loads. The “Car Spawn Tracker” and “Car Spawn Number” variables track whether this upper limit has been reached or not. The “Car Spawn Tracker” variable starts with a value of zero, while the “Car Spawn Number” variable contains the number of cars needed in the level. Node “6” determines whether the “Car Spawn Tracker” value is equal to the “Car Spawn Number” value, which makes it possible for the “Branch” node (3) to either test true or false. When true, the car spawn system stops, while a false value reruns the spawn script.

A “Multi Gate” node (7) is the first script in the spawn system. It randomly selects between the empty actors that are placed within the level. The “Car Spawner 1” node (9) is an example of an empty actor that was referenced. Node (10) and (11) retrieves the location and rotation of this empty actor, which is then fed into a “Make Transform” node (12). This transform node can then be fed into the “Spawn Actor” node (8), which makes use of the “Car to Spawn” node to determine which car type to spawn here. After a car has been placed, the “Car Spawn Tracker” variable (14) is incremented by one using the “+ +” node (15). The custom “Reroll Car Chooser” event (16) is then executed to choose the next car type to spawn.

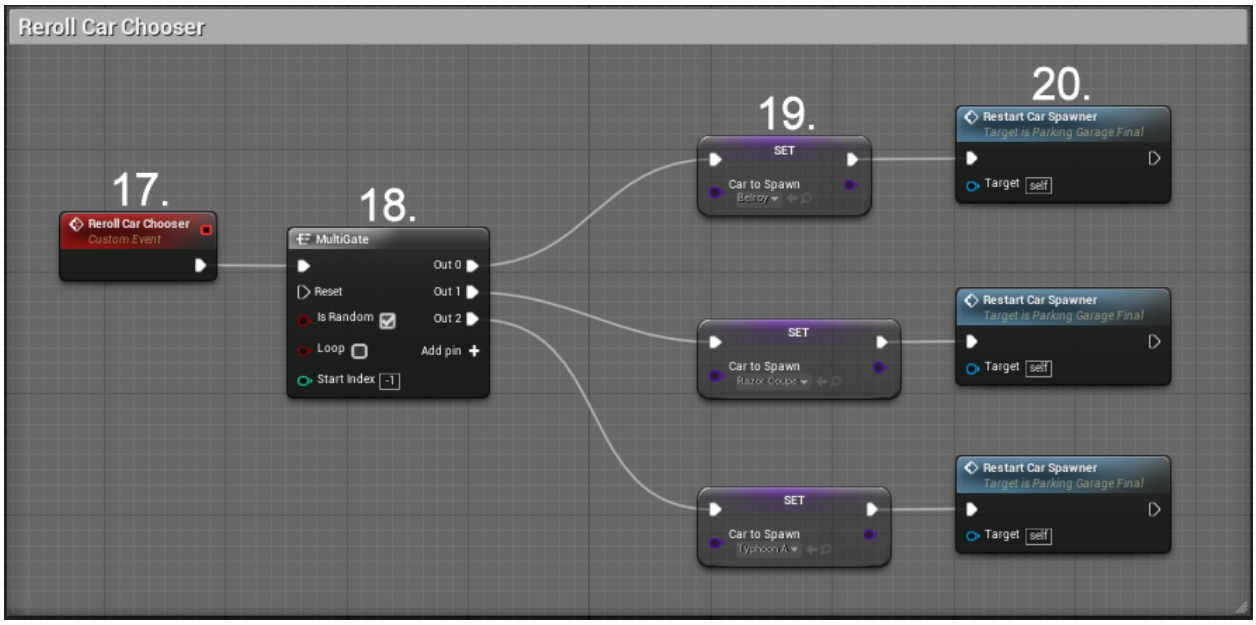


Figure 88. Random procedural car chooser blueprint.

Figure 88 depicts the “Reroll Car Chooser” script. Node (17) launches the script, with the “Multi Gate” node (18) randomly choosing between the various car types available. Only three car types are shown here to illustrate how the system works. The “Car to Spawn” variable is determined using a “Set” node (19), with the relevant car actor being selected in a drop-down on the node itself. After the car is selected, the custom “Restart Car Spawner”

event node (20) is executed. Executing this node takes the sequence of events back to the script shown in figure 87. Node (21) in figure 87 restarts the whole car spawning loop. A "Delay" node is added so that the system can keep up with the other elements that are procedurally spawned.

Appendix 9: Monster Camera Spawn System

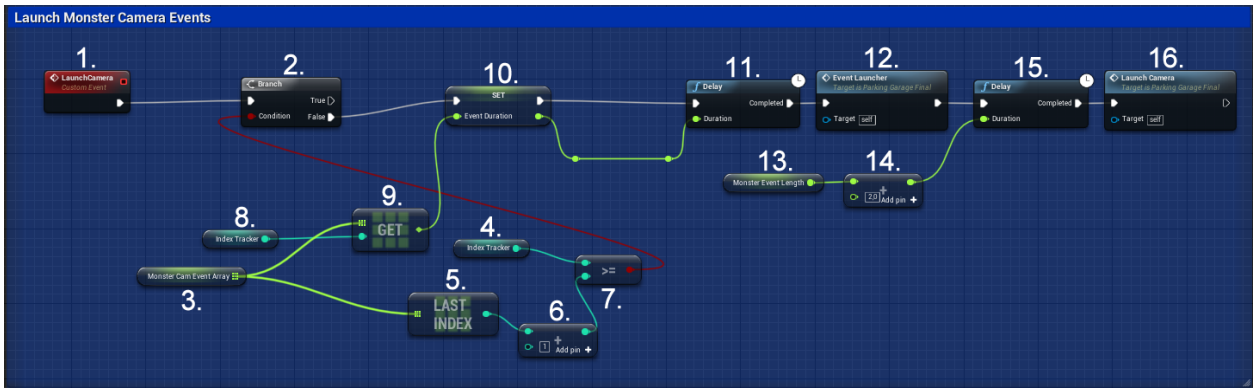


Figure 89. "Monster Camera" launch blueprint.

The "Launch Monster Camera Events" blueprint in figure 89 uses the event array that contains all of the "Monster Camera" even durations and uses those durations to spawn the required events.

The "LaunchCamera" node (1) launches the blueprint with a "Branch" node. This node tests whether the last index in the array has been reached or not. When the last index is reached, all the events have been concluded. The "Monster Cam Event Array" retrieves the array variable that stores all the event durations. Four things need to happen to track the progress within the array. First, the "Index Tracker" node (4), which starts at zero, is retrieved. Next, the "Last Index" node retrieves the integer value of the last entry of the array. If there are six entries, the last entry would be five, because the first entry of an array starts at

zero. Adding a "1" to the retrieved value using the "+" node (6) takes this zero-index into account. The "> =" node then compares the "Index Tracker" value (4) with the "Last Index" value (6) to check whether new events should be spawned or if the spawn system should stop.

When true, the "Branch" node (2) stops the system. When false, the value in the "Index Tracker" variable is used to extract the variable out of the array at that index point using the "Get" node (9). This value is then fed into the "Event Duration" variable using a "Set" node (10), which then drives the duration to the next "Monster Camera" event with the "Delay" node (11). After this delay, the "Event Launcher" function node (12) calls the custom script that launches the "Monster Camera" event. The next "Delay" node (15) prevents the system from rerolling until the "Monster Camera" event has concluded. Extra time is added to account for the fade effects by using the "Monster Event Length" variable (13) and adding two seconds to it using the "+" node (14). After the "Monster Camera" event concludes, the "Launch Camera" custom event node restarts the blueprint at node (1).

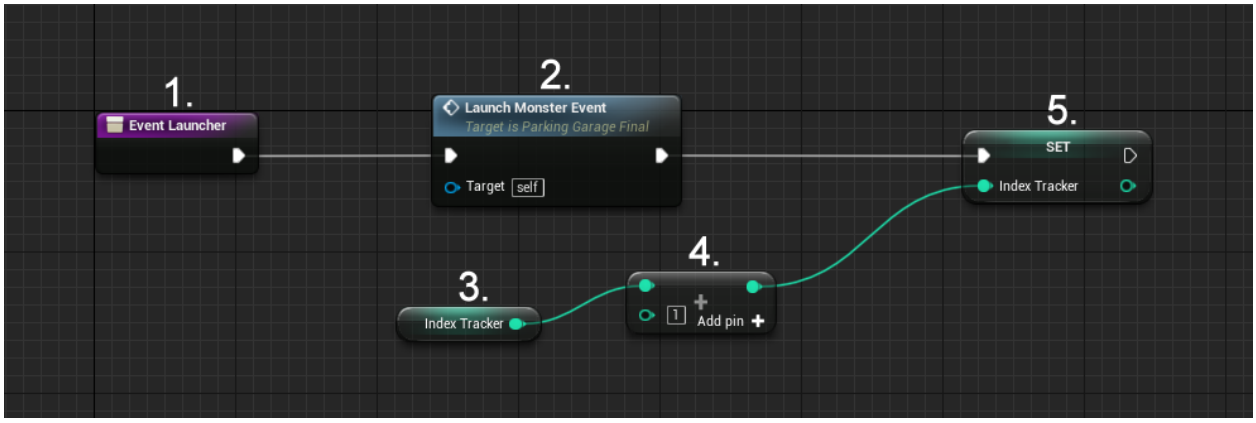


Figure 90. Monster camera "Event Launcher" function.

Figure 90 shows what the "Event Launcher" function looks like inside. Node (1) launches the function that initiates the "Launch Monster Event" blueprint with node (2). This function also increments the "Index Tracker" variable (3) by adding "1" to it using the "+"

node (4) and feeding it back into the “Index Tracker” variable. If, for example, the original value was one, it would now be two after this operation.

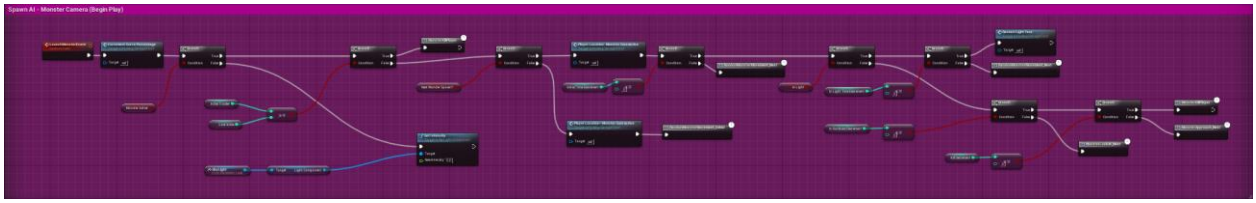


Figure 91. Monster camera “Spawn AI” blueprint.

Figure 91 shows the extent of the blueprint that is responsible for spawning the correct “Monster Camera” events, which the function above executes. Each time a “Monster Camera” event is requested, this script runs through several tests to determine an adequate event to spawn.

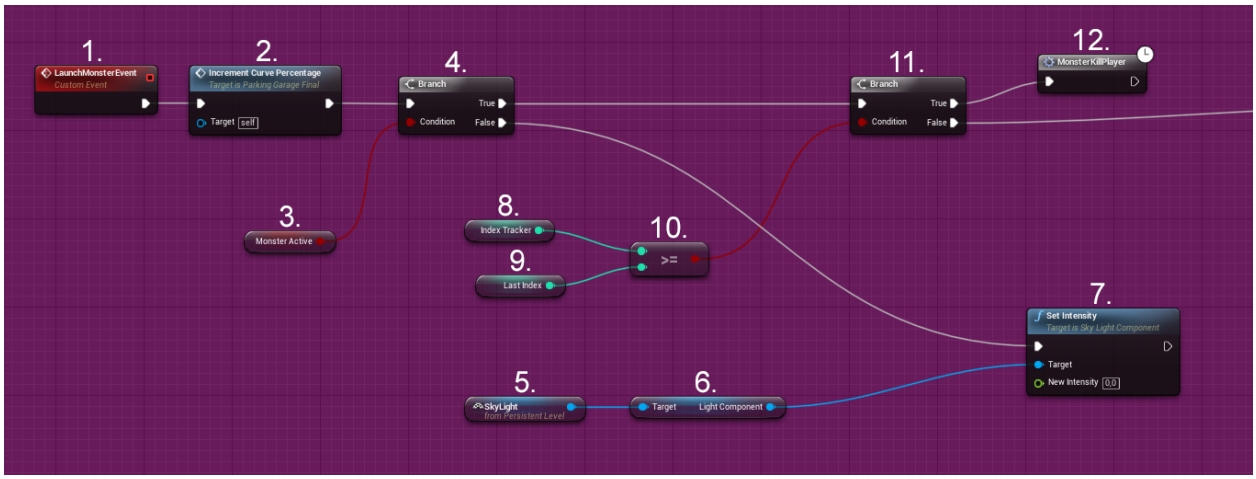


Figure 92. Monster camera “Spawn AI” blueprint closeup - part 1.

Figure 92 shows the first section of this blueprint. The “Launch Monster Event” node (1) is a custom event node that requests a “Monster Camera” event to spawn. The “Monster Active” Boolean node (3) is a variable that was set in another script and determines whether the “Branch” node (4) is true or false. When false, the intensity of the skylight is set to “0.0”. This script resets the luminance variable that was changed or will be changed elsewhere in

this script. The “Sky Light” node (5) references the skylight actor in the level. A “skylight” actor is used to illuminate the level whenever the “Monster Camera” is active. This light was added to make it possible for the monster to see in the dark. The “skylight” actor was the ideal choice due to its ability to illuminate a space evenly without casting shadows. The “Target” node (6) extracts the light component for use with the “Set Intensity” node (7), which resets the light intensity to zero.

The next “Branch” node (11) requires the system to check whether the system has reached the last index in the array, which will execute the “Monster Kill Player” macro (12). It also allows for the “Monster Camera” system to disengage when no events exist in the array. This ability makes it possible to provide for control groups in experiments when no events are loaded into the system.

The “Index Tracker” node (8) returns the current node in the array as an integer value. This value is required when comparing it with the “Last Index” node (9), which contains the integer value of the last index in the array. An example would be the number “6” if six events were registered into the array via the “Event Setup” menu. Node (10), which is plugged in the “Branch” node (11), determines if the value of the “Index Tracker” node is larger than or equal to the value of the “Last Index” node. When true, the “Monster Kill Player” macro (12) is launched. When false, the node executes the next set of nodes in the blueprint sequence.

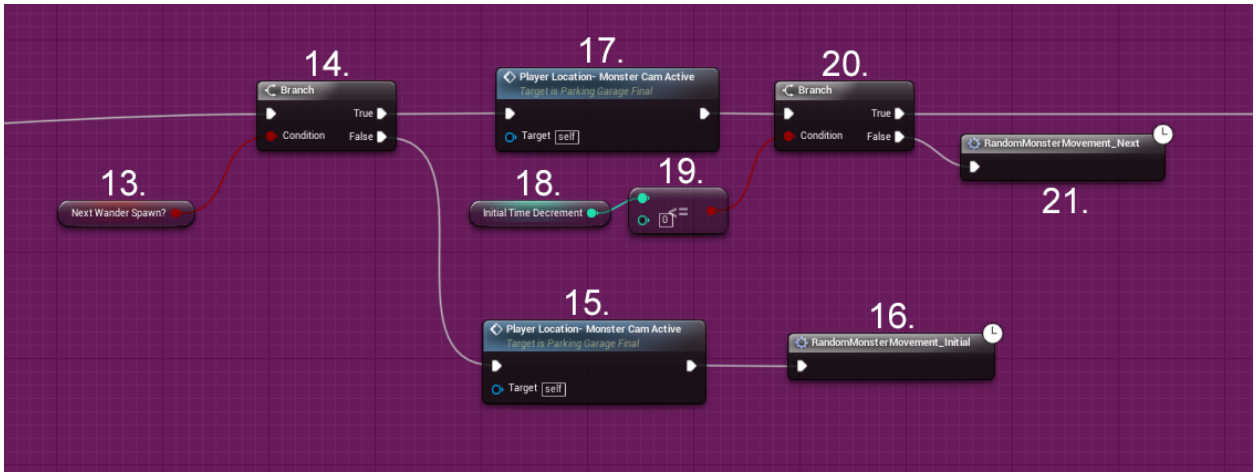


Figure 93. Monster camera "Spawn AI" blueprint closeup - part 2.

In Figure 93, the "Next Wander Spawn?" node (13), contains a Boolean value that determines whether the very first monster spawn has happened yet or not. The first spawn location will always remain the same. Every spawn after this initial spawn will start at a location near where the previous event ended. Adding this spawn point will prevent the monster from teleporting around the level too much. If the "Branch" node (14) tests false, the "Player Location – Monster Cam Active" node (15) will fire. It is a custom event node, which launches a self-contained function.

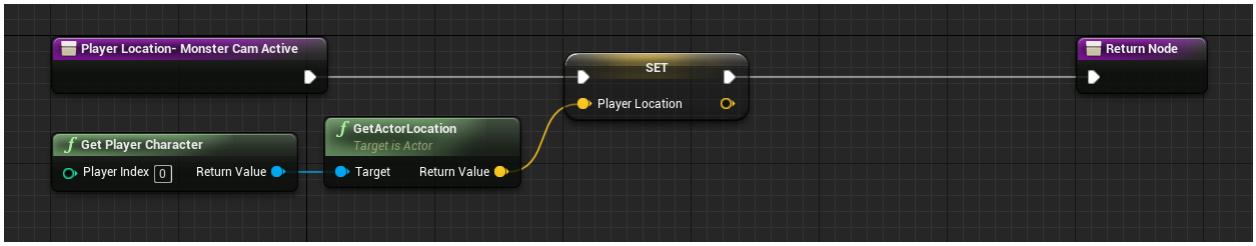


Figure 94. "Player Location – Monster Cam Active" function.

Figure 94 shows what the "Player Location – Monster Cam Active" function does. It is a simple function as it retrieves the location of the player and feeds it into the "Player Location" variable using a "Set" node.

Returning to the script featured in figure 93, the next node that fires is the "RandomMonsterMovement_Initial" node (16), which is one of the primary "Monster Camera" spawning macros. The script within this macro launches a camera event that starts at a fixed location in the level. If the "Branch" node (14) tests true, the custom "Player Location – Monster Cam Active" function node (17) launches again.

The next node in the sequence is another "Branch" node (20). This node makes use of the integer value in the "Initial Time Decrement" and uses node (19) to test whether it is smaller than or equal to zero. Timer calculations in UE4 are inaccurate due to the operation being linked to framerate, so making use of a "< =" node is preferable to using a "= =" node only. If the "Branch" node (20) tests false, a "RandomMonsterMovement_Next" macro node (21) is launched. This macro is nearly the same as the "RandomMonsterMovement_Initial" node (16). The only difference is that it captures the last known location of the monster and uses this location to determine the next starting point. The next set of nodes in the blueprint is executed when the "Branch" node (21) tests true. They will be explored below.

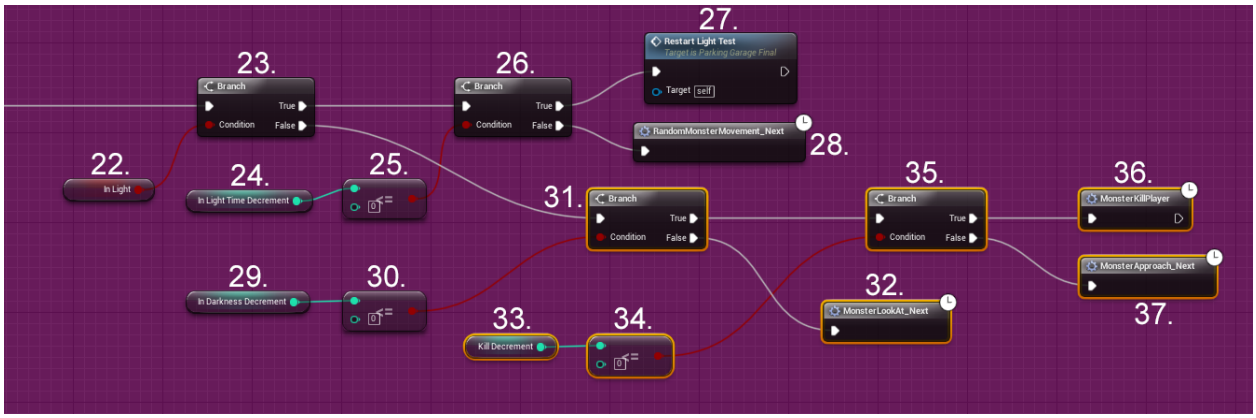


Figure 95. Monster camera "Spawn AI" blueprint closeup - part 3.

In figure 95, a “Branch” node (23) is the very next node that fires. It makes use of the “In Light” variable node (22) to determine what script sequence the “Branch” node fires. The “In Light” node (22) essentially tells the branch node if the player is located in an illuminated area or not. When true another “Branch” node (26) test is executed, this test makes use of the “In Light Time Decrement” variable node (24) to test whether the value is zero using the “<=” node (26). This test checks to see if the timer that tracks the player within a light source has lapsed or not. When false, another “RandomMonsterMovement_Next” macro node (28) is launched. When true, a “Restart Light Test” custom event node (27) is launched. The “Restart Light Test” function contains additional code that determines what happens when the player spends too much time in either brightly lit areas or darkened areas and will be explored below.

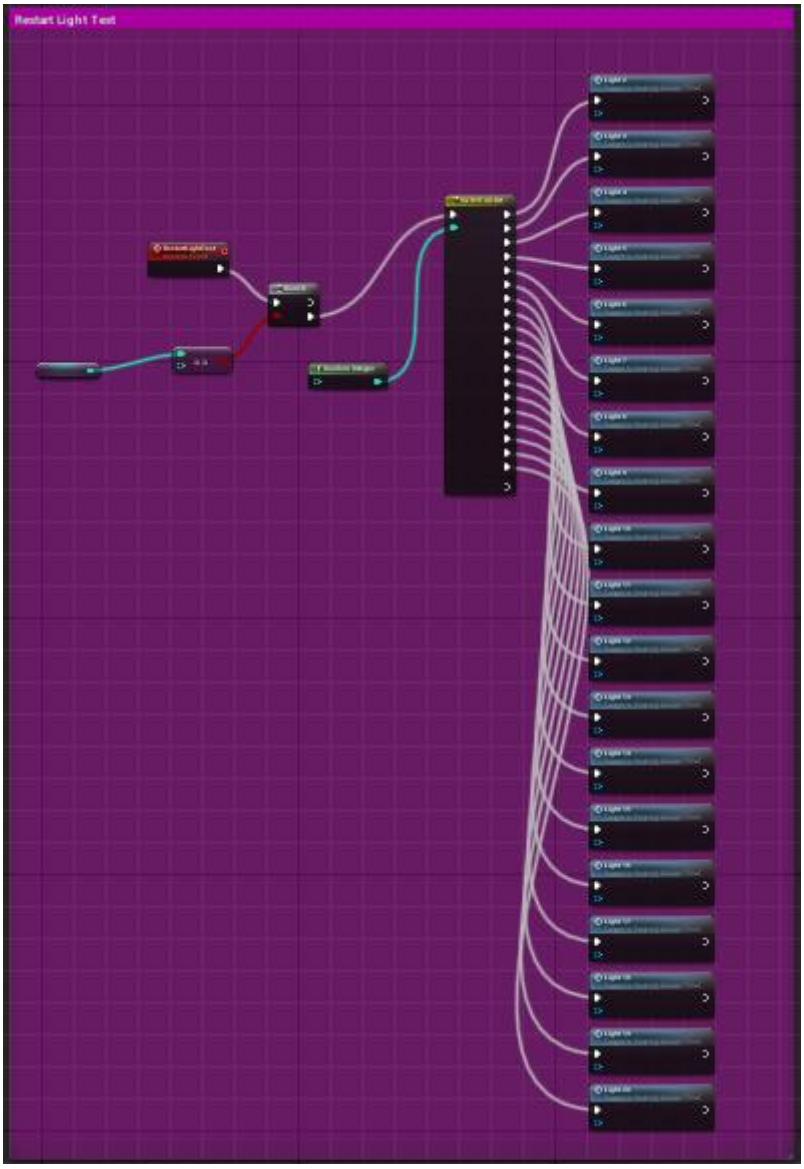


Figure 96. "Restart light test" blueprint.

Figure 96 shows the "Restart Light Test" script. It can push instructions to a light that is referenced within the parking structure and potentially switch it off. Every light within the main play area can be switched off.

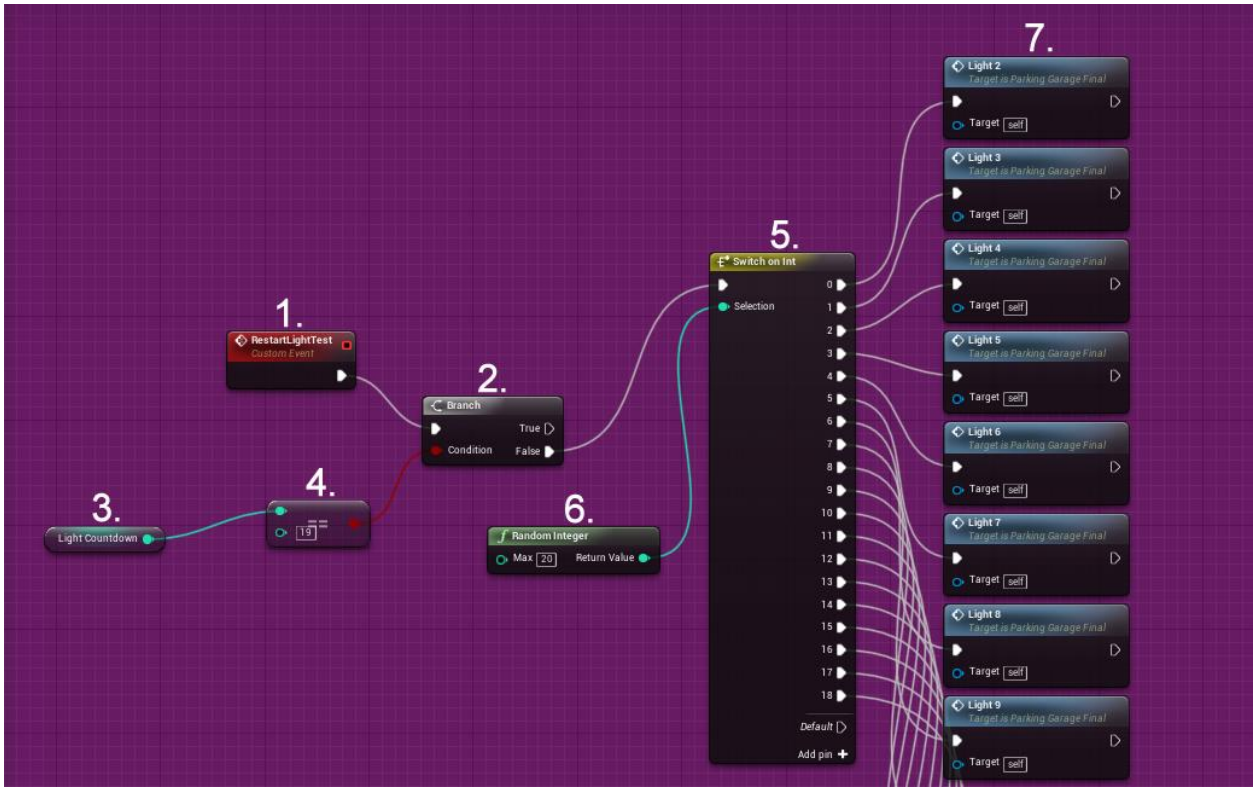


Figure 97. "Restart light test" blueprint closeup.

Figure 97 shows a detailed view of the blueprint. The "Restart Light Test" node (1) launches the blueprint. A "Branch" node (2) is executed next. It retrieves a reference of the "Light Countdown" variable (3), which has a default value of zero, and checks whether the value is equal to "19" with the "= =" node (4) each time this blueprint is executed. When true, the system will be unable to disable lights anymore, as everything is already off by then. The "Switch on Int" node (5) choose the what custom event (7) to execute based on the random integer value that was fed into it by the "Random Integer" node (6).

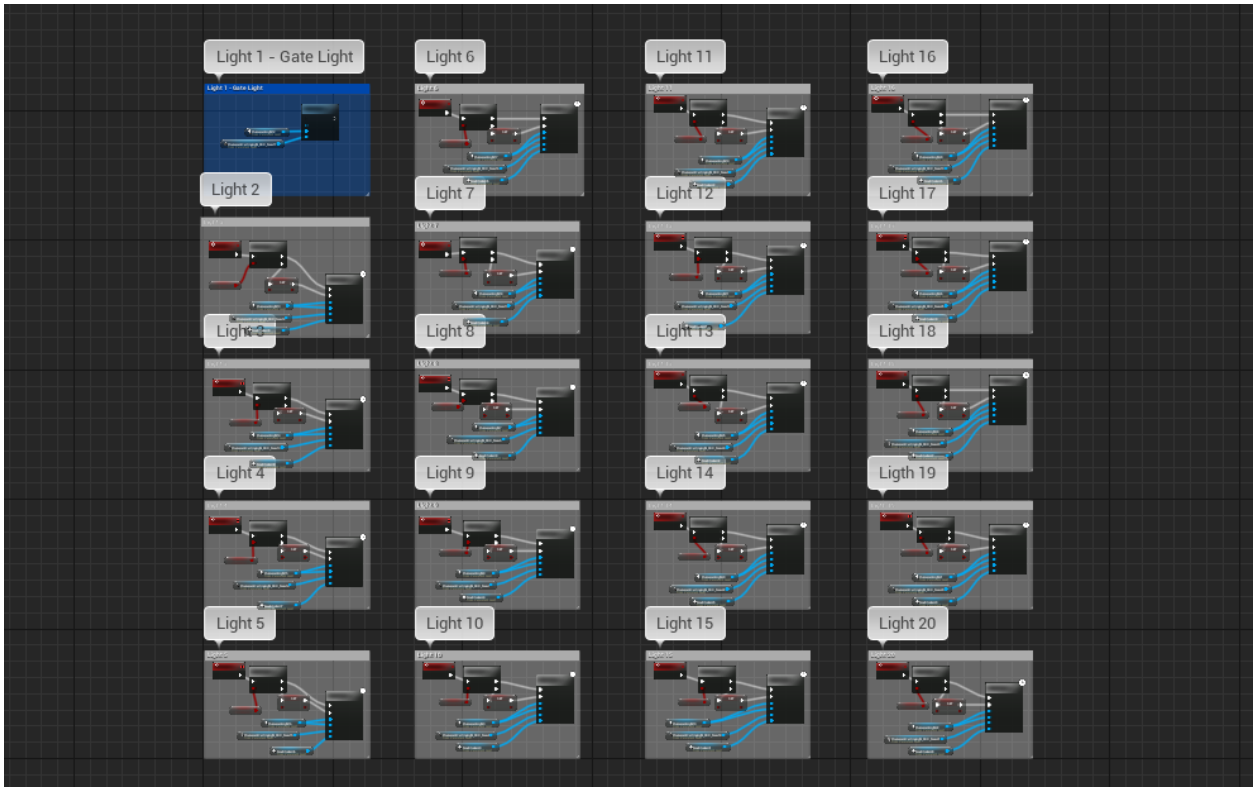


Figure 98. Light selection blueprints.

When one of the custom "Light" events is chosen, one of the twenty blueprint scripts, depicted in figure 98, is executed.

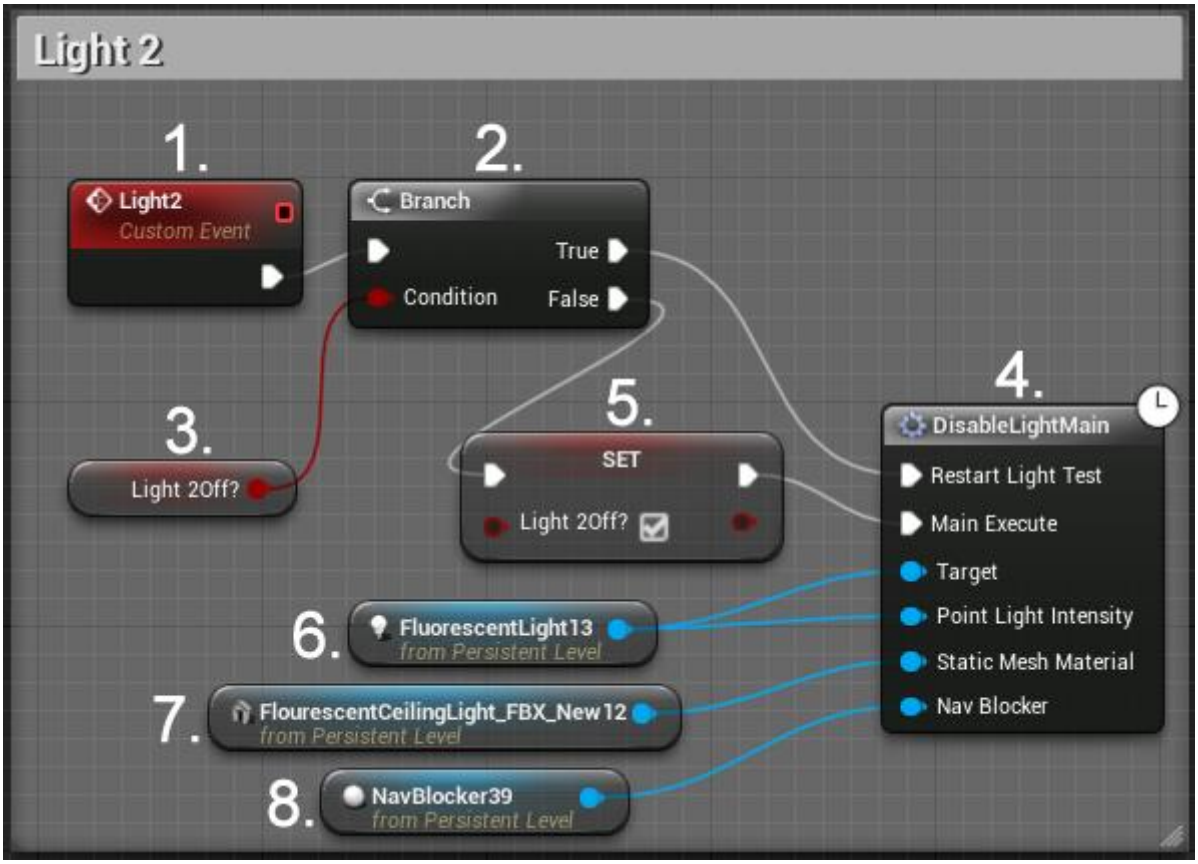


Figure 99. Light selection blueprint closeup.

Figure 99 shows a close-up of one of these "Light" blueprints. The "Light 2" node (1) launches the script. A "Branch" node (2) chooses whether to restart the light test or let the monster disable one of the lights in the parking structure using the "Light 2 Off" variable (3). When true, the "Disable Light Main" macro executes using the "Restart Light Test" input. When false, the "Light 2 Off" variable is enabled so that the system knows already disabled and rerolls the light picking scripts.

A reference of the tube light actor (6), the tube light mesh (7), and the "Nav Blocker" (8) for that light is also plugged into the "Disable Light Main" macro (4).

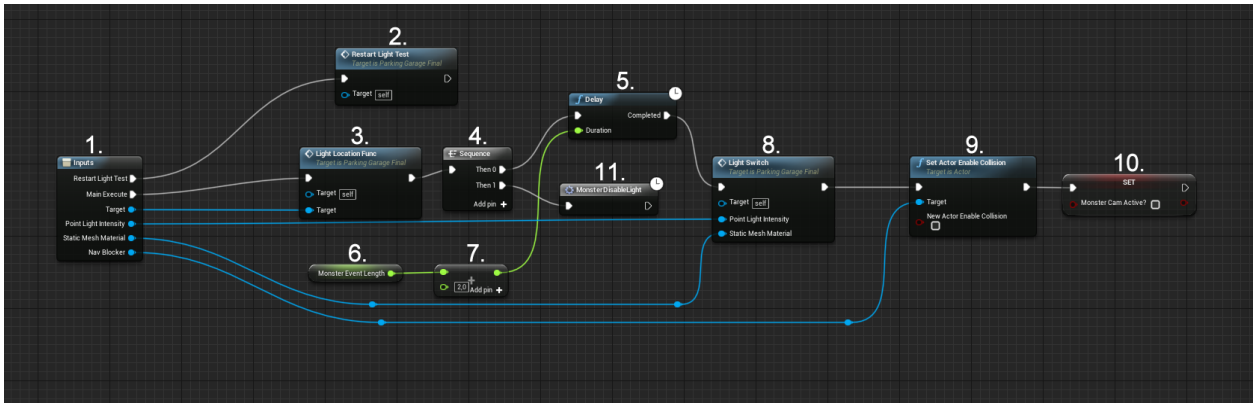


Figure 100. "Disable Light Main" macro.

Figure 100 shows the interior of the "Disable Light Main" macro. The "Inputs" node (1) takes the nodes from the "Light 2" blueprint above and feeds them into additional nodes. The "Restart Light Test" node (2) triggers when the system detects that this light is already disabled and re-rolls the light picking scripts. When the light is still active, the "Light Location Func" node (3) executes. This function retrieves the location of the light source based on the mesh reference that was plugged into the macro and will be explored below. The "Sequence" node (4) plays the "Monster Camera" event macro (11) that shows the antagonist moving to the light source in question, while the "Delay" node (5) syncs the event with the disabling of the light source when the "Monster Camera" event ends. The "Monster Event Length" node (6) retrieves the duration of the monster event, while the "+" node (7) adds two seconds to this duration to factor in the "Monster Camera" screen fades.

The "Light Switch" function node (8) plays additional code that disables the point light actor. The "Set Actor Enable Collision" node (9) disables the "Nav Blocker" for that light source so that the antagonist can get close to it. The "Monster Cam Active?" variable is then set to false with a "Set" node (10). This variable is used by the "Distance Between Points" blueprint, which determines the distance between the player's location and another

point specified. This function prevents the monster from moving too close to the player in "Monster Camera" macros.

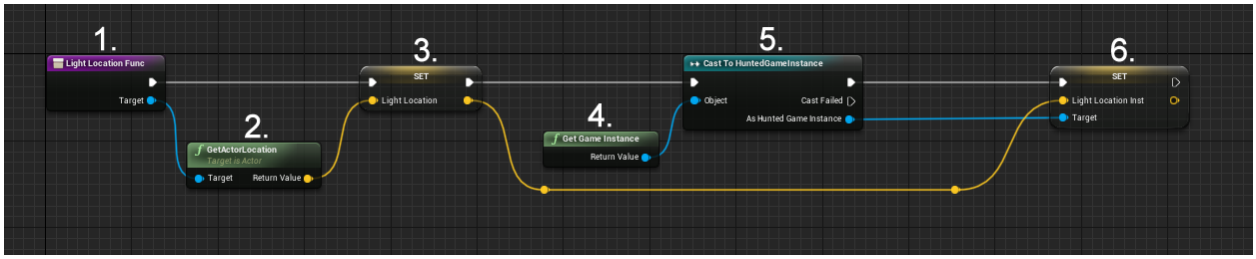


Figure 101. "Light Location Func" function.

Figure 101 shows the "Light Location Func" function. It makes use of the "Get Actor Location" node (2) to retrieve the location of the light source and feeds it into the "Light Location" variable using the "Set" node (3). Nodes (4) and (5) then retrieves and readies the game instance actor and feeds the "Light Location" value into the "Light Location Inst" variable using another "Set" node (6). This action copies this value into a similar variable in the game instance so that it can be used elsewhere.

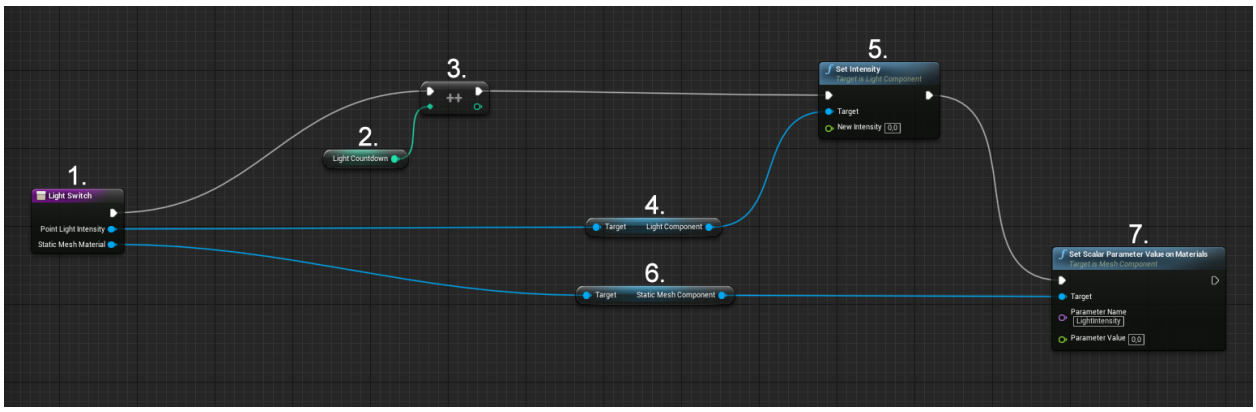


Figure 102. "Light Switch" function.

Figure 102 depicts the "Light Switch" function. The "Input" node (2) executes the "++" node (3), which increments the "Light Countdown" variable (2) by one. Next, a reference of the point light actor (4) is fed into a "Set Intensity" node, so that the light's "Intensity"

value can be set to zero. After the light source is switched off, a reference of the fluorescent light mesh (6) is plugged into the “Set Scalar Parameter Value on Material” node (7). Plugging the static mesh into this node will allow it to access a material parameter called “Light Intensity” so that the emissive value of the fluorescent tube part of the mesh can be set to zero, disabling the corresponding glow effect.

Looking back at the script progression in figure 95 provides an opportunity to see what happens when “Branch” node (23) tests false. A separate script checks whether the player has spent too much time in a darkened area. Another “Branch” node (31) facilitates this by checking the value of the darkness timer. When the integer value of the “In Darkness Decrement” node (29) is tested with the “< =” node (30), a true or false value will determine what set of nodes “Branch” node (31) will execute. When false, the “MonsterLookAt_Next” macro will launch. The “_Next” suffix is used in conjunction with some of the macros. This suffix essentially specifies that the macro makes use of the last known location of the monster to determine the spawn location of the next “Monster Camera” event.

If the “Branch” node (31) tests false, the “MonsterLookAt_Next” macro is launched. This macro is like the “RandomMonsterMovement_Next” macro, in that the monster takes a random route through the parking structure. The only difference is that it looks at the player’s location while walking around.

When the “Branch” node (31) tests true, another “Branch” node (35) is executed. This node (35) tests in the same way as the previous node (31). The only difference is that it makes use of the value in the “Kill Decrement” node (33) to determine if the kill timer has run out. The kill timer initiates the moment the “In Darkness” timer value reaches zero. Players have to make their way back to the closest light source before the kill timer runs out.

If the “Branch” node (35) test returns “true” the “MonsterKillPlayer” macro will launch, which means that the “Monster Camera” event that kills the player will spawn soon. The “MonsterApproach_Next” macro will play, should the “Branch” node (35) test return a “false” value. This macro merely shows the monster approaching the protagonist, which should prompt the player to reach a light source as soon as possible.

Appendix 10: Monster Camera Events – Monster Cam Seek

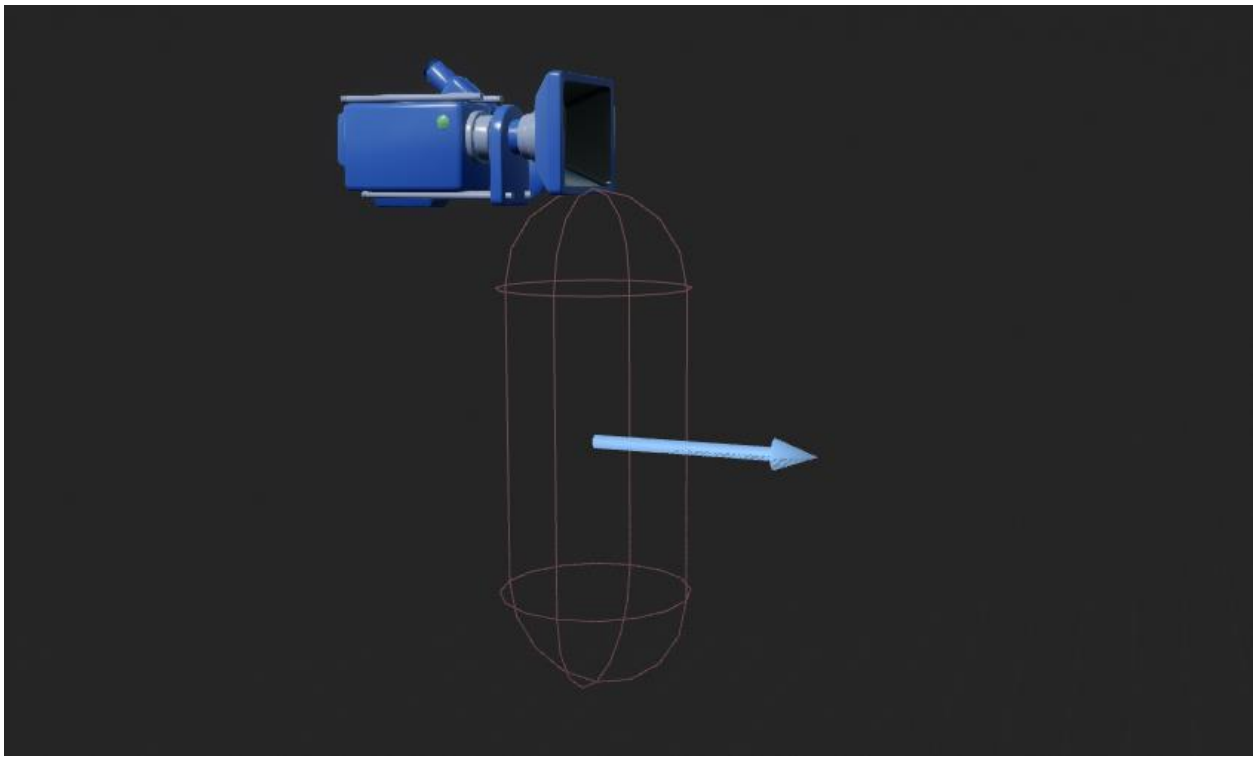


Figure 103. “Character Pawn” used by antagonist.

Figure 103 shows the collision capsule component of the “Character Pawn”. The arrow shows the direction of travel, while the camera object orients the perspective relative to this direction.

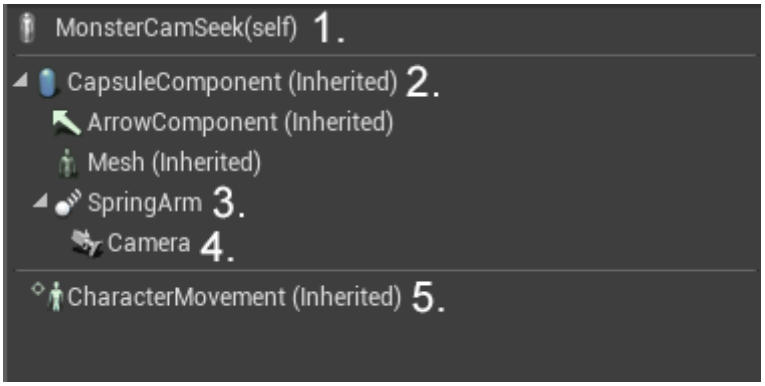


Figure 104. "Character Pawn" blueprint hierarchy.

Figure 104 shows the hierarchy of a "Character Pawn" called "Monster Cam Seek". This actor is used when the monster is aimlessly walking through the level while looking straight ahead. "Monster Cam Seek" (1) references the entire entity, while the "Capsule Component" references the collision capsule contained within the actor. The "Spring Arm" component (3), acts in the same way as a camera boom used in cinematography and is essentially an installation that controls the movement of the camera itself. The "Camera" component (4) represents the viewpoint of the monster. It allows the perspective to shift from that of the protagonist to the perspective of the monster. The "Character Movement" component contains properties such as "speed", "acceleration", "turn-rate", and others.

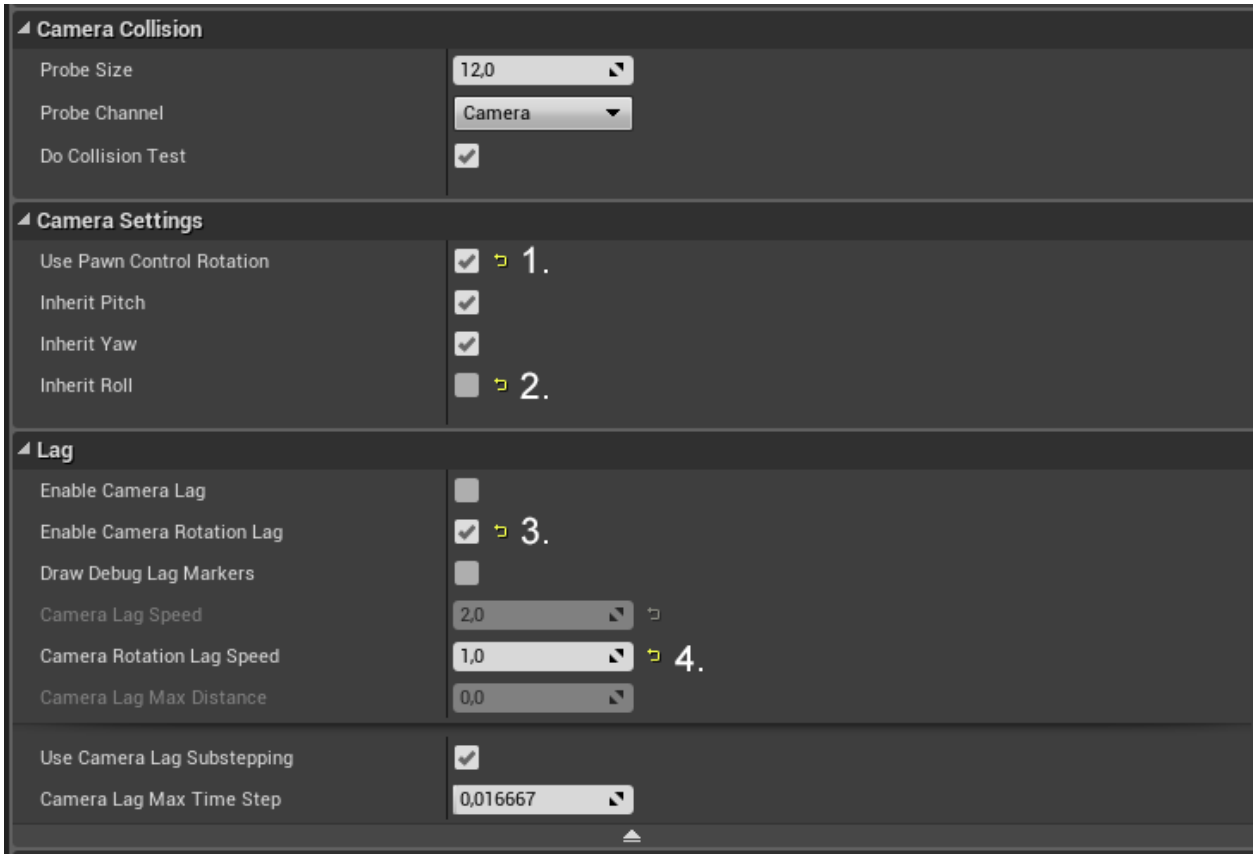


Figure 105. "Character Pawn" camera settings.

When the spring arm component is clicked, a whole host of properties that can be adjusted are made available. In figure 105, several settings were changed to allow the actor to look in the direction of travel. The first setting is the "Use Pawn Control Rotation" (1). The camera follows the rotation of the player and inherits the "Pitch" and "Yaw" as selected below the first property. The "Inherit Roll" setting (2) was disabled to prevent the camera from banking. Under the "Lag" category, the "Enable Camera Rotation Lag" setting (3) was enabled. This setting adds some lag to the camera, to prevent any abrupt movement of the camera that might seem too jarring for a living creature. The "Camera Rotation Lag Speed" setting provides an amount of time for the camera to speed up or come to a complete stop during the lag operation.



Figure 106. "Character Pawn" camera "Spring Arm" length.

Figure 106 shows one of the "Camera" settings that were changed when the "Spring Arm" component was clicked. The "Target Arm Length" component represents the length of the "Spring Arm" component. The value was set to zero because a first-person perspective is used for the "Monster Camera" actor. Larger values are usually used when a third perspective of the "Character Pawn" is required.



Figure 107. "Character Pawn" movement settings.

Figure 107 shows the properties that are available when the "Character Movement" component is clicked in the character hierarchy. Only one setting was changed here, however. The "Braking Friction" (1) was changed to "0.1", while the "Use Separate Braking Friction" setting was enabled. This setting works in the same way as the "Camera Rotation Lag" setting in figure 105. When the actor reaches the end of its route, it evenly reduces its speed

until it comes to a complete stop. It makes the actor's speed changes less abrupt when moving through the level.

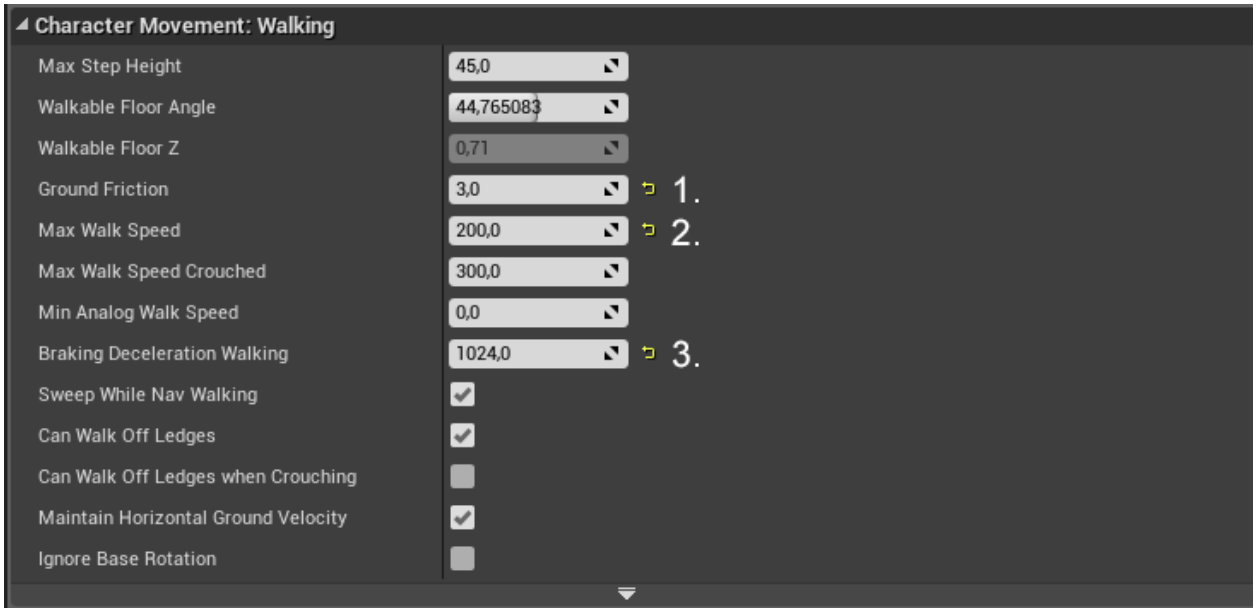


Figure 108. "Character Pawn" walking movement settings.

Figure 108 shows the "Character Movement: Walking" setting of the "Character Movement" component. The "Ground Friction" setting (1) is increased and works in tandem with "Braking Friction" and "Use Separate Braking Friction" setting shown in figure 107 to ramp acceleration changes evenly. The "Max Walk Speed" setting (2) is self-explanatory in that it sets the maximum possible speed of the actor. The "Braking Deceleration Walking" setting (3) sets the deceleration rate of the actor when slowing down.



Figure 109. "Character Pawn" rotation settings.

Figure 109 shows the "Character Movement (Rotation Settings)" roll-out. The "Orient Rotation to Movement" setting is enabled so that the capsule collider rotates to face the direction of movement automatically.



Figure 110. "Character Pawn" navigation mesh settings.

In the "Nav Movement" roll-out (figure 110), the "Fixed Path Braking Distance" setting (1) was set to two hundred. This setting enables the character to come to a complete stop within "200" unreal units. The "Use Fixed Braking Distance for Paths" setting (2) needs to be enabled for this braking function to work correctly. It is used in conjunction with the other settings to gradually slow the monster's speed as it nears its destination.

Appendix 11: Monster Camera Events – Monster Cam Look

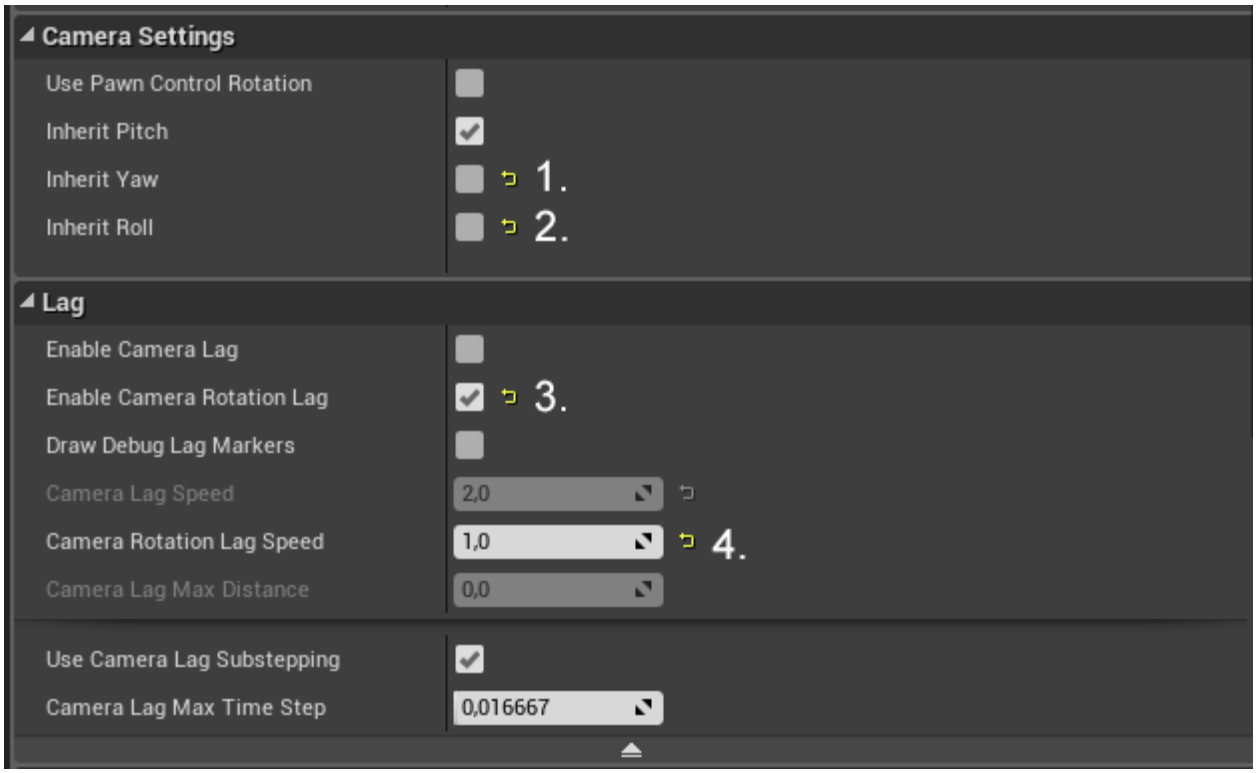


Figure 111. "Monster Cam Seek" camera settings.

Figure 111 shows the only way it differs from the "Monster Cam Seek" avatar. Both the "Inherit Yaw" setting (1) and "Inherit Roll" setting (2) was disabled to allow the camera to be orientated to face the protagonist. Similar to the previously mentioned avatar, the "Enable Camera Rotation Lag" setting was also enabled to provide gradual movement to the rotations of the camera. Enabling this setting makes the movement less abrupt and jarring.

Appendix 12: Monster Camera Events – Monster Cam Kill

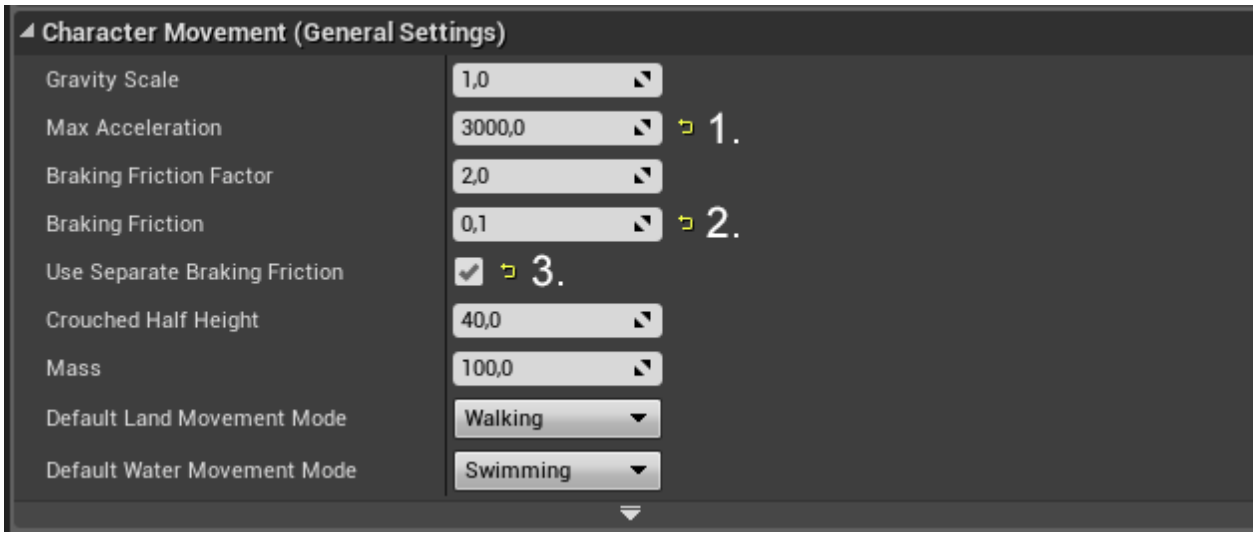


Figure 112. "Monster Cam Kill" movement settings.

Figure 112 shows the settings that were enabled within the "Character Movement" roll-out to enable the features mentioned above. The "Max Acceleration" setting (1) was set to "3000" unreal units. Setting it this amount allows the "Monster Cam Kill" avatar to accelerate to its run speed rapidly. The "Braking Friction" setting (2) was set to "0.1" to smooth out the deceleration and make it less abrupt. The "Use Separate Braking Friction" setting (3) had to be enabled, to make this possible.



Figure 113. "Monster Cam Kill" walking movement settings.

Figure 113 shows the settings that were changed within the "Character Movement: Walking" roll-out. The "Ground Friction" setting (1) was set to "3.0" and used in conjunction with the previously mentioned "Braking Friction" setting to slow down the "Monster Cam Kill" avatar. The "Max Walk Speed" setting is the top speed at which the avatar can travel. This value represents the higher run speed for this avatar. The "Braking Deceleration Walking" setting was set to "1024.0" to help the avatar decelerate quicker when moving at its top speed. It is used in conjunction with the "Ground Friction" and "Braking Friction" settings.

Appendix 13: Monster Camera Events – Head Bob

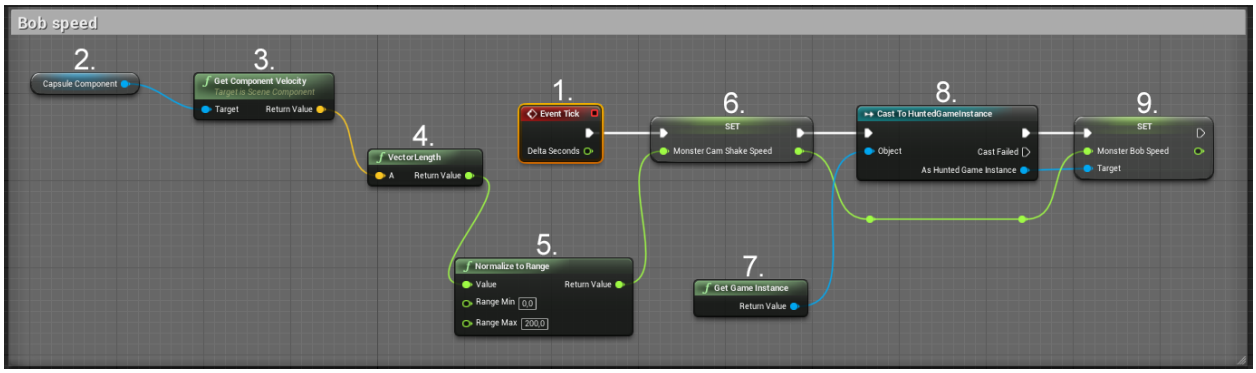


Figure 114. Head bob speed blueprint.

Figure 114 shows the code that samples the speed of the "Monster Camera" and feeds that value into the intensity of the "head bob" effect. The blueprint is embedded within the "Monster Cam Seek" avatar so that the constituent components can be sampled. The "Capsule Component" node (2) samples the physically simulated collision capsule that allows the monster to navigate through the level. The "Get Component Velocity" node (3) retrieves the velocity of the collision capsule while the monster is walking around. The "Vector Length" node (4) transforms the constantly updating vector nodes into a vector length, which is a single float value denoting the current speed of the collision capsule. The "Normalise to Range" node (5) takes this value and normalises it to fit between the two ranges specified. The minimum value will therefore always be "0", and the maximum value will never exceed "150". This normalised vector length is then fed into the "Monster Cam Shake Speed" variable node (6). This value denotes the speed at which the monster is currently moving. For this value to be used globally, it needs to be fed into the game instance. The "Get Game Instance" node retrieves the game instance, while the "Cast" node (8) accesses the game instance so that it can be used. The "Set" node (9) simultaneously retrieves

the “Monster Bob Speed” float variable out of the game instance and feeds the “Monster Cam Shake Speed” float variable into it.

Appendix 14: Monster Camera Macros – Movement Initial

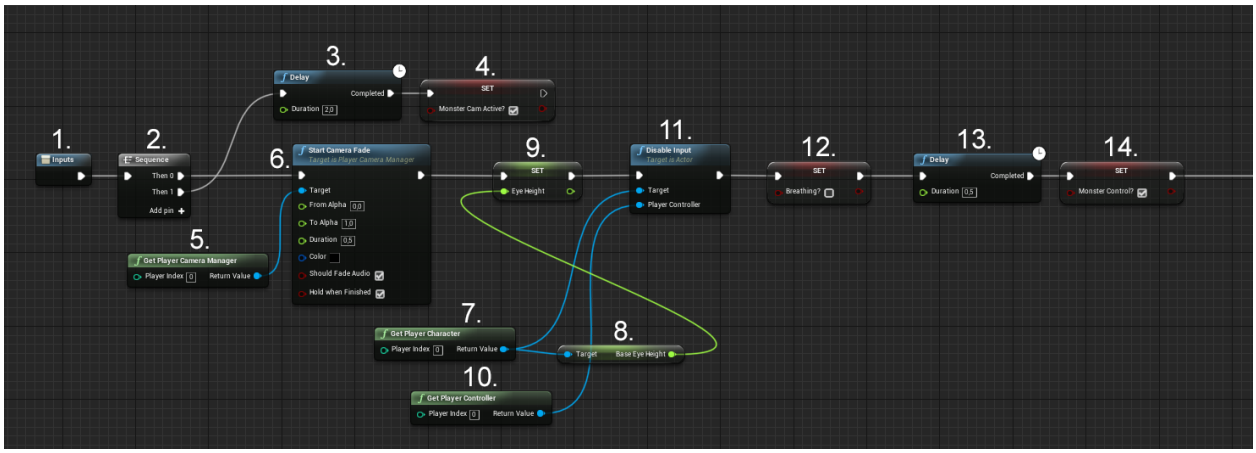


Figure 115. “Monster Camera” macro closeup – part 1.

Figure 115 shows the first part of this macro. The “Inputs” node (1) is where the other external scripts plug into the macro to activate it. The “Sequence” node (2) activates two pieces of code simultaneously. The “Delay” node provides the “Start Camera Fade” node with an adequate amount of time to perform a full fade effect before the “Set” node (4) sets the “Monster Cam Active?” Boolean value to true. The “Get Player Camera Manager” node references the player’s camera to fade the player’s perspective to a solid black colour using the “Start Camera Fade” node (6). Right after the player’s camera completes its fade to black, the player’s eye height is saved for later use. The “Get Player Character” node (7) references the player character that enables the “Target” node (8) to extract the “Base Eye Height” of the player character. This value is then fed into the “Eye Height” variable using the “Set” node (9).

While this operation takes place, the player's controls need to be disabled while the "Monster Camera" is active. The "Get Player Controller" node (10) retrieves the controller used by the player and deactivates it using the "Disable Input" node (11). The "Get Player Character" node (7) is also required as an input by the "Disable Input" node (11) for this function to work. After the controller is disabled, a Boolean variable called "Breathing?" is set to false using a "Set" node (12). This disables the breathing sound that is audible whenever the player is in control. After a "Delay" node (13) another "Set" node sets the "Monster Control?" Boolean variable node (14) to true. Setting this Boolean variable to true makes it possible for the system to check when the monster is controlled, enabling it to activate additional effects within this period.

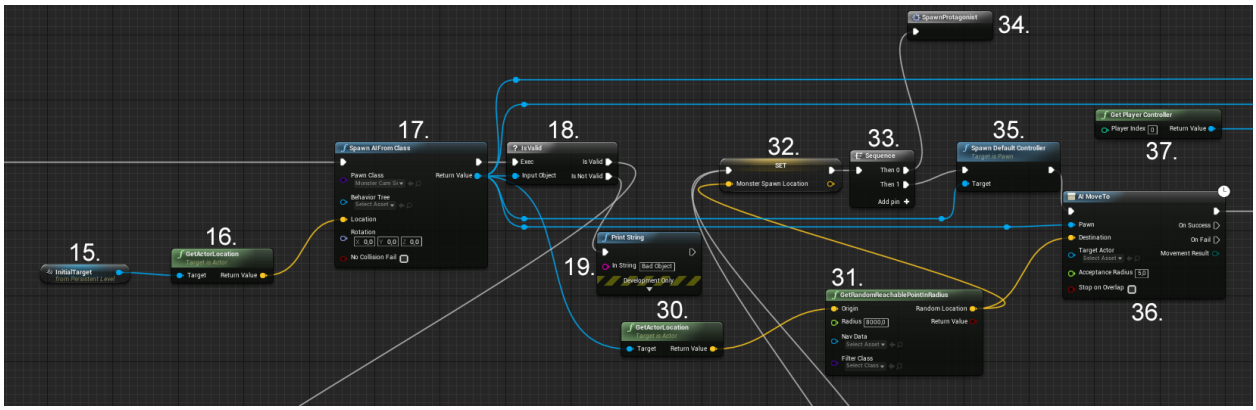


Figure 116. "Monster Camera" macro closeup – part 2.

Figure 116 shows the next part of the macro that spawns the correct avatar in the level. The "Initial Target" node (15) extracts the empty actor node that was placed within the level. This actor will act as the fixed spawn location for the first event. The "Get Actor Location" node (16) extracts the location of this empty actor that then plugs into the "Location" slot on the "Spawn AI from Class" node (17). The "Monster Cam Seek" avatar is selected in the "Pawn Class" drop-down. This avatar will, therefore, be spawned when this node is executed. The "? Is Valid" node (18) checks whether the spawn is valid. If the spawn

is invalid, a message that states “Bad Object” is printed on the screen using the “Print String” node. Because a fixed location is used to spawn the avatar, it is highly unlikely that the spawn will be invalid, however. When valid, the next bit of code is executed. This code will be explored below before the rest of the nodes in figure 116 are explored.

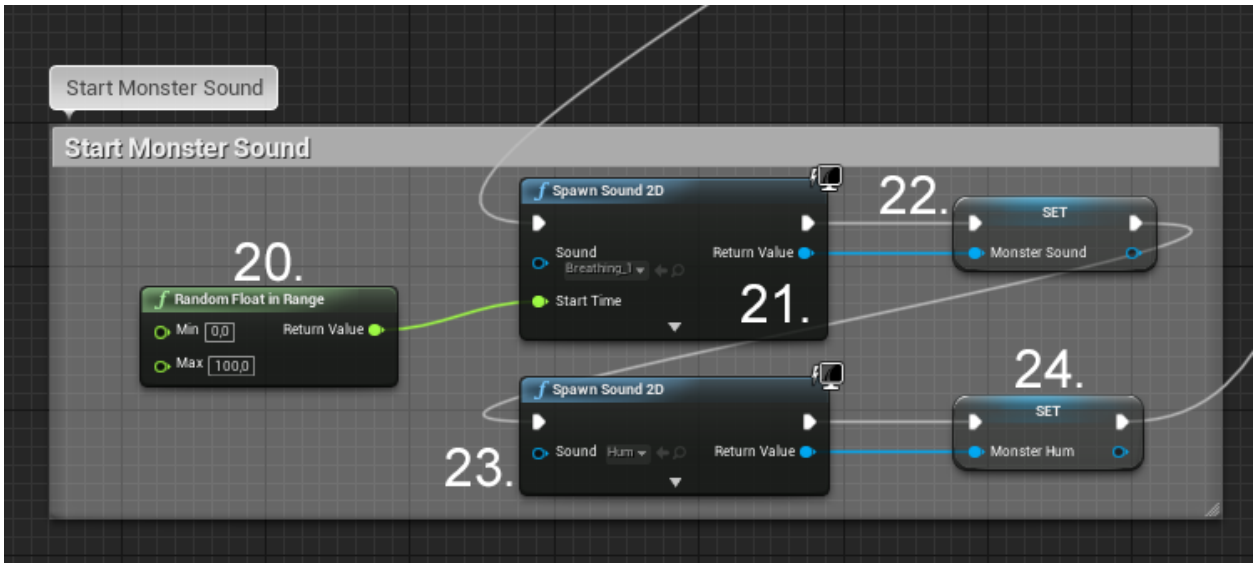


Figure 117. “Monster Camera” macro closeup – part 3.

Figure 117 shows the nodes that start playing the monster’s breathing sound. A “Random Float in Range” node (20) is used to determine a random float between “0.0” and “100.0”, which will be fed into the “Start Time” plug on the “Spawn Sound 2D” node (21). The random value allows the monster breathing sound, which is a looping sound, to start at a random point within the duration of the clip. This will reduce the repetition of the clip, every time a “Monster Camera” event is launched. This sound is fed into a variable called “Monster Sound” using the “Set” node (22). Feeding the sound object into a variable makes it easier to use elsewhere. A subtle humming sound is played on top of the monster’s breathing sound by using another “Spawn Sound 2D” node (23). This sound is also fed into a unique variable called “Monster hum” using the “Set” node.

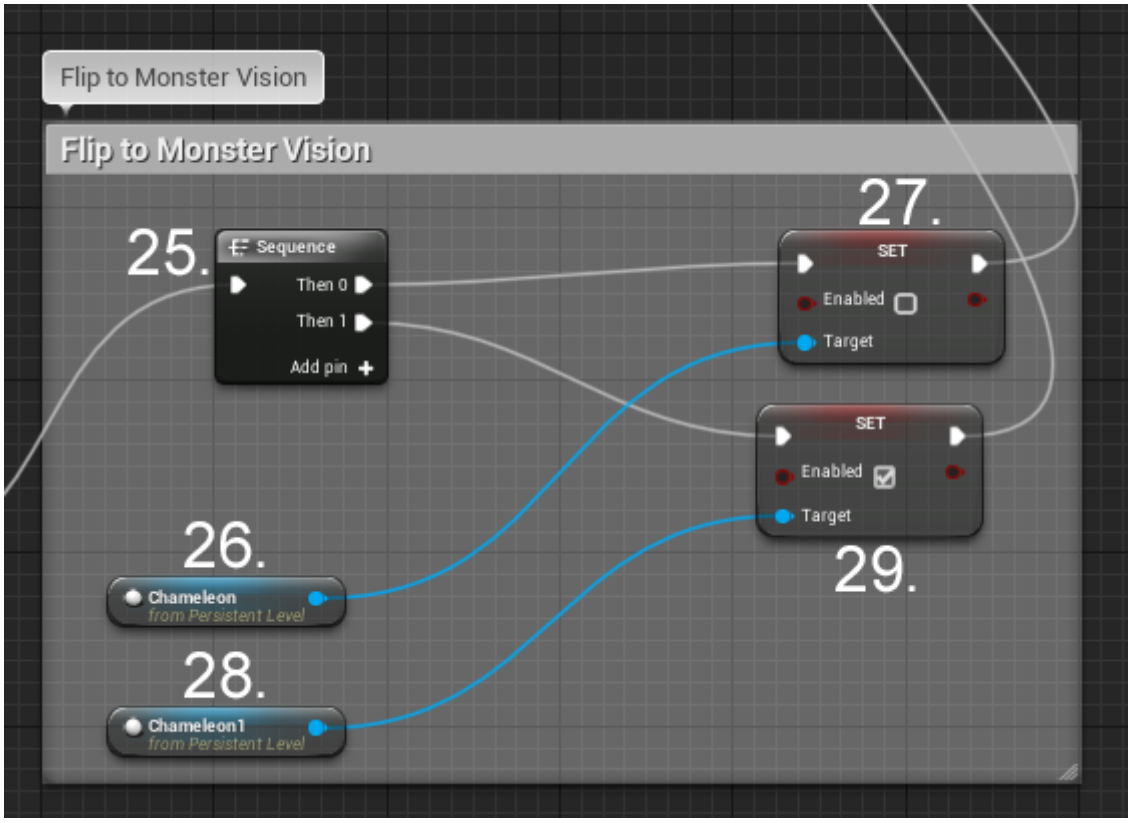


Figure 118. "Monster Camera" macro closeup – part 4.

Figure 118 is the next part of the macro that handles the monster vision component. A post-processing effect pack called "Chameleon" was used to achieve the "Monster Vision" effect, as seen in figure 23. The effect pack can be enabled or disabled as needed. The pack does not work locally and affects all cameras equally, so the required effects need to be enabled or disabled when needed. Two copies of the chameleon effects pack were placed within the level. The first one represents the vision of the protagonist, while the second one represents the vision of the monster. The "Sequence" node (25) simultaneously activates the first pack while deactivating the other one. The "Chameleon" node (26) references the first effects pack placed within the level, while the "Chameleon1" node (28) references the other one. A "Set" node disables the "Chameleon" effects pack, while another "Set" node enables the duplicate.

After the “Monster Camera” post-processing effect was enabled the next node in the series is executed. This investigation will continue with node (32) in figure 116. The “Get Actor Location” node (30) retrieves a location reference of the “Monster Cam Seek” avatar that was spawned earlier on. It then takes this location and finds a random location within the bounds of the navigation mesh using the “Get Random Reachable Point In Radius” node (31). The radius for this node was set at “8000” unreal units. This radius prevents the system from selecting a position that is too close to the player. This value is then fed into a float variable called “Monster Spawn Location” using a “Set” node (32). This variable will be used by other macros to find the starting point for their “Monster Camera” events.

The next node is a “Sequence” node (34) that is used to execute two pieces of code simultaneously. The first script that is executed is a macro called “Spawn Protagonist” (34). This macro will be explored later. A “Spawn Default Controller” node is the next node that the sequence executes. The AI avatar requires that a controller be assigned to it before it can move. The benefit of using “Character Pawns” is that players can inhabit and control them. Controlling the monster avatars will not be possible in “Interloper”. Spawning a “Default Controller” will be required, nevertheless. The “Monster Camera” is essentially another player controller, but in this case, it is controlled by an AI script. The next node in the sequence is the “AI Move To” node (36). This node takes the avatar and moves it to the destination specified. The random location found by node (31) will be connected into the “Destination” plugin node (36). The “Get Player Controller” node (37) will be used in the next set of nodes specified in figure 124.

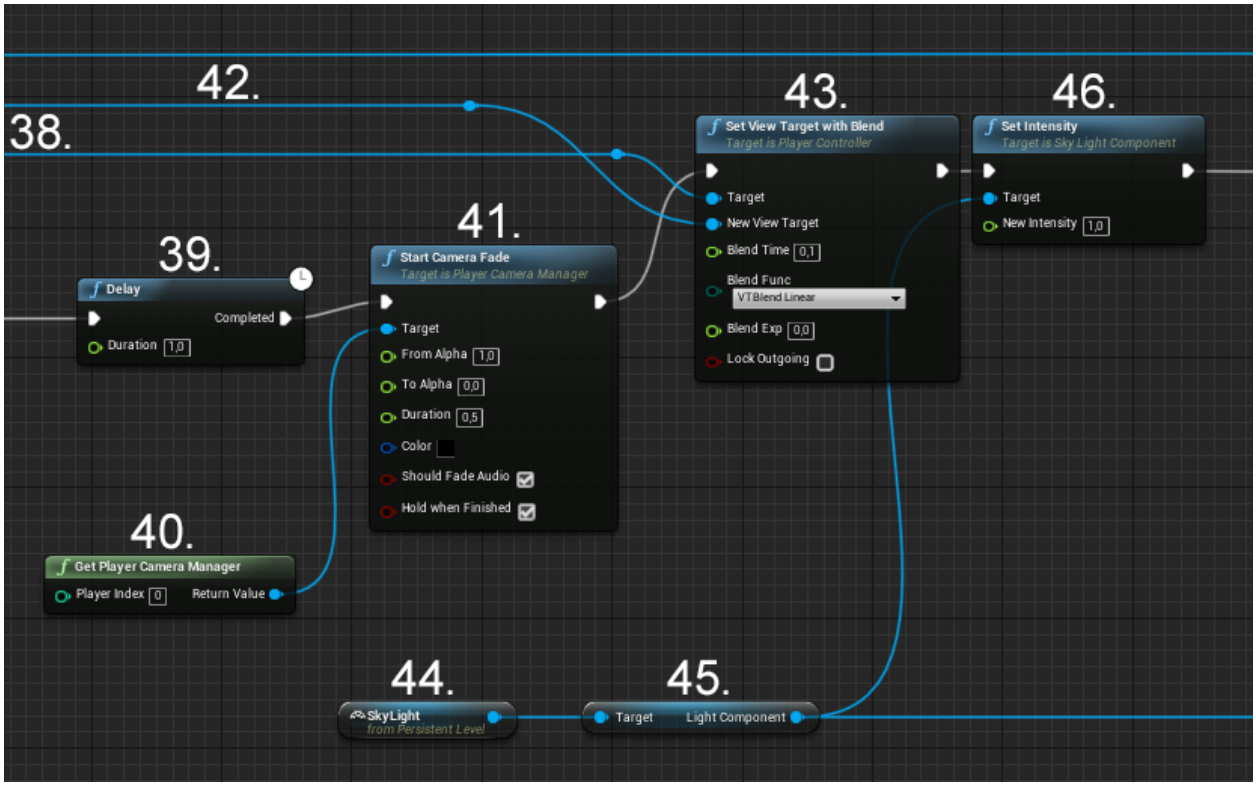


Figure 119. "Monster Camera" macro camera fade.

In figure 119, a "Delay" node (39) is the first node that is executed. It provides the "Start Camera Fade" node (41) with ample time to complete its fade action. The "Get Player Controller" node mentioned in figure 116 connects to the "Target" plug of the "Set View Target with Blend" node (43), which is the next node in the sequence. This node transitions the camera from the perspective of the player to that of the monster. The "Sky Light" node (44) references the skylight actor in the level. The "Target" node references the "Light Component" so that the "Set Intensity" node (46) can alter its brightness. The reason for this was explained on page 106. In figure 124, the intensity is set back to "0.0", which deactivates the effect that is enabled here.

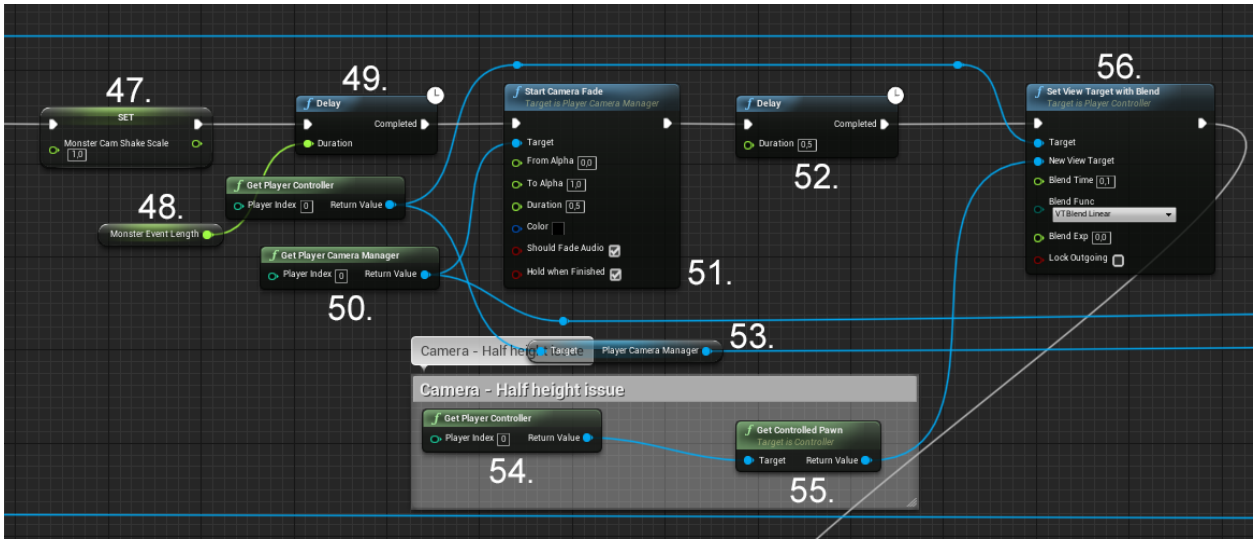


Figure 120. "Monster Camera" macro - monster event length.

Figure 120 is the next piece of code that enables the head bob effect and sets the duration of the "Monster Camera" event until it is deactivated. The value of "1.0" is fed into the "Monster Cam Shake Scale" variable with the "Set" node (47). This variable enables the head bob effect for the "Monster Camera" event. The length of the entire "Monster Camera" event is saved in the "Monster Event Length" float variable and is referenced by node (48). This value is then fed into the "Delay" node (49), which tells the system exactly how long to run the "Monster Camera" event until the camera fades back to the perspective of the player. The "Get Player Camera Manager" node references the camera that is currently in use and fades it to black using the "Start Camera Fade" node (51). Another "Delay" node (52) allows for an adequate amount of time to pass for the screen fade effect to complete. The "Get Player Controller" node is fed into the "Player Camera Manager" variable node (53) for later use.

The "Set View Target with Blend" node (56) transitions the camera perspective back from that of the monster to the perspective of the player. An issue was encountered when these nodes were used initially. When the camera transitioned back, the height of the player

character's camera halved. A fix was introduced to counter this. The "Get Player Controller" node (54) referenced the controller of the player before the transition. The "Get Controlled Pawn" extracts the values of this controller and fixes the camera height issue when fed into the "Set View Target with Blend" node (56). This node leads to another string of nodes, which will be explored below.

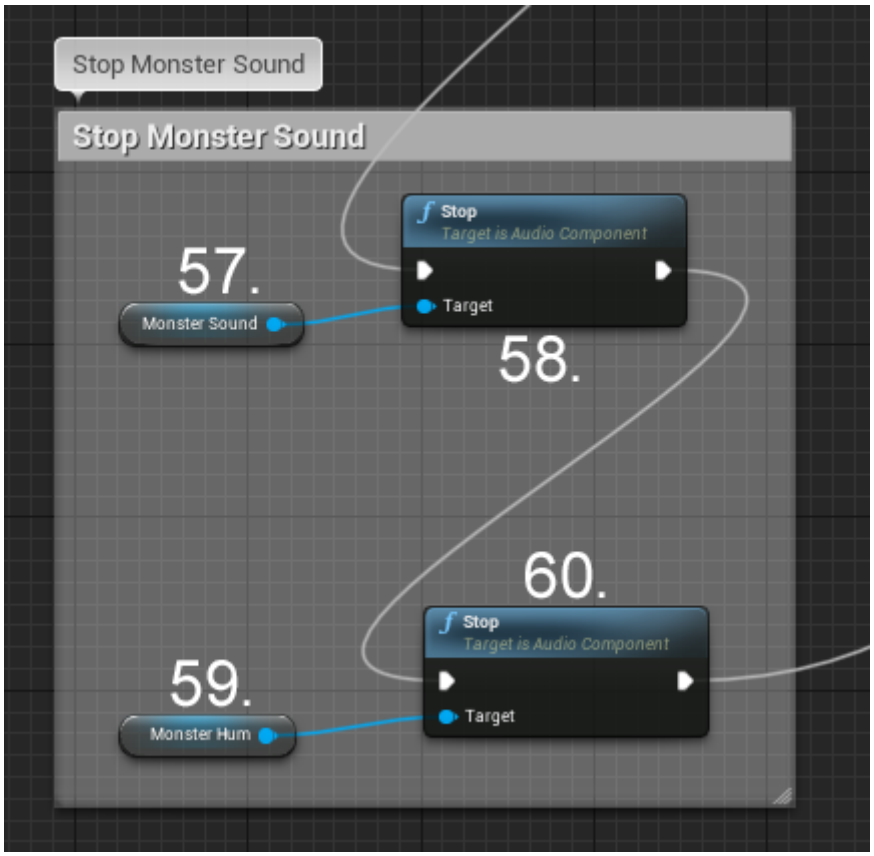


Figure 121. "Monster Camera" macro - sounds.

Figure 121 depicts the nodes that disable the monster breathing and hum sounds. The "Monster Sound" node (57) takes the reference that was set up earlier, and the "Stop" node (58) stops the sound from playing. Nodes (59) and (60) perform the same operation for the "Hum" sound.

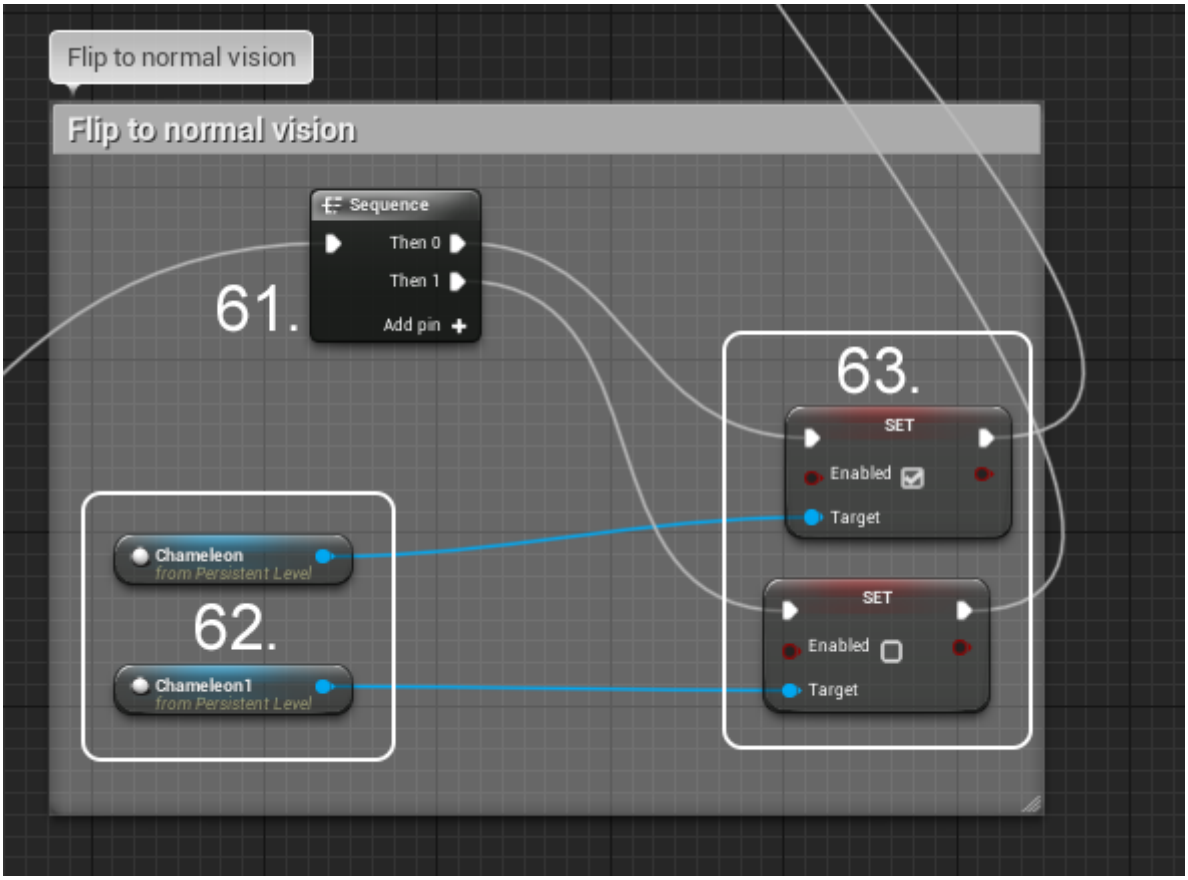


Figure 122. "Monster Camera" macro – monster vision.

Figure 122 switches back between the two post-processing effect packs and works in the same way as the method used in figure 118. The "Sequence" node takes the "Chameleon" effects packs (62) and uses the "Set" nodes (63) to reverse the effect pack enablement to what it was previously.

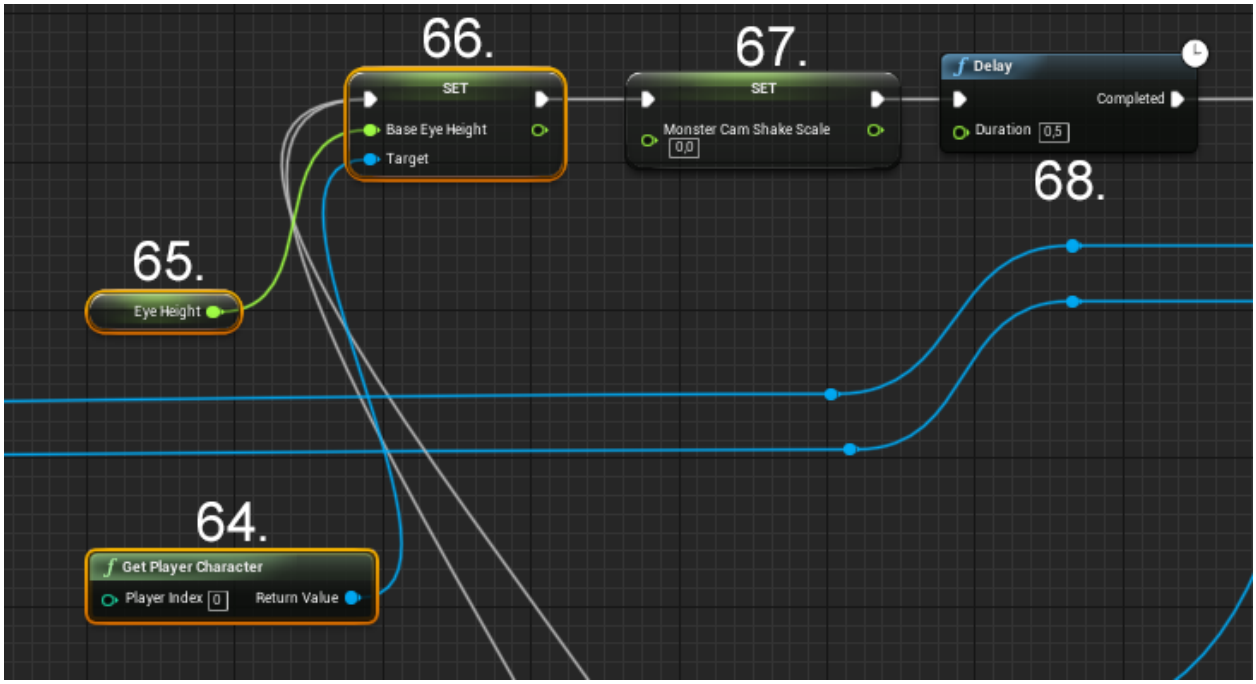


Figure 123. "Monster Camera" macro – camera half-height issue.

Figure 123 continues from the previous set of nodes and assists with countering the camera half-height issues that were discussed above. The "Get Player Character" node (64) is referenced so that the "Base Eye Height" value can be defined using the "Set" node (66). The "Eye Height" node is a reference of the variable that was set up in figure 115. This value is then fed into the "Set" node (66). The "Monster Cam Shake Scale" variable is set back to "0.0" using another "Set" node (67). This node disables the head bob effect that was enabled earlier on. The "Delay" node allows enough time to pass for the next "Start Camera Fade" node to complete. This node is part of the sequence of nodes that will be explored next.

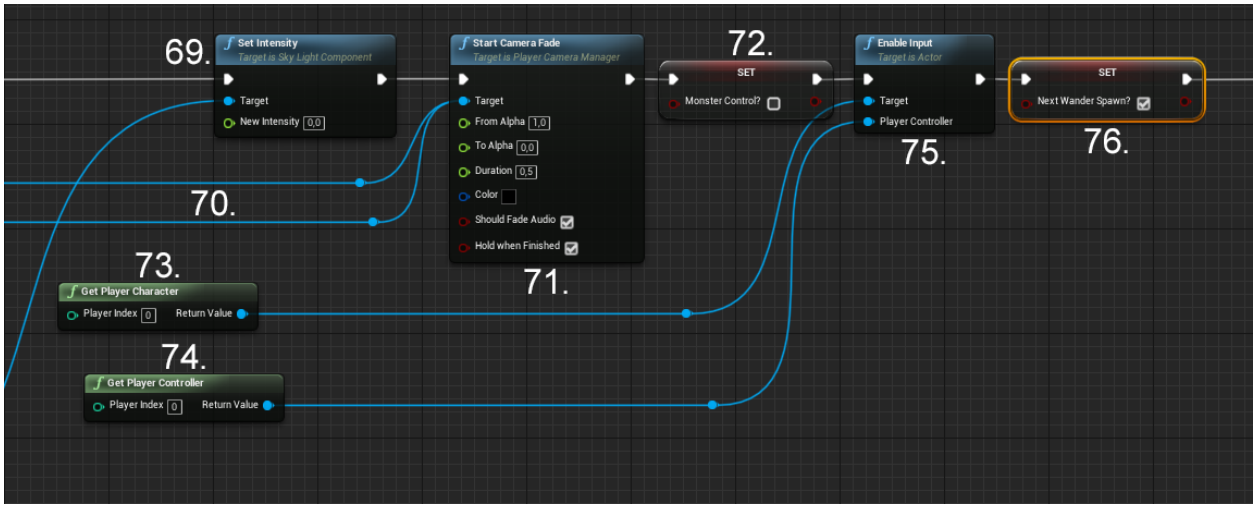


Figure 124. "Monster Camera" macro – transition back to player.

The "Set Intensity" node (69) references the "Sky Light" actor that was defined in figure 124. The intensity is set to "0.0" so that the additional ambient lighting is disabled for the player. The "Start Camera Fade" node makes use of two nodes (70) to drive it. These two nodes can be viewed in figure 120 and include the "Get Player Camera Manager" node (50) and the "Player Camera Manager" variable node (53). Back in figure 124, a "Set" node (72) disables the "Monster Control?" Boolean variable. The "Enable Input" node (75) takes the "Get Player Character" node (73) and the "Get Player Controller" node (74) to re-enable the controls for the player. The next "Set" node (76) sets the "Next Wander Spawn?" Boolean variable to true. This Boolean value tells the "Monster Camera" spawn system that is visible in figure 93 that the initial spawn location was used and that every macro with a "_Next" tag should be used going forward.

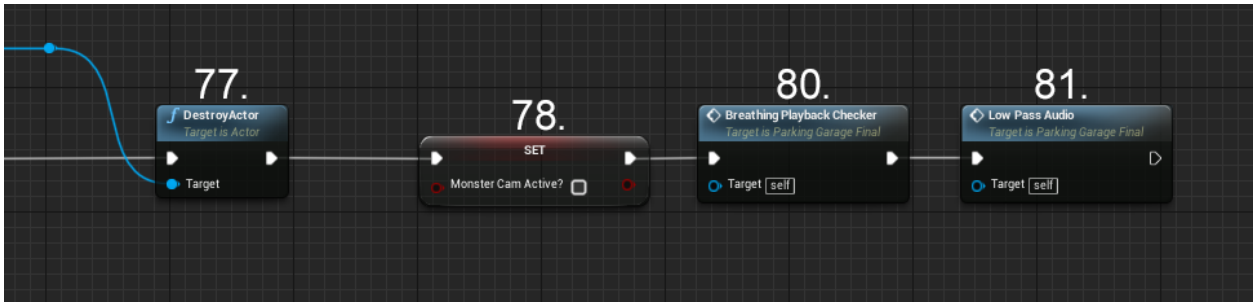


Figure 125. "Monster Camera" macro – additional effects.

Figure 125 shows the last string of nodes in the blueprint. The "Destroy Actor" node (77) takes the avatar that was spawned previously and removes it from the scene. The "Set" node (78) sets the "Monster Cam Active?" Boolean variable to false. This variable makes it possible for other parts of the system to know if the "Monster Camera" is active or not. The "Breathing Playback Checker" node (80) and the "Low Pass Audio" node (81) are custom audio nodes that add additional audio effects after each "Monster Camera" event. These functions will be explored in the section that deals with the breathing system.

Appendix 15: Monster Camera Macros – Movement Next

The "_Next" designation is used by the remaining macros. This designation means that the macro makes use of the monster's last known location to spawn the next starting point.

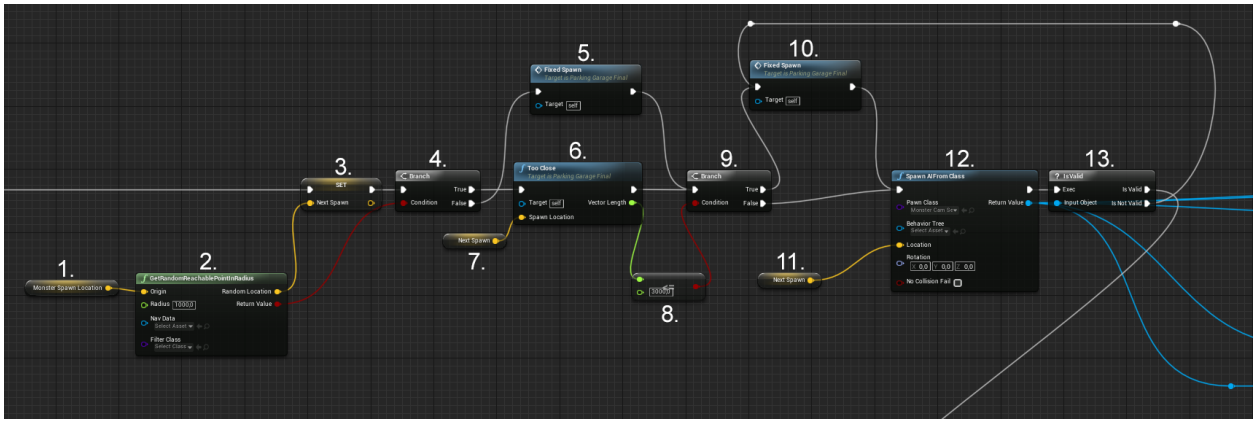


Figure 126. "Monster Camera" macro "_Next" variant.

Figure 126 shows how these macros differ, while the rest are the same. These additional nodes would fit in between the nodes shown in figure 115 and figure 116.

The "Monster Spawn Location" node (1) is the variable node established in figure 116. The "Get Random Reachable Point in Radius" node (2), finds the starting point in a radius around the point referenced in "Monster Spawn Location" node (1). This value is fed into a variable called "Next Spawn" using a "Set" node (3). This variable will be used throughout the blueprint, but also be fed into the "Next Spawn" node as soon as the blueprint reaches the end. This "Next Spawn" variable is created so that when the blueprint restarts, it uses the previously known location to generate a new starting point for the next "Monster Camera" spawn. The "Branch" node extracts a Boolean value out of node (2). This value checks to see if the spawn is valid. Creating a valid spawn point does sometimes fail due to engine limitations. This Boolean test enables the establishment of a contingency plan. When the "Branch" node tests false, a "Fixed Spawn" function node (5) fires. Several static spawn points were placed throughout the level, which the system can use to spawn the avatar. The use of static spawn points is usually a reliable alternative.

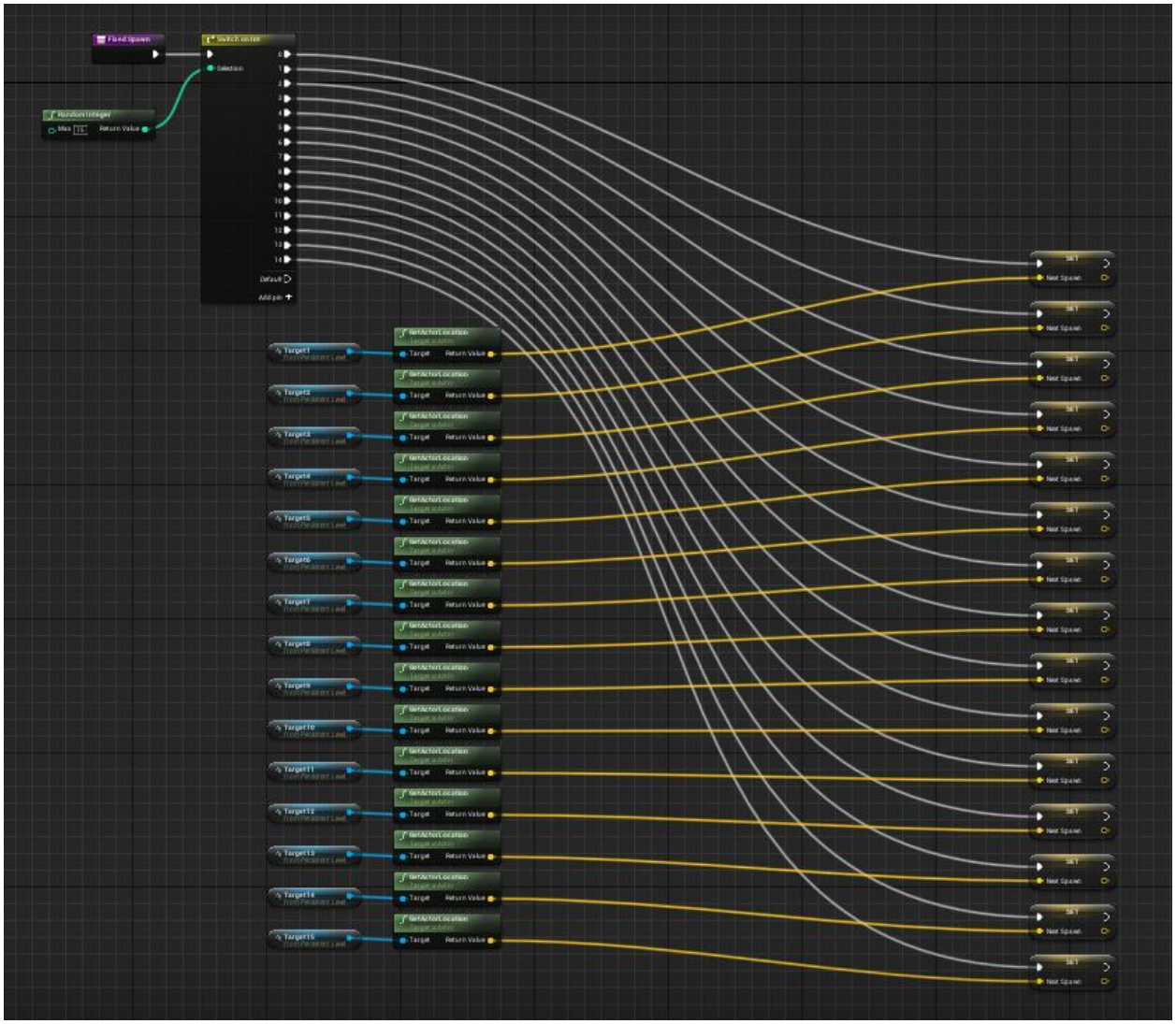


Figure 127. "Fixed Spawn" function.

Figure 127 shows the appearance of the "Fixed Spawn" function. It references the various static spawn points, randomly selects the one it wants to use and then feeds the vector value into the "Next Spawn" variable.

When the "Branch" node (4) in figure 126 tests true, the custom "Too Close" function node (6) is executed, which takes the "Next Spawn" variable node (7) as its input, the output of this node is then compared using the "< =" node (8) to check against a value of "3000". This Boolean value determines whether the next "Branch" node (9) is either true or false.

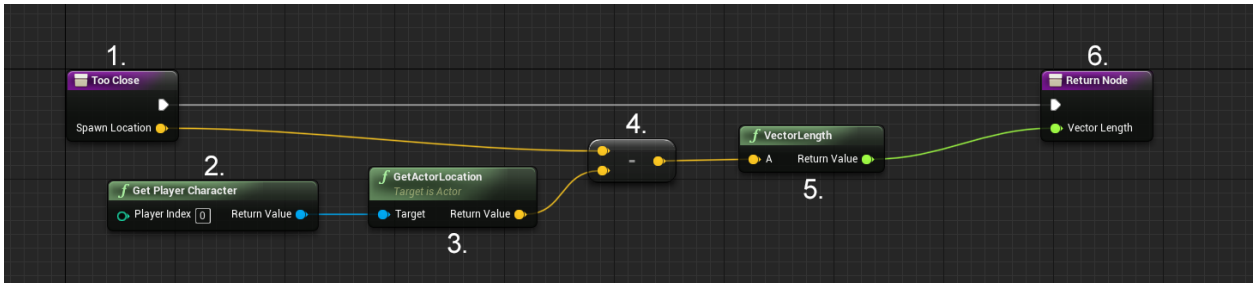


Figure 128. "Too Close" function.

Figure 128 shows the "Too Close" function that is used to check when a destination node for the "Monster Camera" is too close to the player. Node (1) is the input that fires the rest of the nodes. A "Get Player Character" node (2), which references the player, is used to extract its location using the "Get Actor Location" node (3). A "-" node (4) then takes the "Spawn Location" input, which is the "Next Spawn" variable shown in figure 126 and subtracts it from the player's location. This result is then transformed into a vector length variable using the "Vector Length" node (5). This value is then plugged into the "Vector Length" output of the "Too Close" function.

The "Branch" node (9) will test true when the above value is tested against node (8) back in figure 126 and is deemed smaller than or equal to the value of "3000" unreal units. This result means that the monster's avatar is too close to the player. When true, the "Fixed Spawn" function is called again (Figure 127). When false, the "Next Spawn" node (11) is used to determine the location of the "Spawn AI from Class" node (12), which then spawns the avatar chosen in the "Pawn Class" drop-down. Another "? Is Valid" is used to determine if the spawn point is valid. When "Not Valid", the "Fixed Spawn" function is called again, to make sure that the avatar spawns in the level.

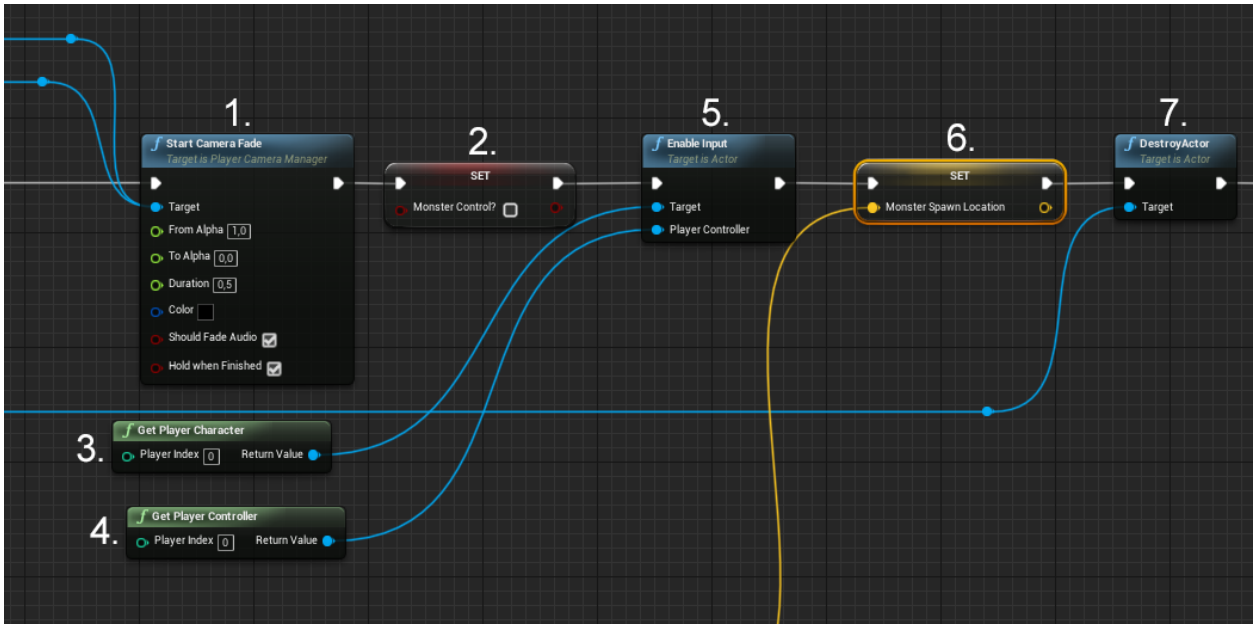


Figure 129. "_Next" macro – "Monster Spawn Location".

Figure 129 shows the addition of a variable node called "Monster Spawn Location" in the "_Next" macros. This variable is like the one used in figure 126. It effectively takes a destination that was generated using the "Next Spawn" variable's location and then feeds it into this variable using a "Set" node (6).

This node (6) replaces the "Next Wander Spawn" variable node shown in figure 124 and fits in between the "Enable Input" node (75) in figure 124 and the "Destroy Actor" node (77) shown in figure 125.

The "MonsterApproach_Next" macro adds a blueprint that checks to see if the destination location for the monster avatar is too close to the player. This blueprint was added to make it appear as if the monster is approaching from afar.

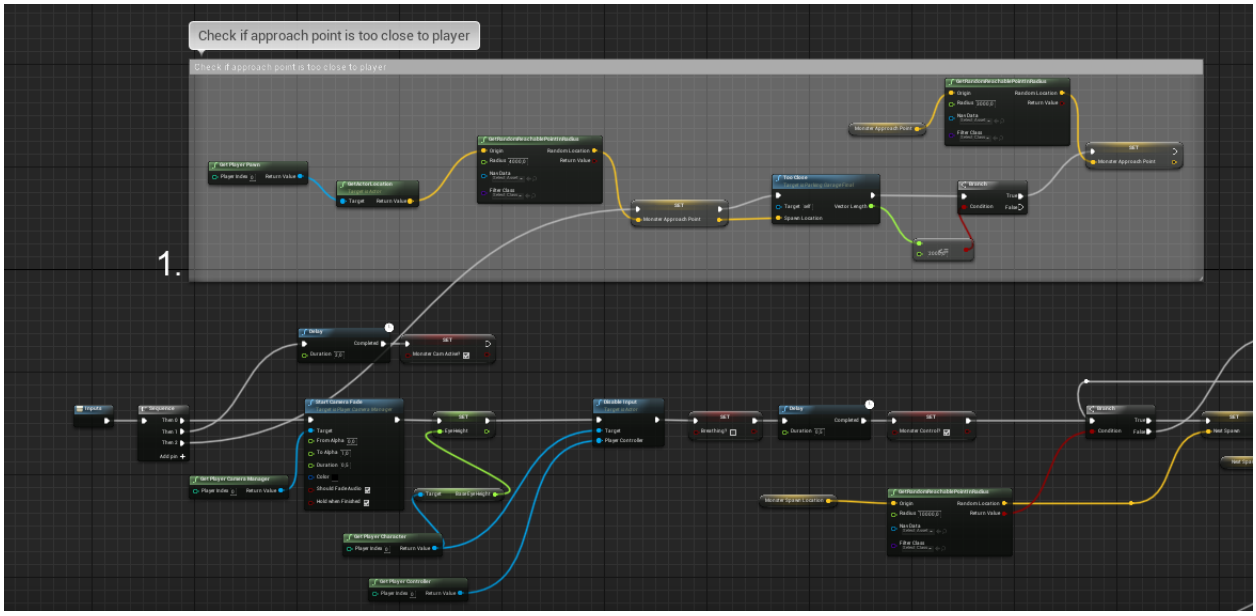


Figure 130. "MonsterApproach_Next" macro – approach point.

Figure 130 shows how this was implemented in the "MonsterApproach_Next" macro (1). The code in question (1) is launched at the same time as the other nodes.

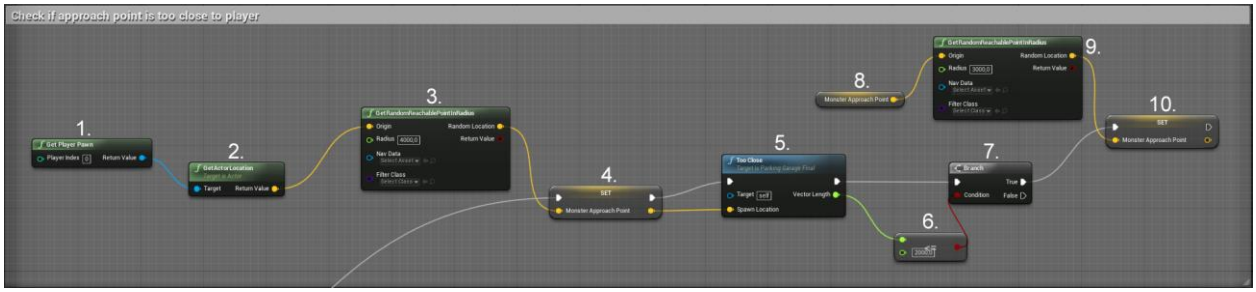


Figure 131. "MonsterApproach_Next" macro – approach point closeup.

Figure 131 shows a close-up of this script. The "Get Player Pawn" node (1) references the location of the player pawn, while the "Get Actor Location" node (2) extracts its location. This location is fed into the "Get Random Reachable Point in Radius" node (3) to generate a destination within the given radius of "4000" unreal units. The "Set" node (4) then takes the generated location and feeds it into the "Monster Approach Point" variable. The value of this variable is then fed into the custom "Too Close" function node (5) to determine if it is too

close to the player. The "< =" node (6) determines whether the destination is closer than "2000" unreal units to the player. When the result of the "Branch" node (7) is false, nothing happens because the "Monster Approach Point" variable already contains the last best value. When the result of the "Branch" node (7) is true, then a "Monster Approach Point" variable node (8) is used to calculate another random destination. The "Get Random Reachable Point in Radius" node (9) is used to achieve this random destination, which is then fed into the "Monster Approach Point" variable using a "Set" node (10).

Appendix 16: Monster Camera Macros – Kill Player

The "MonsterKillPlayer" macro is similar to the "_Next" variant, but with some minor adjustments.

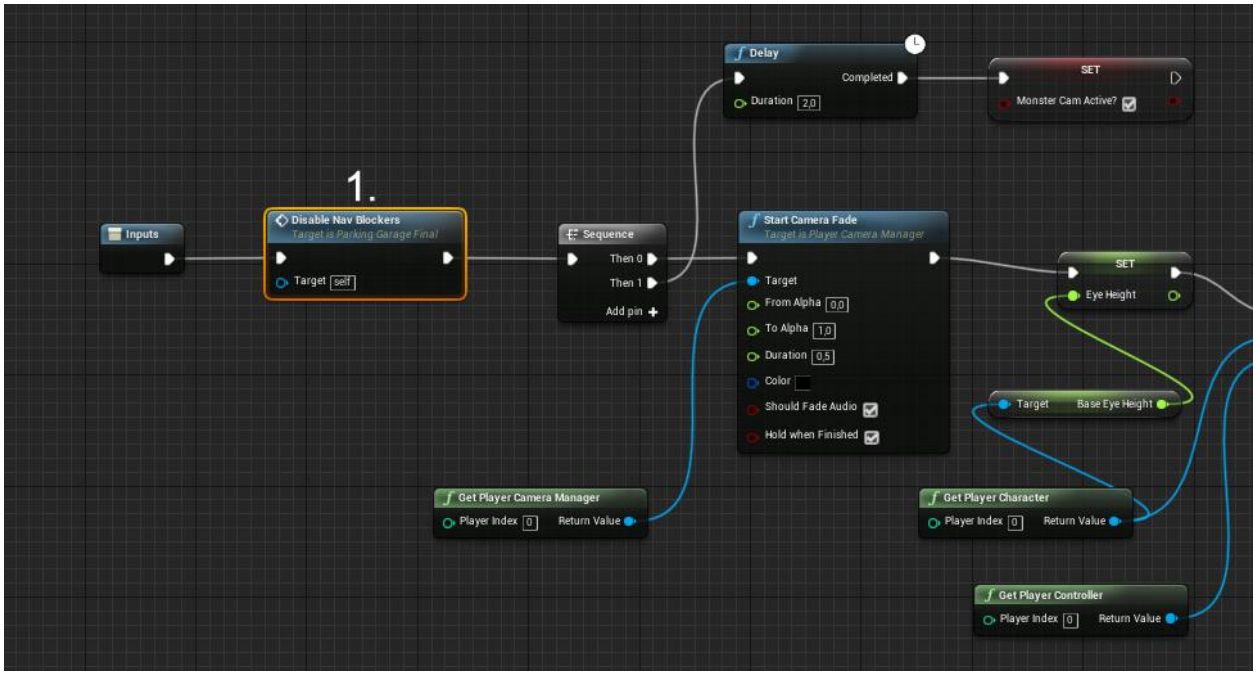


Figure 132. "MonsterKillPlayer" macro – part 1.

Figure 132 shows the first change at the start of the script. The “Disable Nav Blockers” node (1) launches a custom function. This function disables all nav blockers.

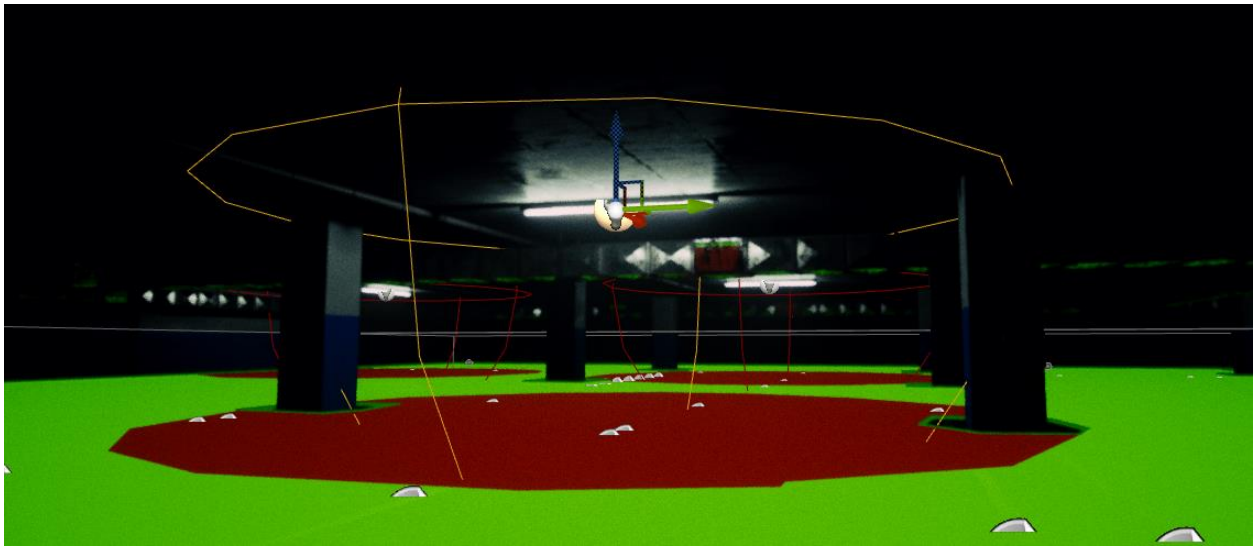


Figure 133. Nav Blocker.

Nav blockers are custom collision spheres that prevent the AI avatars from navigating through lit areas (figure 133). The various “Nav Blocker” actors are placed below every light in the level. Adding these blockers will make it appear as if the monster is afraid of lit areas.

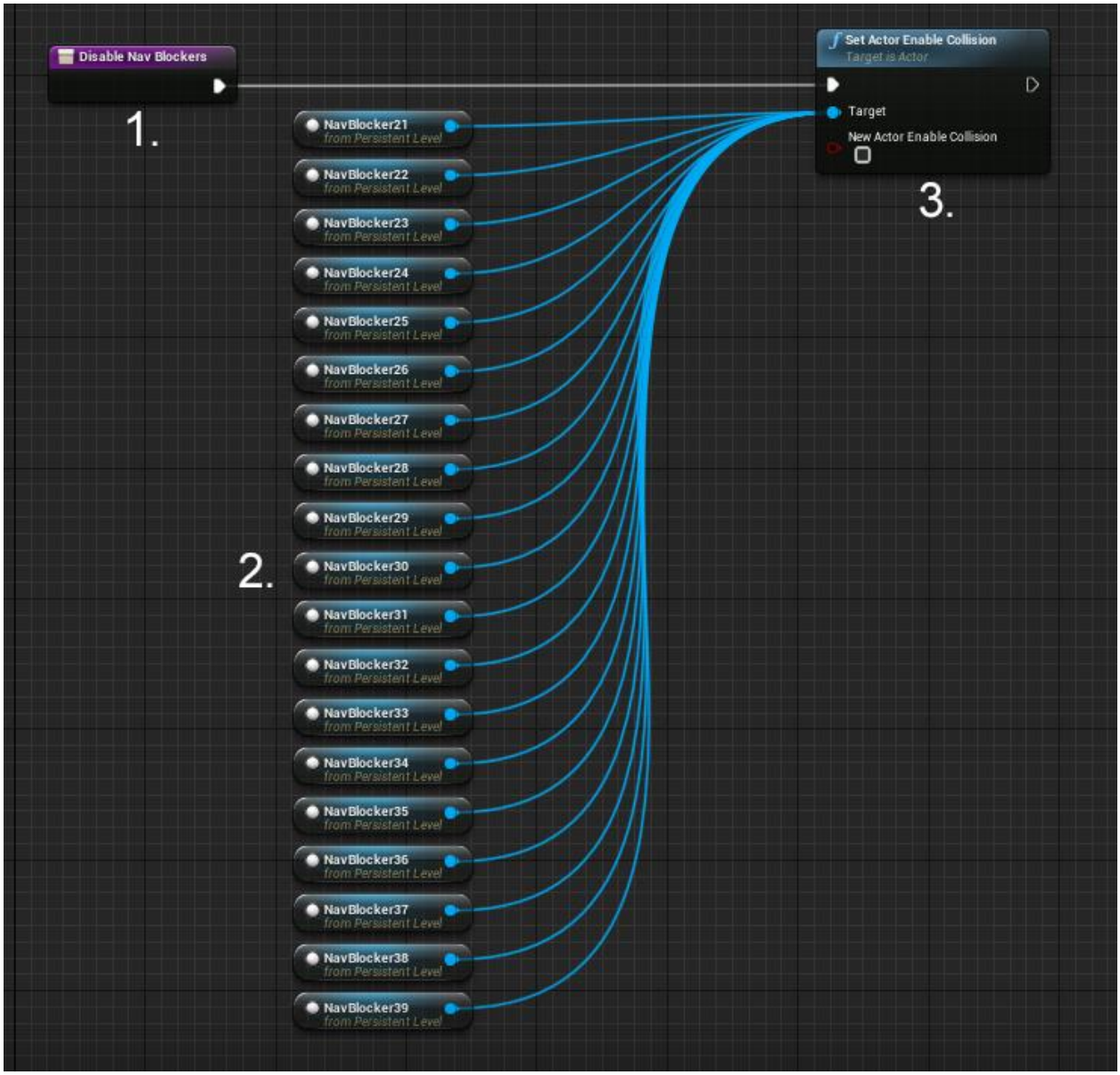


Figure 134. "Disable Nav Blocker".

Figure 134 shows the input of the "Disable Nav Blockers" function (1). The "Nav Blocker" actors (2) are referenced in the function so that their collision features can be disabled. These actors do not block anything except the navigation mesh component that is set up within the level. The AI actor is, therefore, unable to traverse these areas. The "Set Actor Enable Collision" node (3) disables the collision for these actors.

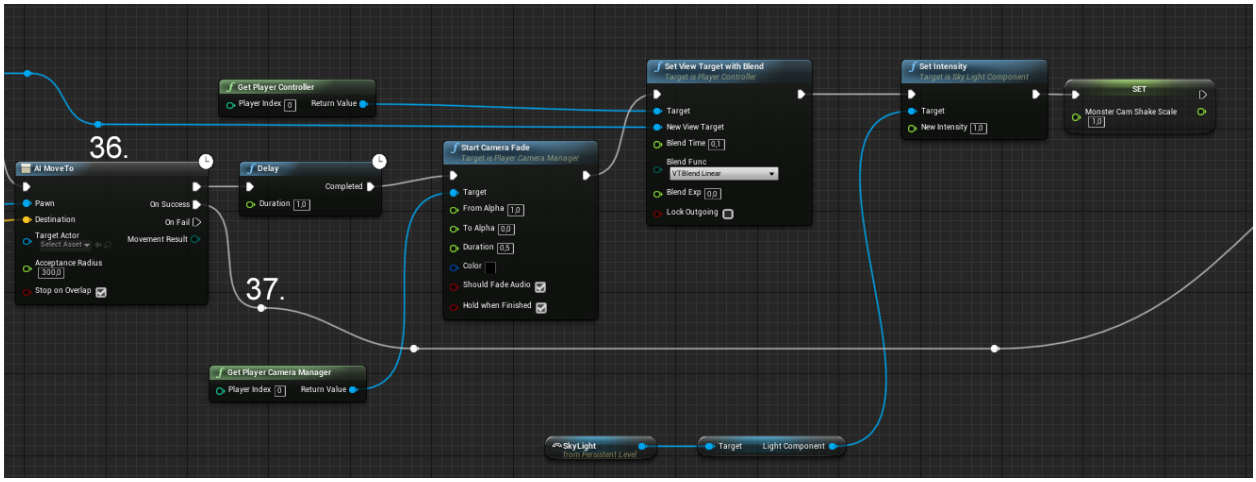


Figure 135. "MonsterKillPlayer" macro – part 2.

Figure 135 shows that right after the "AI Move To" node (36) is called, the screen fades to the "Monster Camera" perspective as per usual, but that it does not fade out after a while. Removing the fade takes the amount of time it will take for the monster to reach the protagonist into account. Only when the "On Success" state (37) is reached can the final script execute. The "On Success" state is reached when the actor reaches its destination. The final nodes will be explored below.

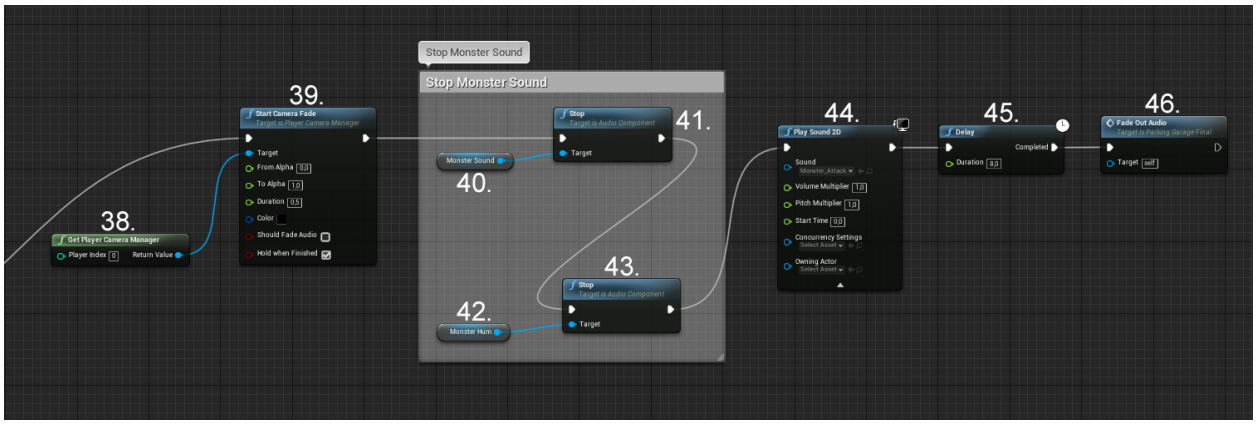


Figure 136. "MonsterKillPlayer" macro – part 3.

In figure 136, the "Get Player Camera Manager" node (38) retrieves a reference of the current camera manager, which belongs to the monster's avatar. The "Start Camera Fade"

node (39) fades the camera of the monster back to black. There is no intention to fade it back to the perspective of the player, however, as this sequence is intended to be the end game state where the player dies. Nodes (40), (41), (42), and (43) are used to stop the sounds of the monster breathing and the subtle hum that accompanies it. A “Play Sound 2D” node (44) is then used to play the sound of the monster attacking and killing the player, which is followed by short delay (48) and the “Fade Out Audio” node (46) which is a custom function node that fades the ambient sounds of the level out to silence.

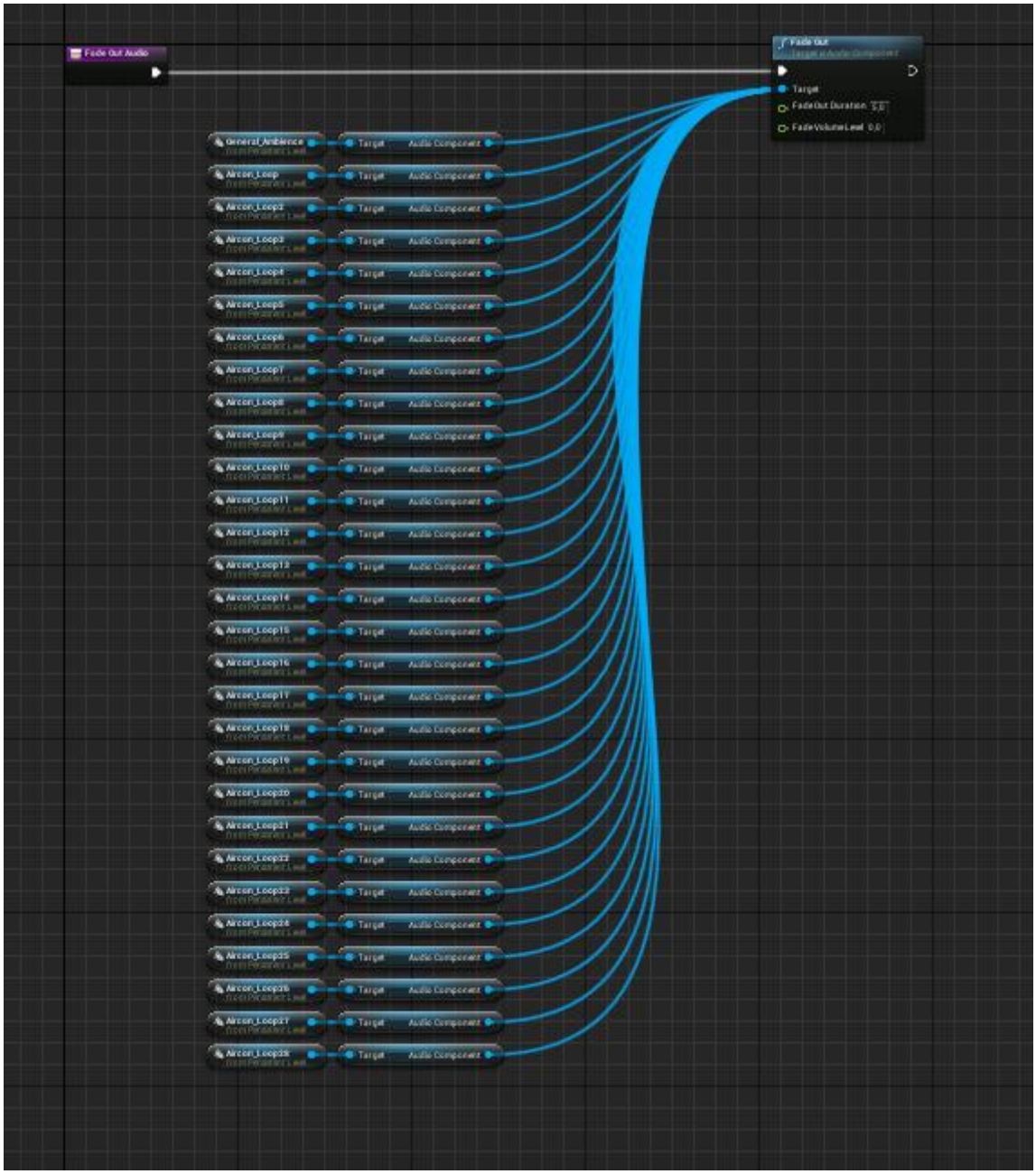


Figure 137. "Fade Out Audio" function.

Figure 137 shows the "Fade Out Audio" function in question.

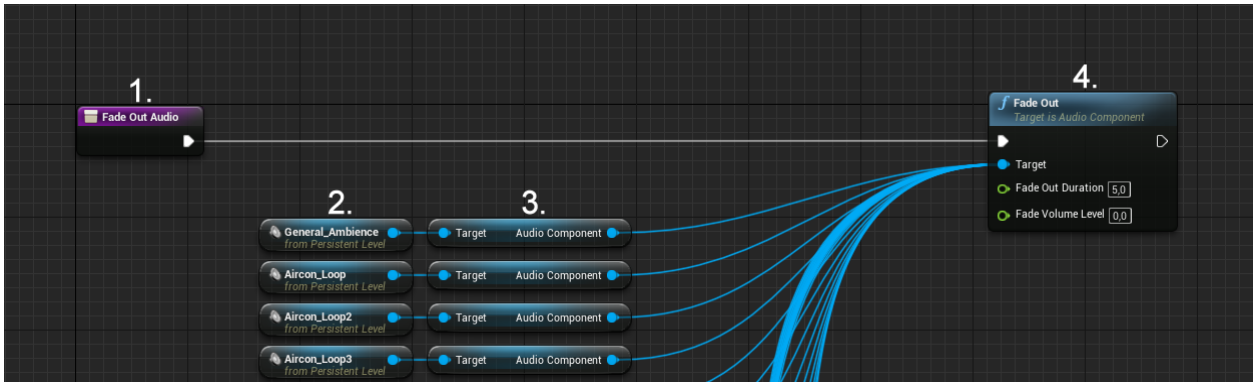


Figure 138. "Fade Out Audio" function close-up.

A close-up of the "Fade Out Audio" function (figure 138) shows that the group of nodes under (2) and (3) targets all of the audio samples in the level, which are then fed into the "Fade Out" node that slowly fades the volume out over five seconds.

Appendix 17: Monster Camera Macros – Disable Light

The "Monster Disable Light" macro works the same way as the "_Next" variant, except for the location where the "Monster Camera" spawns. It does not use the previously known location of the monster.

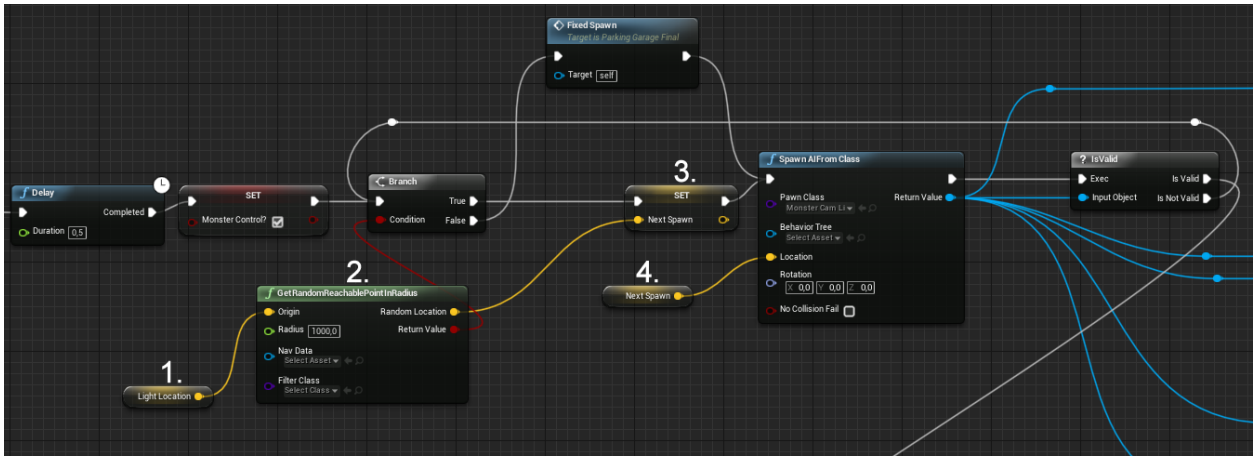


Figure 139. "Monster Disable Light" macro.

Figure 139 shows how the “Disable Light” macro spawns the character near a light source. The “Light Location” variable, which was set up in figure 101, is used as the origin point for the “Get Random Reachable Point in Radius” node (2) to create a destination within “1000” unreal units. This destination is then fed into the “Next Spawn” variable with a “Set” node (3). This variable (4) is used as the spawn location for the “Monster Camera”. Not depicted in figure 139 is the destination point, which is calculated using the starting location of the spawned “Monster Camera” and another “Get Random Reachable Point in Radius” node with a radius of “1000” unreal units.

Appendix 18: Monster Camera Timers

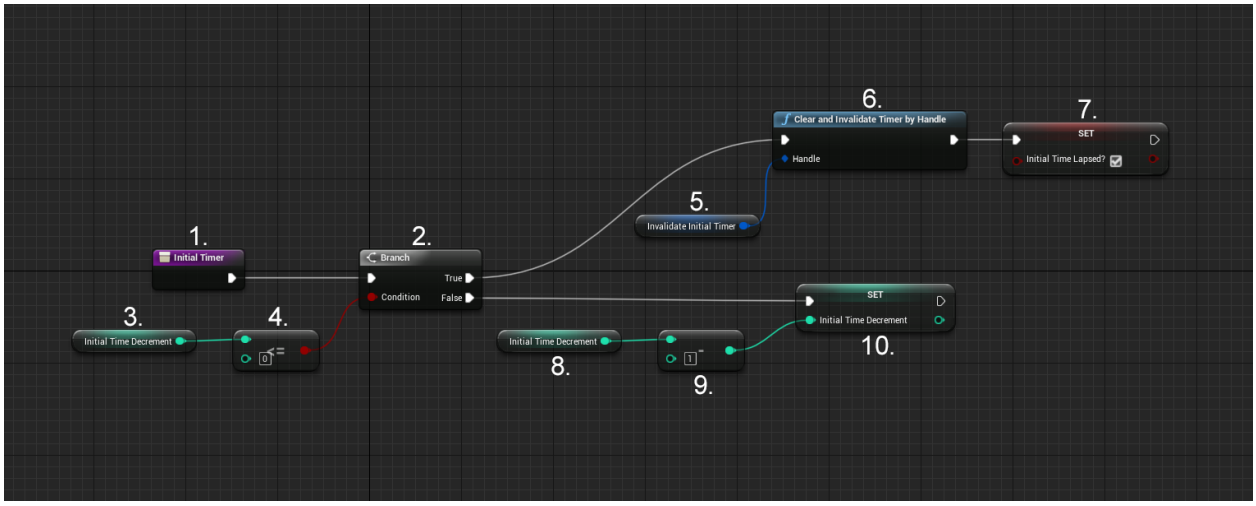


Figure 140. “Initial Timer” function.

Figure 140 shows the “Initial Timer” function that is the very first timer that is executed the moment the player collects the flashlight. Within the period of this timer, the antagonist merely roams around, looking for the player.

The “RandomMonsterMovement_Initial” and “RandomMonsterMovement_Next” macros are executed within this period. The “Initial Timer” node (1) is the node that executes the remaining nodes in the function. A “Branch” node (2) tests whether the “Initial Time Decrement” variable (3) is smaller than or equal to zero (4). When the result of the “Branch” node (2) is false, the “Initial Time Decrement” variable (8) is decreased by “1” (9) and fed back into the “Initial Time Decrement” variable using a “Set” node (10). When the “Branch” node (2) test returns true, a reference of the timer, which is saved into the “Invalidate Initial Timer” variable (5), is used to clear the timer using the “Clear and Invalidate Timer by Handle” node (6). The “Initial Time Lapsed?” Boolean variable (7) is also set to true.

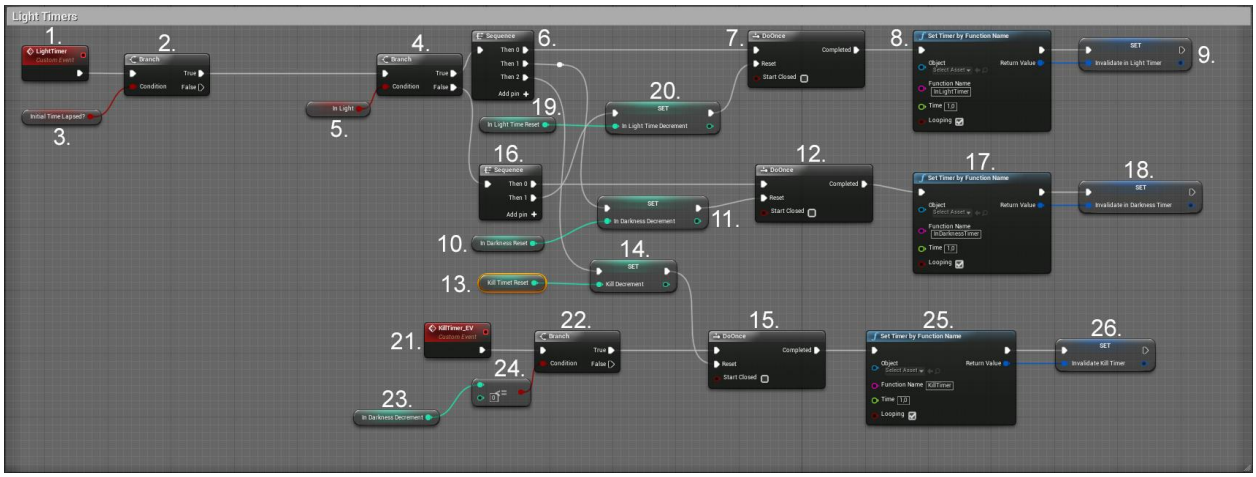


Figure 141. “Light Timer” blueprint.

Figure 141 shows the primary timer manager, which becomes active as soon as the “Initial Timer” lapses. It chooses which timer to play depending on the actions of the player in the level. The “Light Timer” node (1) is the custom event node that launches the timer operation. The “Initial Time Lapsed?” variable will determine if the rest of the code executes via the “Branch” node (2). The next test takes the “In Light” variable to determine if the “In Light Timer” or if the “In Darkness Timer” launches. This blueprint is a “tick” operation, so “Do Once” nodes (7, 12, and 15) are needed to launch the timers once only. The “Set Timer

by Function Name" nodes (8, 17, and 25) are set to execute once every second and loop indefinitely. These timers are stopped within the various timer functions. There are three different timer functions that the "Set Timer by Function Name" nodes (8, 17, and 25) can execute, namely the "In Light Timer", the "In Darkness Timer" and the "Kill Timer". The "Kill Timer" is launched by the custom "KillTimer_EV" event call (21). The "In Darkness Timer" only launches when the "In Darkness Decrement" variable (23) is equal to or below zero (24). All three timers are fed into variables (9, 18, 26) that allow the timers to be disabled within the three timer functions. Three variable nodes namely the "In Light Time Reset" (19), the "In Darkness Reset" (10) and the "Kill Time Reset" (13) store the original durations of each timer. These variable nodes are used to repopulate the three "Decrement" variables (11, 14, and 20) so that when these timers reset, they start counting down from their original values.

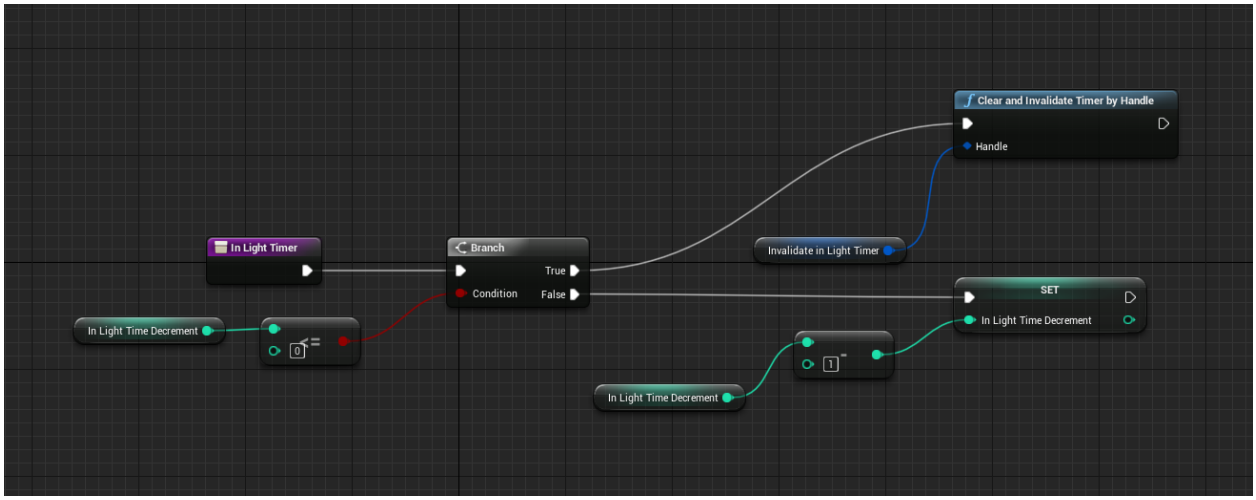


Figure 142. "In Light Timer" function.

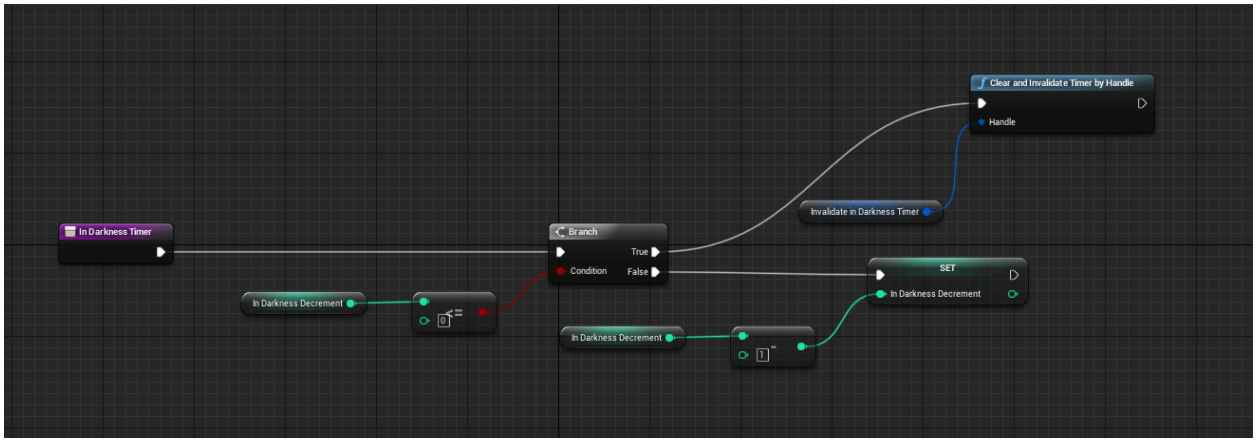


Figure 143. "In Darkness Timer" function.

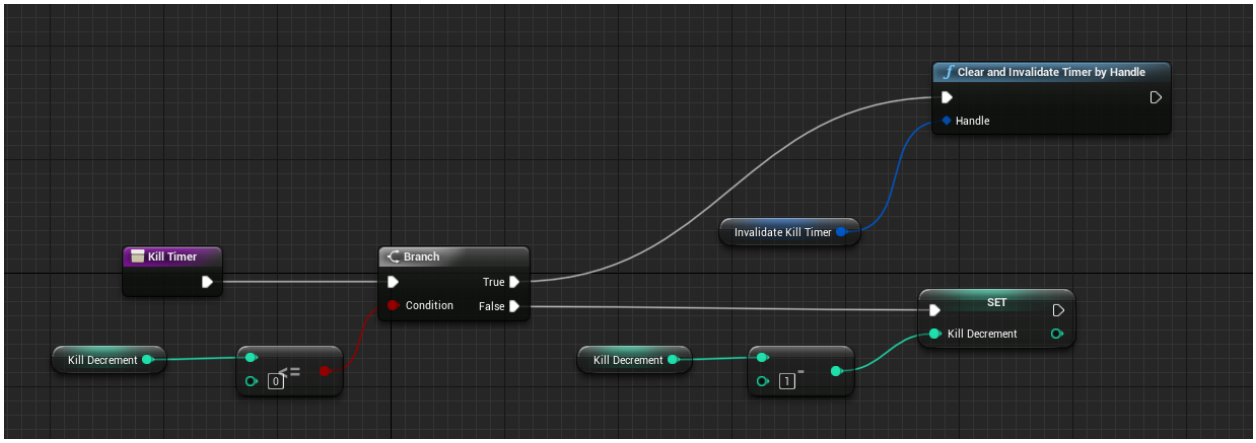


Figure 144. "Kill Timer" function.

Figures 142, 143, and 144 show the various timer functions. They all work in the same way as the "Initial Timer" function, bar their respective "Time Decrement" variables.

Appendix 19: Player Avatar

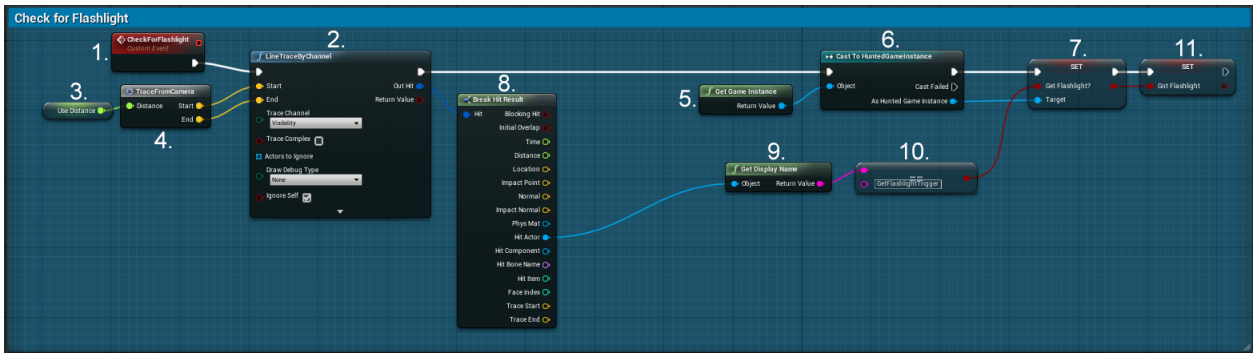


Figure 145. "Check for Flashlight" blueprint.

Figure 145 depicts the nodes within the "Check for Flashlight" blueprint. The "Check for Flashlight" node (1) is the first node that executes the rest. The "Use Distance" node (4) contains a variable that determines the length of the trace that is called within the "Trace From Camera" macro (4). The length determines how far the player must be from the object in question before the player can interact with it. The "Line Trace by Channel" node (2) shoots out the trace that was called by the macro (4). The "Out Hit" output contains information regarding the objects the trace hits. The "Break Hit Results" node (8) breaks the information open so that the relevant item can be retrieved and used. The "Hit Actor" output sends the name of the actor the trace hits, while the "Get Display Name" node (9) retrieves the name of this actor. The "= =" node determines whether the actor that the trace hits has a similar name to the one entered into the node. The resulting Boolean value is then fed into the "Get Flashlight?" variable node, which exists within the game instance (5, 10), and the "Got Flashlight" variable node, which is a local variable.

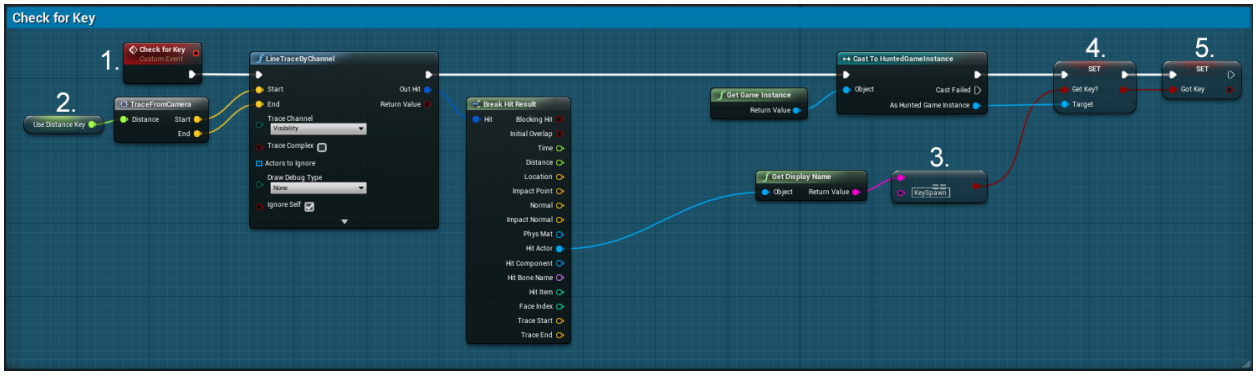


Figure 146. "Check for Key" blueprint.

The "Check for Key" blueprint in figure 146, works the same way as the "Check for Flashlight" blueprint. The only difference is that the value stored in the "Use Distance Key" variable node (2) is more than the value used in figure 146. This increased value compensates for the height of the player. Players need to be able to pick the key up when they look down at it from a standing height. The Boolean variables stored in nodes (4) and (5), lets the system know that the key can be picked up.

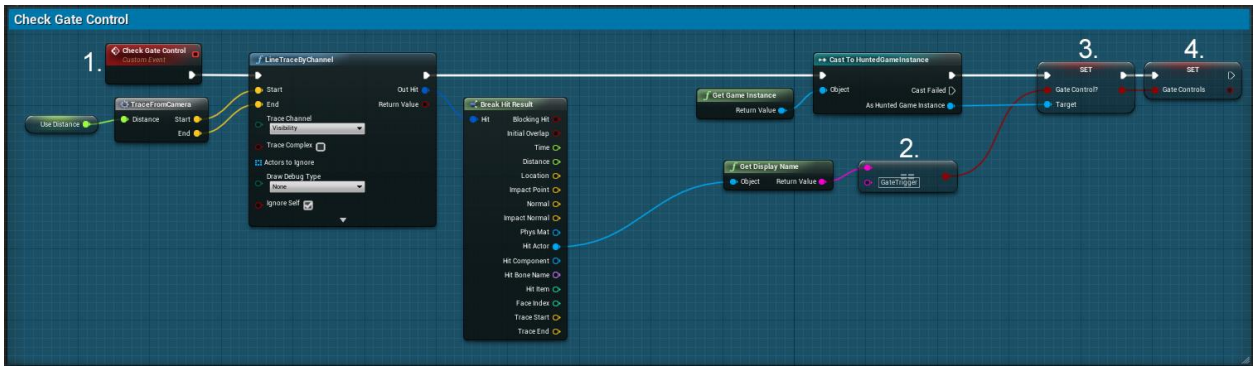


Figure 147. "Check Gate Control" blueprint.

The same is valid for nodes (3) and (4) in the "Check Gate Control" blueprint shown in figure 147.

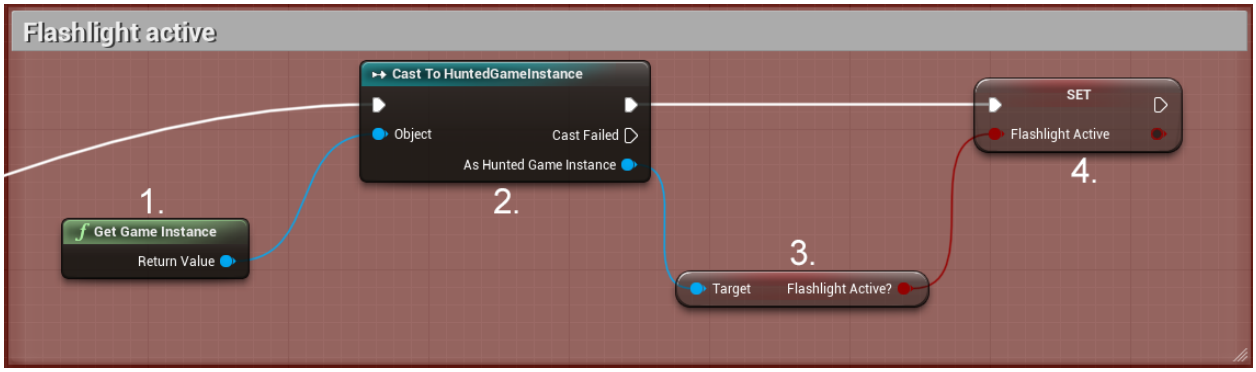


Figure 148. Cast "Flashlight Active?" state to game instance.

The player's avatar also needs to check whether the flashlight has been collected yet or not. In figure 148, the "Flashlight Active?" variable (3) that is pulled out of the game instance (1, 2) is pushed into the "Flashlight Active" variable. This variable tells the player's avatar that the flashlight can be activated when the flashlight key is pressed.

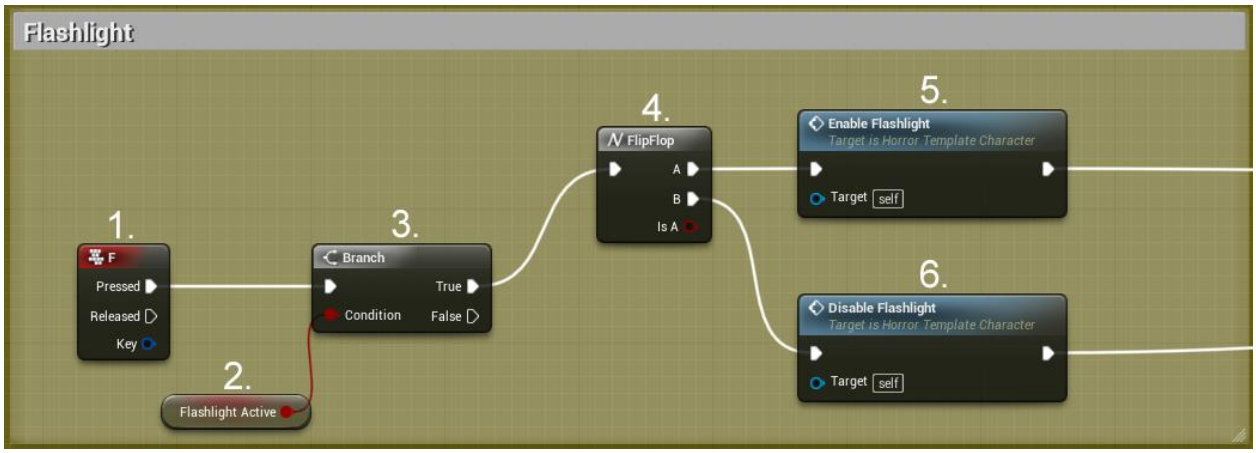


Figure 149. Flashlight switch blueprint.

Figure 149 shows where the "Flashlight Active" variable is used. Whenever the player presses the "F" key (1), a "Branch" node determines if the flashlight can be activated or not. The "Flashlight Active" variable (2) determines this outcome. When true a "Flip Flop" node (4) launches either the "Enable Flashlight" or "Disable Flashlight" function, these functions enable and disable the light source on the flashlight but will not be covered in this paper.

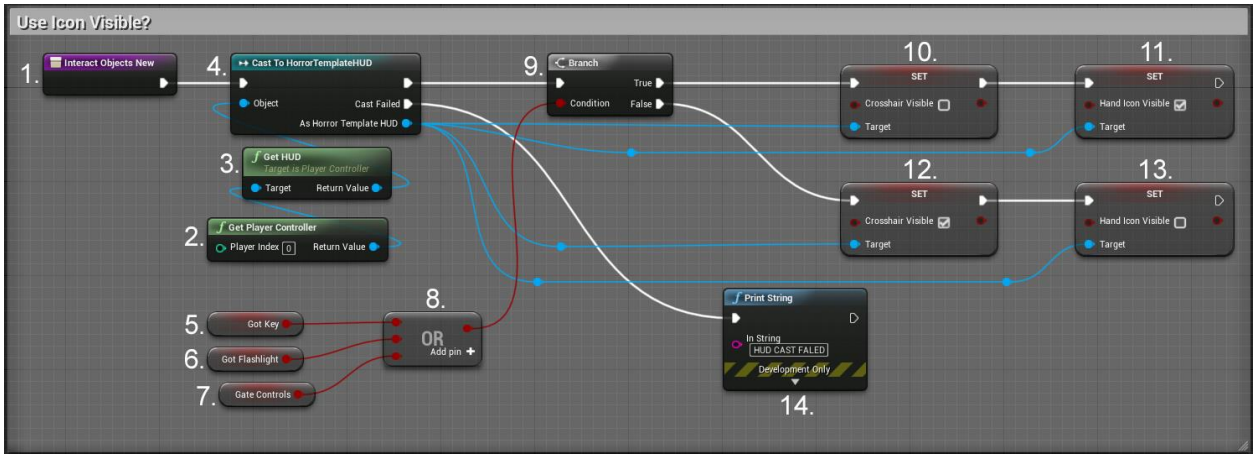


Figure 150. Use icon visibility checker blueprint.

Figure 150 shows where the Boolean variables (5, 6, 7), which are determined by the "Check for" functions, are used to determine whether the "use" icon is shown or not. A "Get Player Controller" node (2) is needed to target the player avatar's heads up display component. A "Get HUD" node (3) can retrieve the heads up display component while the "Cast to HorrorTemplateHUD" node (4) sends a message to the heads up display telling it to change in the way that is specified by the next string of nodes. The "Branch" node tests, whether either of the Boolean variables (5, 6, 7) are currently set to true using the "Or" node (8). When true, the crosshair (10) is hidden while the hand icon (11) is set to visible. When the "Branch" node tests false, the crosshair icon (12) is shown while the hand icon (13) is hidden.

Appendix 20: Player Avatar – Breathing System

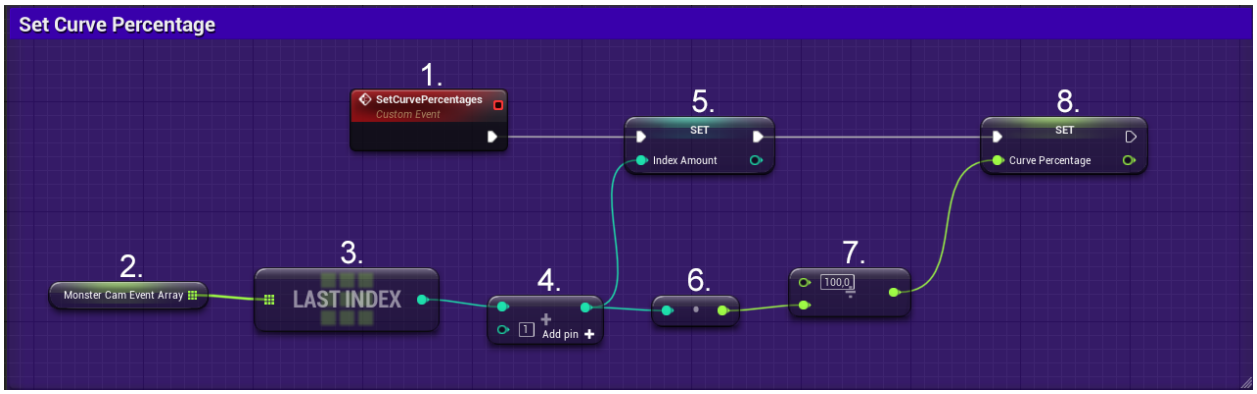


Figure 151. "Set Curve Percentage" blueprint.

Figure 151 shows the function that sets the percentage slices of each event within the suspense curve. The algorithm proposed here is to divide one hundred with the number that constitutes the last index of the array, which is the total amount of items in the array. The result is the percentage amount for one slice in the entire suspense curve. If there are five items in the array, for example, the percentage amount for one event slice would be twenty.

The "Set Curve Percentages" node (1) executes the script as soon as the level starts up. The "Monster Cam Event Array" variable node (2) is a reference of the array, which contains the number of events on the suspense curve and the delay between each that was fed into this variable node earlier. The "Last Index" node (3) extracts the number of the last event in the array, and the "+" node (4) increments this value by one. The value needs to be incremented by one because arrays in UE4 start recording at zero. When the "Last Index" value is four, there really are five array elements. After incrementing the value, it is fed into the "Index Amount" variable using the "Set" node (5). Node (6) converts the integer value into a float value so that a more accurate percentage value can be calculated. A value of "100" is then divided through the converted value with node (7). The resulting value is then fed into the "Curve Percentage" variable with another "Set" node (8).

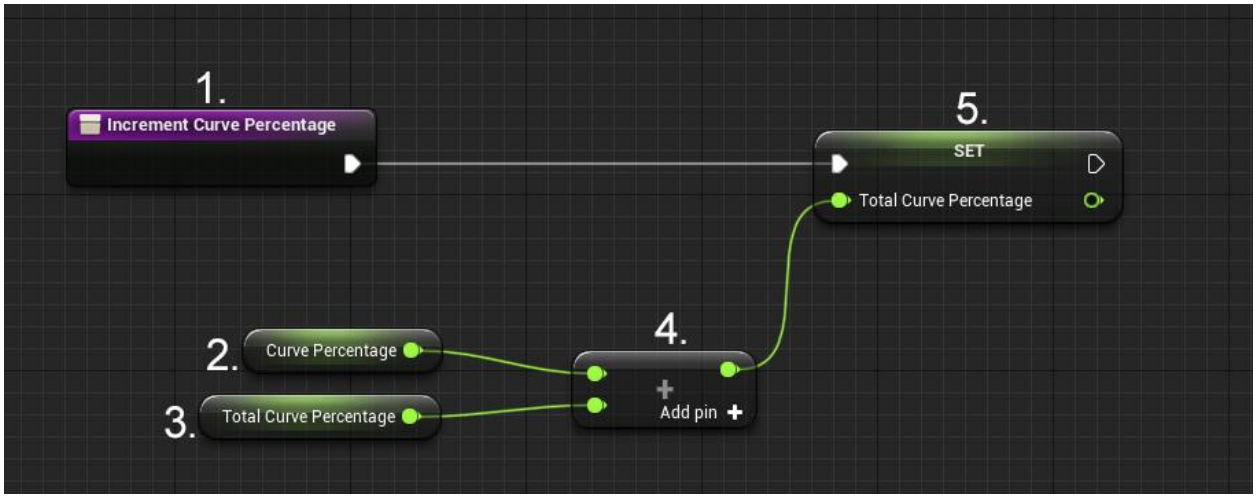


Figure 152. "Increment Curve Percentage" function.

Figure 152 shows the function that takes the "Curve Percentage" variable (2) that was set up above and increments it with the same amount. The "Total Curve Percentage" variable node (3) starts at a value of zero. The "+" adds the "Curve Percentage" value to this value and saves it back into the "Total Curve Percentage" variable with a "Set" node (5). This function will keep on firing until the last event is reached, or when the value of the "Total Curve Percentage" variable is "100".

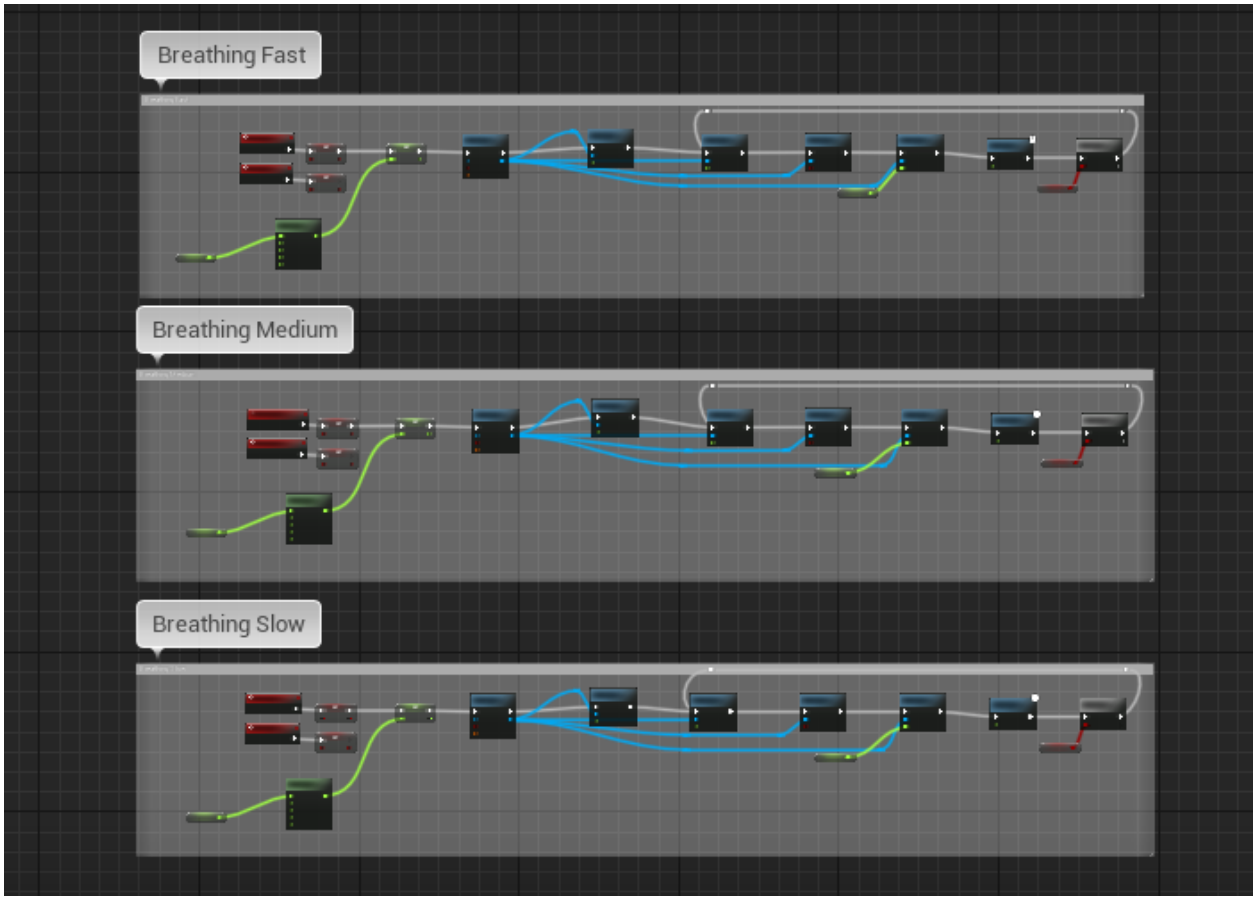


Figure 153. Breathing system.

Figure 153 shows the breathing system that fires the respective breathing sounds. There are three separate breathing speed audio cues, that can be played depending on the percentage value in the “Total Curve Percentage” variable.

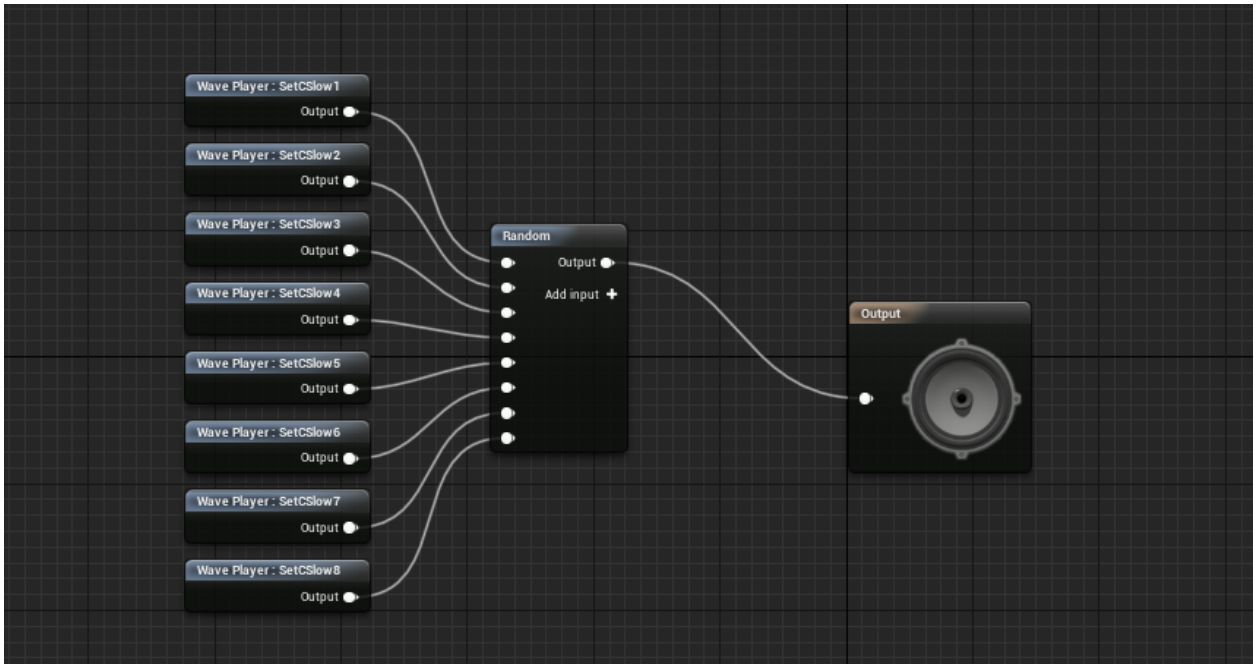


Figure 154. Audio cue.

Figure 154 shows an example of an audio cue. It contains several distinct audio clips of a person breathing. Each time the cue is played, the "Random" node shuffles the input, which selects a unique clip each time.

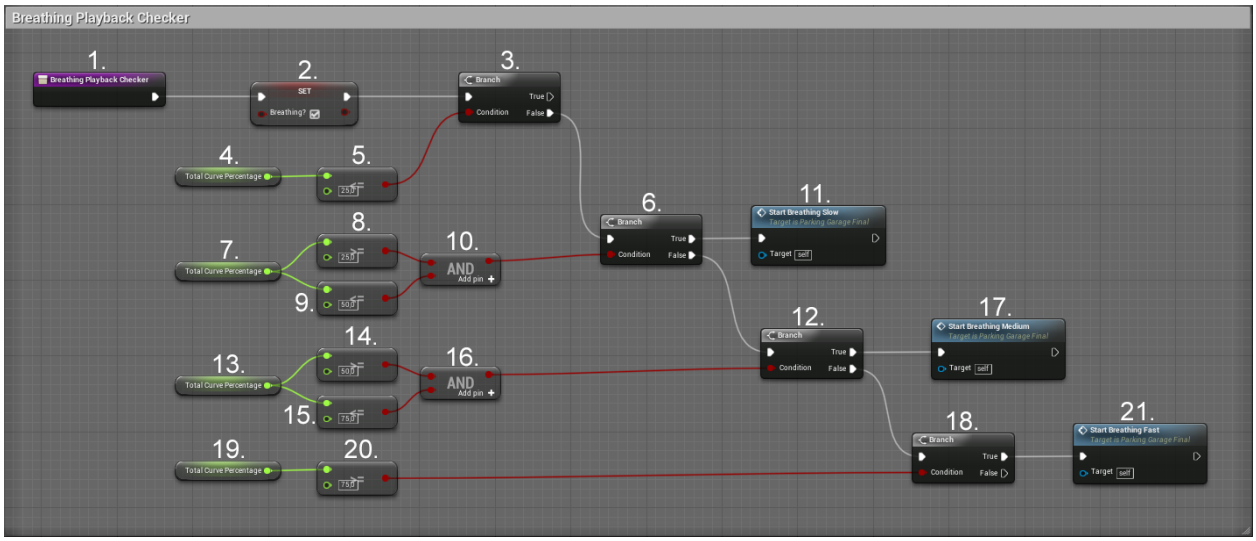


Figure 155. "Breathing Playback Checker" function.

The “Breathing Playback Checker” function shown in figure 155, determines what breathing speed blueprint needs to be executed. The “Breathing Playback Checker” node (1) launches the function, while the “Breathing?” variable is set to “true” by node (2). Four “Branch” nodes (3, 6, 12, 18) divide the blueprint into increment tests of twenty-five per cent each, with nodes (11, 17, 21) launching each respective breathing blueprints. Each “Branch” test uses the “Total Curve Percentage” variable (4, 7, 13, and 19) to determine which breathing blueprint needs to be played. Node (5) checks whether the curve percentage is below or equal to twenty-five per cent using node (5). Nodes (8) and (9) checks whether it is between twenty-five and fifty per cent, while nodes (14) and (15) checks whether it is between fifty and seventy-five per cent. Node (20) checks for any value above seventy-five per cent.

When the value is under twenty-five per cent, nothing plays. The “Start Breathing Slow” node (11) is executed when the value is between twenty-five and fifty per cent, while the “Start Breathing Medium” node (17) is launched above fifty but below seventy-five per cent. Any value above seventy-five per cent launches the “Start Breathing Fast” node (18).

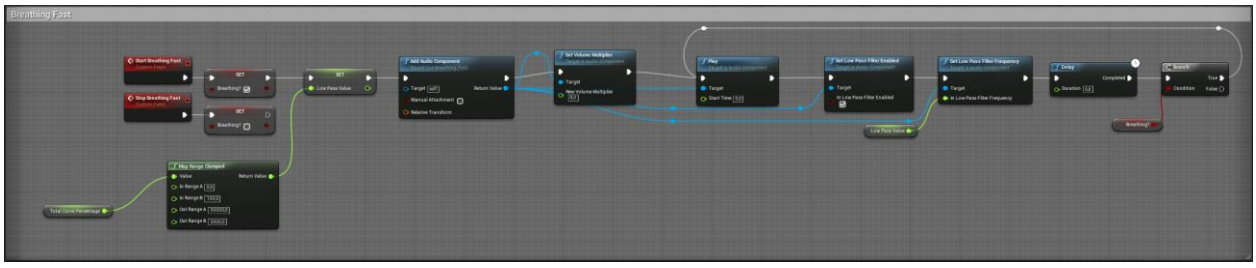


Figure 156. “Breathing Fast” blueprint.

Because the three breathing speed blueprints are the same, only one of them will be investigated carefully. Figure 156 depicts the appearance of the “Breathing Fast” blueprint.

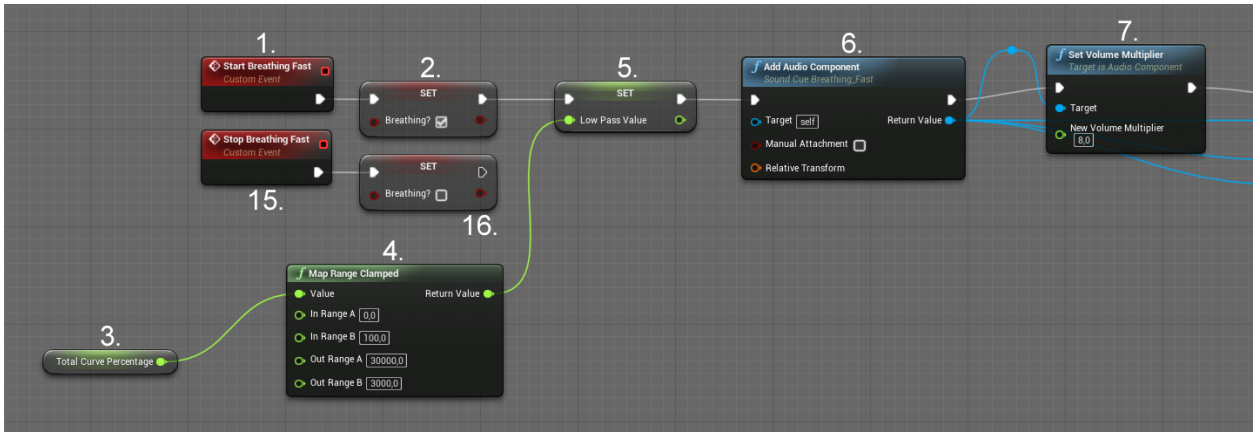


Figure 157. "Breathing Fast" blueprint – closeup part 1.

Figure 157 shows the first part of the "Breathing Fast" blueprint. The "Start Breathing Fast" node (1) is a custom event node that starts the breathing sequence. The "Breathing?" variable is set to true using a "Set" node, making it possible for the system to check whether the breathing has commenced later. The "Total Curve Percentage" node (3) retrieves a reference of the curve percentage and feeds it into "Map Range Clamped" node (4). This note takes the percentage input and converts into the range specified. If, for example, range A, with a value of one to a hundred, is converted to range B, with a range of zero to one, an input value of "25%" will convert to "0.25". The destination ranges of between "30000" and "3000" are the values that were selected for the low pass filter. The value of "30000" makes the clip sound clear while the value of "3000" makes it sound muted and disembodied.

The converted value from the "Map Range Clamped" node (4) is then stored in the "Low Pass Value" variable using a "Set" node (5), which will be used later. The "Breathing_Fast" cue is then added using the "Add Audio Component" node (6), while the audio level of this audio component is set using the "Set Volume Multiplier" node (7).

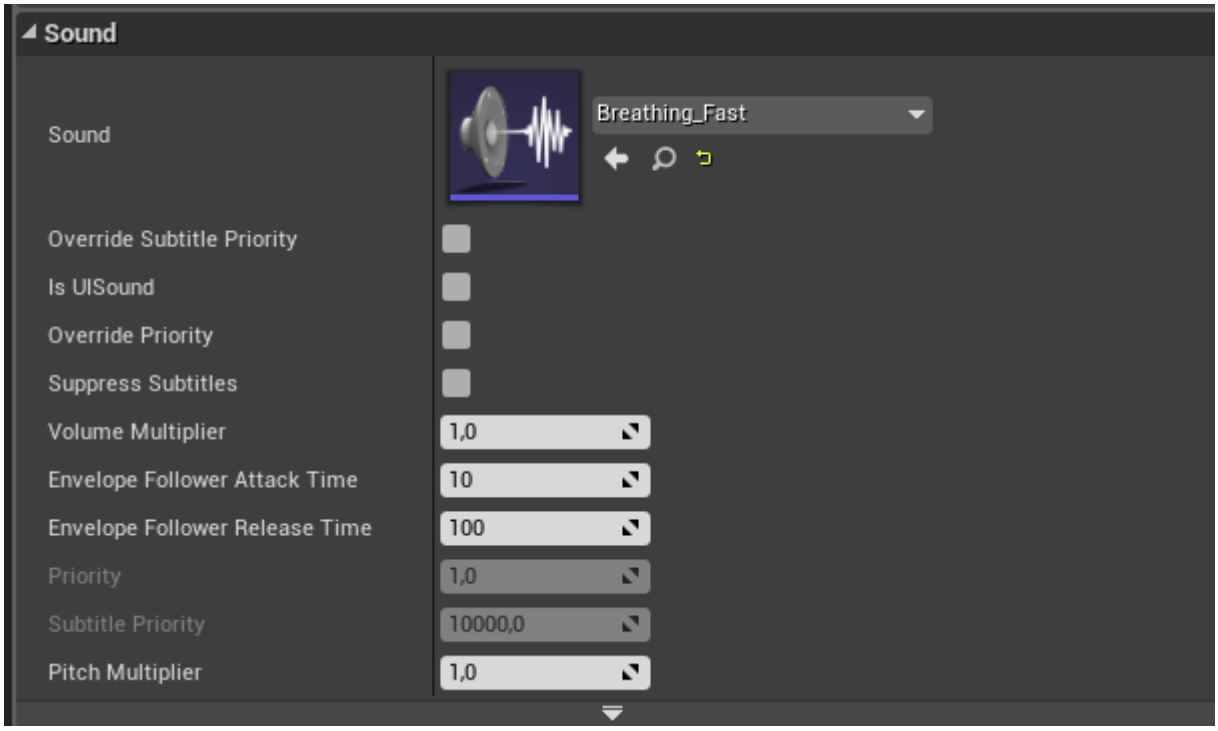


Figure 158. "Add Audio Component" node properties.

Figure 158 shows how cues are added to the "Add Audio Component" node mentioned above. Clicking the node opens several properties. The audio cue is added by clicking on the drop-down next to "Sound".

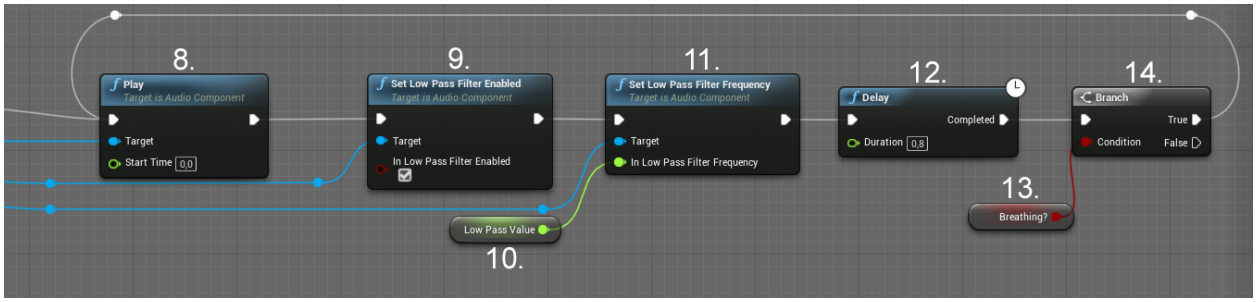


Figure 159. "Breathing Fast" blueprint – closeup part 2.

The last part of the "Breathing Fast" blueprint is depicted in figure 159. The "Play" node (8) plays the clip. Next, the "Set Low Pass Filter Enabled" node (8) enables the low pass effect for this cue. The "Set Low Pass Filter Frequency" node (11) takes the amount

stored in the “Low Pass Value” variable node (8) and sets the frequency of the low pass filter. The “Delay” node makes sure that there is an ample amount of time between each breath to make the breathing loop sound natural. The “Duration” value is set to the length of the breathing clip that is the longest within the “Breathing_Fast” cue. It should be noted that nodes (8) to (14) are linked so that the breathing cue can be looped indefinitely until the “Branch” node (14) breaks it. The “Breathing?” variable node (13) is used to determine if the loop continues or not.

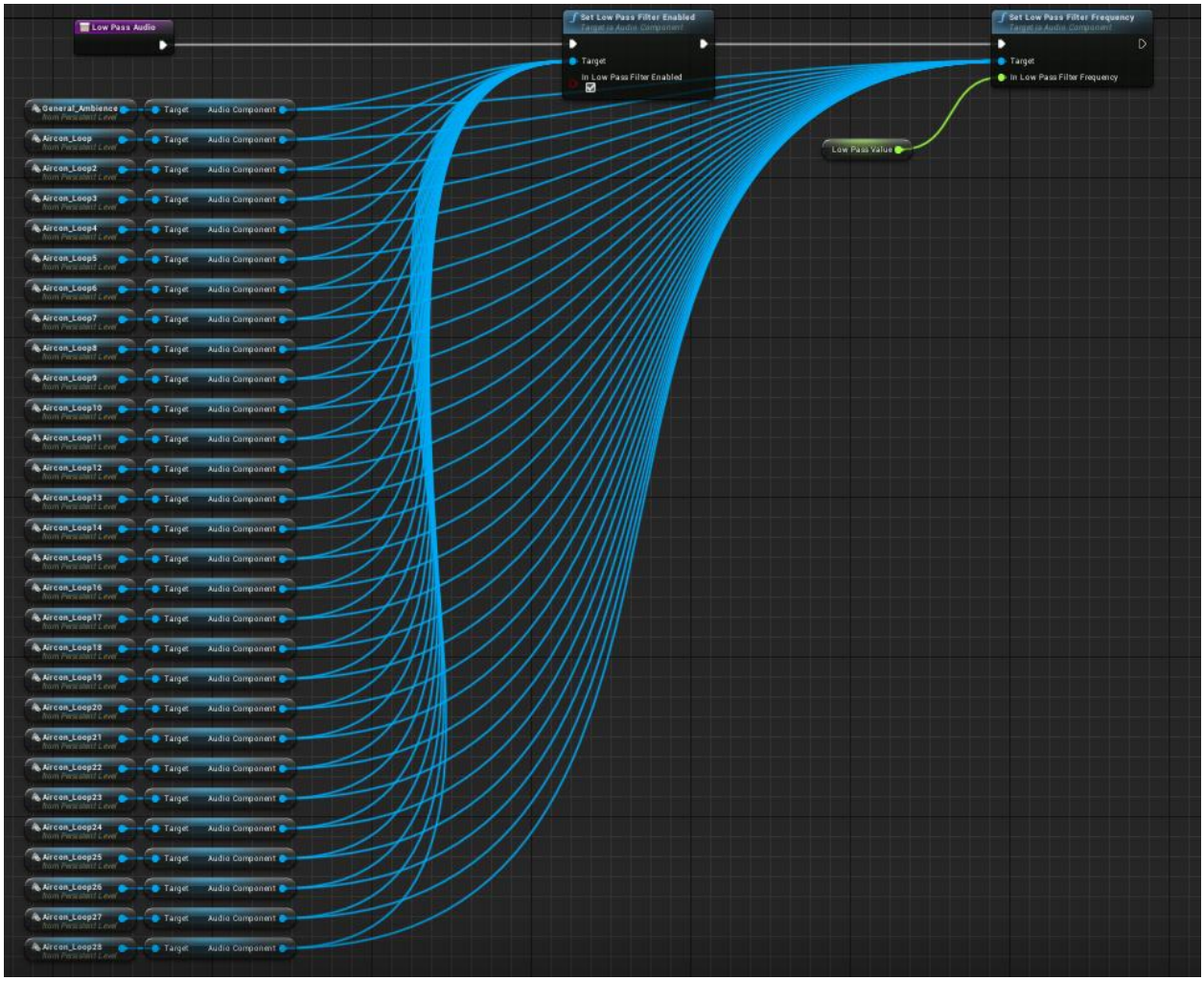


Figure 160. Level “Low Pass Audio” function.

In figure 160, the sound clips of the entire level are referenced and piped through another low pass filter. This function was left out initially, but after only processing, the breathing sounds with a low pass filter felt out of place, it was included.

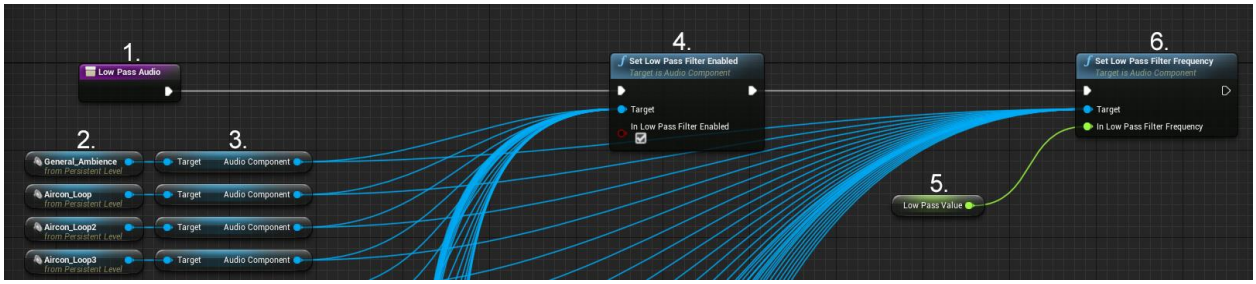


Figure 161. Level "Low Pass Audio" function closeup.

Figure 161 shows a close-up view of the function mentioned above. The "Low Pass Audio" node (1) executes the script. The "General_Ambience" node (2) is an example of one of the audio actors retrieved from the level, while the "Target" node selects the "Audio Component" for use. The "Set Low Pass Filter Enabled" node (4) enables the effect, while the "Set Low Pass Filter Frequency" node (6) sets its frequency using the value within the "Low Pass Value" variable node (5) that was established previously.

What makes this audio system unique is that it not only tracks the progression on the curve to determine what audio cues to play but also how much the low pass filter affects each cue. This operation makes the audio reflect the overall progression of the suspense curve.