Tampereen ammattikorkeakoulu

Progression Systems in Roguelite Games

Eino Kammonen

BACHELOR'S THESIS
April 2025

Degree Programme in Business Information Systems
Games Production

TIIVISTELMÄ

Tampereen ammattikorkeakoulu

Tietojenkäsittelyn tutkinto-ohjelma

Games Production

KAMMONEN, EINO:
Etenemisjärjestelmät roguelite-peleissä

Opinnäytetyö 32 sivua
Huhtikuu 2025

Tämän opinnäytetyön tarkoituksena oli suunnitella, toteuttaa ja ohjelmoida KILLBEAT-pelin etenemisjärjestelmiin sisältöä. KILLBEAT on Unityllä tehty roguelite-peli, jossa toimintoja tehdään rytmin tahtiin. Pelin tekijä on Kalma Games, joka myös toimii tämän opinnäytetyön toimeksiantajana. Kaikki opinnäytetyötä varten tehty työ on tehty osana opinnäytetyön kirjoittajan työharjoittelua Kalma Gamesilla.

Työn tavoitteena oli kehittää pelin moninaisuutta tuottamalla uusia esineitä, kuten aseita, joita pelaaja löytää pelin aikana. Tuotetun sisällön oli oltava miellyttävä käyttää, ainutlaatuista muuhun sisältöön verrattuna ja pelin tasapainon mukaisesti suunniteltu. Työhön kuului myös analysoida samanlaisia pelejä ja niiden etenemisjärjestelmiä, jotta toteutettavasta sisällöstä saatiin luotua laadukas.

Työn tuloksena pelin moninaisuus kasvoi merkittävästi. Uusien vaihtoehtojen avulla pelaaja pystyy tarkemmin harkitsemaan omaa pelityyliään ja etsimään valintojen välisiä synergioita.

Opinnäytetyöraportti on jaettu teoreettiseen ja käytännön osuuteen. Teoreettinen osuus sisältää työtä varten tehdyn analyysin niin roguelite-pelien historiasta kuin pelien etenemisjärjestelmien tärkeydestä. Käytännön osuudessa käsitellään peliin tuotettua sisältöä, suunnitteluprosessia ja toiminnallisuuksien ohjelmointia usean esimerkin avulla.

Avainsanat: roguelite, etenemisjärjestelmät, ohjelmointi, pelisuunnitelu, Unity

ABSTRACT

Tampere University of Applied Sciences

Degree Programme in Business Information Systems

Games Production

KAMMONEN, EINO:
Progression Systems in Roguelite Games

Bachelor's Thesis 32 pages
April 2025

The purpose of this thesis was to design, implement and program interesting and varied content for the progression systems of the game KILLBEAT. KILLBEAT is a rhythm-based action roguelite made in Unity and developed by Kalma Games. The game studio was also the client of this thesis.

Before the implementation itself, a part of the work was to analyze other similar games and research common design conventions. The biggest challenge was to make the content itself genuinely interesting and something the player would actually choose to use, while still not being always the most obvious choice.

The results of this work were not only almost doubling the number of passive items in the game, but also additional active items and multiple unique weapons. The replay value of the game was vastly improved as the player had been given more assets to choose from, with multiple synergies and support for different playstyles.

This report includes an analysis of progression systems in roguelite games and how to both design and implement content for them. The process of the work with multiple examples is also included.

Key words: roguelite, progression systems, programming, game design, Unity

CONTENTS

1 INTRODUCTION 6

2 ROGUELITES AND PROGRESSION SYSTEMS 7

2.1. The roguelike design principles 7

2.1.1 Roguelikes defined 7

2.1.2 The variance and utility of procedural generation 8

2.1.3 The stakes of permadeath 9

2.1.4 Procedural generation and permadeath combined 9

2.2. Roguelites, progression systems and meta progression 10

2.2.1 The importance of progression systems in games 10

2.2.2 How progression systems like meta progression shape roguelites 12

3 ROGUELITE PROGRESSION SYSTEM CONTENT DESIGN AND IMPLEMENTATION 15

3.1. Working on a roguelite's progression systems as a designer and a programmer 15

3.1.1 Passive items 15

3.1.2 Weapons 23

3.1.3 Active items 27

4 CONCLUSION 30

REFERENCES 32

GLOSSARY

Meta progression	A type of progression system in video games that permanently empowers the player in future playthroughs.
Permadeath	Short for permanent death, refers to the game design principle of the player having to start the game over from the beginning upon failure.
Procedural generation	The automatic creation of data according to rules set by an algorithm
Progression systems	Mechanical systems, such as levels and rewards, in a video game that ensure a sense of forward momentum for the player
Roguelike	A video game genre, named after the 1980 video game Rogue, which popularized the combination of procedural generation, permadeath and character progression.
Roguelite	A video game genre and the successor of the roguelike genre, the main difference being a progression system referred to as meta progression
Unity	A game engine used for video game development

1 INTRODUCTION

Progression systems are an integral part of game design. As Dave Eng puts it, the ability to interact, change and progress with games are some of the most engaging aspects of games and this is what progression systems as a whole represent (Eng 2024). Whether it's the player leveling up, choosing items or fighting enemies, progression systems cast a wide net in the ways they are designed and implemented to create a sense of forward momentum for the player to keep them engaged.

The roguelite genre of games is something that has risen in popularity over recent years. Some notable games in the genre include the top-down action game Hades (2020), the third-person action game Risk of Rain 2 (2020), the first-person shooter Roboquest (2023) and the card game Slay the Spire (2019). Just by comparing these games it is evident that they couldn't be more different from each other. What do these games have in common and what makes them specifically roguelites?

Roguelites are almost identical to the roguelike genre, which means that they more often than not follow the same design principles, those being procedural generation, character progression and permadeath, as defined by Eric Switzer (Switzer 2021). What sets one apart from the other is that roguelites feature a unique progression system design often referred to as meta progression. "When you die in a roguelite game, you take something with you into the next run" (Switzer 2021). In essence, while the player has to start over every time they die, something has changed from the previous run. Whether it's the player unlocking new powers, items or something else that couldn't be accessed in previous runs, the point is that the player is given a sense of progression even upon reaching a failure state.

2 ROGUELITES AND PROGRESSION SYSTEMS

2.1 The roguelike design principles

The reason roguelites/roguelikes are often hard to define is because they are almost never the one and only designation of genre for a game. As it was noted in the introduction chapter, roguelites can be anything from first person shooters to deck building card games. This is because, as Eric Switzer (2021) puts it, roguelites/roguelikes can be thought of as sensibilities, rather than just genres. The words roguelite or roguelike can often be seen as a sort of suffix when the genre of a game is defined, such as “a third-person action roguelite”, or “a deck builder roguelite”.

Roguelikes existed before roguelites. As Albert (2023) notes, roguelites are the result of years of iteration on roguelikes. Therefore, roguelikes should be defined before getting to its successor.

2.1.1 Roguelikes defined

There isn’t really one definite and official way that roguelikes are actually defined. There have been some attempts, such as the Berlin Interpretation, as explained by Eric James Michael Ritz (2014). However, in the very same blog he points out how outdated and even contradictory the Berlin Interpretation is. Therefore, all references to roguelikes and eventually roguelites in this paper are not based on one singular definition, but rather on how the articles, blogs and other sources referenced here interpreted them. As Cartlidge (2024) puts it, categorizing video game genres is an ongoing task unburdened by sharp boundaries and usually led by the fans of the games themselves.

The roguelike genre is named after the game Rogue, released in 1980. While not being the first of its kind, it is what popularized the game design principles people refer to when talking about roguelikes. Over the years, more roguelikes were released with their own designs and the genre had a dedicated but niche following for over 20 years. More modern roguelikes include the 2D platformer Spelunky

(2008) and the strategy game FTL: Faster Than Light (2012). This was also around the time when requirements for a game to be referred to as a roguelike began to loosen, as they were no longer exclusively role-playing games. (Josh Bycer, 2019.)

What has been the common denominator for roguelikes in the past 50 years? As Bycer (2019) proposes, the two features that are found in almost all roguelikes are procedural generation and permadeath.

2.1.2 The variance and utility of procedural generation

"Procedural generation is a method of creating things with the use of algorithms, as opposed to manually crafting those things" (Annand 2024). Instead of a developer manually creating a detailed world, a procedural generation system can be set to "paint" a whole landscape with premade assets like trees, rocks and grass. With the help of predetermined conditions, a procedural generation system creates randomized content while still following set rules. Procedural generation is a way to save time for the developer while creating a sense of freshness and randomness for the player (Annand 2024.)

Because it is assumed that players play roguelike games multiple times, procedural generation is used to create the feeling of each "run" being unique. Procedural generation can be used on almost anything from creating maps, determining loot, randomizing a selection of character upgrades for the player to choose from, to setting enemy spawns. As Bycer (2019) puts it, it is this variance of gameplay elements between runs that keeps players replaying the game even upon beating it once.

Procedural generation isn't only used in roguelikes. One of the original reasons for the use of procedural generation was to actually use less memory. As Annand (2024) iterates, developers can set up predetermined values for a procedural generation algorithm to achieve a desired outcome every time, these are referred to as "seeds". Instead of content such as the geometry of planets in the game being stored to memory, they are generated dynamically from the seed while playing.

2.1.3 The stakes of permadeath

The other common feature in roguelikes is called permadeath, which is short for permanent death. A game having permadeath means that upon reaching a failure state, which usually in roguelikes is the player character dying, the game is over. This means that there are no save states or checkpoints to go back to and the player will have to start the game over from the very beginning. As Bellow (2024) explains, the player not only loses their character, but all progress they have made, from items to skills and even to the story.

The threat of losing all progress alters the ways the player interacts with the game. Because death is permanent, each choice has more weight, and the player has to carefully consider each action they take. The heightened stakes and high-risk-high-reward style of gameplay that can come from games with permadeath can be very exciting for players. The only thing the player can take over from a failed run to the next is the lessons they have learned. Mechanics and information such as item interactions, enemy behaviour patterns and optimal strategies are intended for the player to learn over multiple playthroughs. Learning leads to mastery and finally giving the player the ability to complete the game in a single run from the beginning to end, which leads to a great sense of accomplishment (Paul Bellow 2024.)

2.1.4 Procedural generation and permadeath combined

As put together by Bellow (2024), procedural generation and permadeath work in tandem to create a unique experience every time the player dies and has to start over. Not only does this combination increase the variance of roguelikes, but it also increases the difficulty.

Because each run is different, the player can’t just trial and error their way through the exact steps they need to take to reach the end credits. This way the player interacts with the challenges of the game more reactively in order to keep the run going, while still considering the risk and reward of each action they take when it

comes to the long-term of the playthrough. As noted by Bellow (2023), the unpredictability of this design combination creates a sense meaningful consequence and therefore is a defining aspect of roguelikes.

Conversely, roguelikes can be seen as difficult and frustrating, because of how punishing failure can be. Having to start over every time you lose might feel too extreme for some players. “Roguelike games can be incredibly rewarding to play, but the only progression you can make in a roguelike game is in your own skill” (Switzer 2021). While progression exists in roguelikes inside the ongoing run, it is all gone upon failure. Because progression in between runs isn’t something perceivable, such as a checkpoint or a level, for some players it might feel like there exists no progression at all. As framed by Switzer (2021), roguelikes come with a level of friction that in the end isn’t very accessible for casual players, which make the majority of the audience.

2.2 Roguelites, progression systems and meta progression

As mentioned before, roguelites are an iteration of roguelikes and incredibly similar to the point that they are often treated as the same thing. The design principles of procedural generation and permadeath exist in both, as does the assumption that the player will play through the game multiple times. However, roguelites have one design difference that makes distinguishing one from the other necessary. Roguelites are essentially the result of trying to make roguelikes more accessible, by reducing the amount of friction that comes from the player losing all their progress whenever they fail. As mentioned in the introduction, this design technique is referred to as meta progression. Before getting to how and why meta progression actually works, defining progression systems in games as a whole should come first.

2.2.1 The importance of progression systems in games

Progression systems are any mechanical system in a game that allows the player to pursue and track goals and get rewarded for achieving them with things like new content, items and abilities. They exist to give the player a sense of progression to keep them engaged and playing the game. They also reinforce the

player’s actions by assisting them to achieve mastery through the core game play loop (Eng 2024.)

Eng (2024) categorizes common progression systems as the following:

- Horizontal progression systems, which give the player choices of different gameplay elements, such as weapons and spells, giving them the freedom of choice and the ability to strategize.
- Vertical progression systems, which are measured by scale, such as health and the strength of abilities and characters, giving player both the sense of getting stronger but also the challenges getting more difficult the more they progress.
- Cyclical progression systems, which cycle elements categorized as hard and soft cycles at a consistent and predictable rate, until the end of the cycle is reached and then started anew. Hard cycles include things like increasing player power and challenge that reset upon permadeath, while soft cycles include things like item decay that resets upon the item breaking and the player having to find a new one.
- Player character-based progression systems, which is the progression of the player’s character. The player character improves through things like gaining experience or finding items by fighting enemies to increase their character’s power. The player character can improve with things such as increases to statistics like health or new abilities. The player can also be presented with choices, such as a skill tree, where they can allocate their improvements as they see fit, increasing player agency.
- Mission, world and level progression systems, which are the advancement of levels or stages with the help of goals like missions or quests.
- Resource progression systems, which allow the player to gain, spend and balance resources like gold to acquire things, like new items or other game components.
- Narrative progression systems, where the story of the game unfolds as the player progresses.
- Legacy progression systems, where the player is rewarded upon starting a new playthrough with improvements that they didn’t have access to before.

- Time-based progression systems, where progression is tied to the amount of time spent playing the game, meaning the player is rewarded the more they play.
- Luck-based progression systems, where progression is based on chance.

Progression systems aren't mutually exclusive and more often than not games are made up of a combination of them. For example, a player character-based progression system can be combined with a luck-based progression system in a way that the player gets to choose a new skill from a randomized selection instead of a preset skill tree upon leveling up.

As Bycer (2014) discerns, player progression exists separately from progression systems, referring to the skills and knowledge the player amasses while playing and learning the ins and outs of the game. Together the player's personal progression and the game's mechanical progression systems create a feedback loop where the player is encouraged to learn and improve and then rewarded for it.

Ultimately the purpose of progression systems is to make the player feel like the time spent playing is worth their while. With progression systems content can be spaced out in order to not overwhelm the player and to make sure that their experience changes over time and isn't static. If the player gets stuck or straight up fails, giving them other ways to progress can be encouraging to continue and overcome those obstacles later. Well thought out progression systems ensure a sense of forward momentum no matter what, which is paramount to keeping the player engaged (Bycer 2014.)

2.2.2 How progression systems like meta progression shape roguelites

As it was established before, different roguelikes and by extension roguelites can be very varied, which means that all types of progression systems can be applied depending on the design. However, there are some common design strategies.

Character-based, horizontal and cyclical progression systems are often at the centre of the design in something called build construction. As Torick (2024) elaborates, build construction means that with each gameplay choice, such as the character they play, the weapon they use and the skills they choose, the player builds their character to overcome the challenges of the game. The player is encouraged to acquire more power as games like roguelites often scale up the difficulty at a static rate with obstacles like enemies getting stronger and harder.

With new choices the player may seek synergies with the choices they have already made or make them as an investment for potential future synergies. For example, the player uses a burning weapon and then chooses a skill that increases burn damage, making it a synergistic choice as the new choice empowers something the player already has, leading to increased player power and therefore making the game easier to complete. With clever choices the player may get strong enough to push past the scaling difficulty of the game, giving the player a satisfying sense of power.

Luck-based progression systems and procedural generation go hand in hand in roguelites. In order to make each run different, character progression is often rigged to the procedural generation system instead of the player having static choices that are the same in each run. For example, the player gets a random selection of skills upon leveling up, enemies drop random gear for the player character to equip or the player has to choose which room to enter based on randomized room rewards. While procedural generation isn't truly random, instead being based on algorithms as established before, there is still an element of luck-based progression. As explained by Torick (2024), the player can get a stroke of luck in the middle of the run, such as acquiring a synergistic combination of weapons and skills or getting an incredibly rare and powerful item, allowing them to push against the difficulty curve and progress more easily until the game catches up with things like enemy scaling. Conversely, this can have an opposite effect where the player gets unlucky and doesn't get the synergies they need, meaning they might fall behind the difficulty curve. Making your own luck is an integral part of roguelites, both figuratively and literally. Roguelites often feature ways to manipulate the odds to more favourable rewards through in game

choices and making do with what you have is an important part of the learning curve.

The progression system that sets roguelites apart from roguelikes is called meta progression, which would fall under the category of legacy progression systems. While the idea of permadeath still persists, failure doesn't mean a complete reset of all progression. The point of meta progression systems is to empower the player permanently in future runs, such as giving the player new characters and weapons to use, increased health for the player character, more starting currency or increased odds to get rare items. Meta progression is often advanced through something found inside the game loop, for example a separate currency that the player gets from trading off power ups in the current run, as rewards based on how far the player has gotten or by pursuing and completing predetermined challenges, such as beating the game with a specific character. Meta progression is often presented visually in between runs in either a menu or a hub area where the player gets to choose what to unlock, such as different characters or between distinct and sometimes even mutually exclusive power ups (Switzer 2021.)

As established before, the biggest issues with roguelikes are the inaccessibility of losing all your progress upon failure and having to mostly rely on your own skill, it's these issues of accessibility that roguelites try to fix. With meta progression systems, the player gains a sense of progression and improvement even upon failure. As exclaimed by Torick (2021), roguelites are often designed in a way that meta progression is paramount to victory, meaning that beating the game in the first few runs is almost impossible, ensuring that the player gets to see, interact with and feel the effects of meta progression before winning their first run. Meta progression provides goals outside of just beating the game, which completely shifts the player's focus from reaching the end in a single run to instead always looking forward to the next one.

3 ROGUELITE PROGRESSION SYSTEM CONTENT DESIGN AND IMPLEMENTATION

3.1 Working on a roguelite's progression systems as a designer and a programmer

The theoretical part of analysing and establishing the importance of progression systems in roguelites was in part to lay the groundwork for why spending developer time and resources on them is important. Ultimately this was the reason why the client of this thesis wanted a lot of the work done on designing and implementing content for the progression systems of the rhythm action roguelite game KILLBEAT they were developing. The content to be created were for different character progression systems, namely passive items, weapons and active items, each furthering contrasting design goals. The work included both the design and implementation of the content. While the functionality for the progression systems themselves had already been done long before the start of this work, they were still something that had to be learned from inside out in order to navigate and adjust them as needed to do the work sufficiently. The following segments include the work done step by step with multiple examples for each of the three previously mentioned progression system categories.

3.1.1 Passive items

Up first was work on what are called passive items in the game. Passive items in this roguelite are one of the two main forms of character progression. Upon starting a level, the game procedurally generates a map comprised of multiple premade rooms. Upon entering a room, the player has to complete an objective inside to acquire its reward. The most common types of rooms are combat rooms, in which the player is locked inside until they defeat all enemies, and puzzle rooms, which provide rewards upon completing a puzzle. There also exist unique event rooms that further the story of the game and boss rooms that are much like combat rooms, but the player has to fight a strong boss enemy to get rewards and more importantly, get access to the next level. The main way of getting pas-

sive items is as room rewards, in which a player gets to choose one from a randomized selection of three passive items. They can also be bought from a shop in each level with its own randomized selection for currency that is found in rooms.

Passive items in the game refer to boons that passively boost the player's power in different ways. They modify the player character's statistics and performable actions in ways that, for example, enable them to sustain more damage, defeat enemies faster or encourage and reward them for doing specific deeds. Passive items range from simple numerical changes, such as increased health or an additional weapon slot for the player character, to more advanced upgrades that require a certain trigger, such as causing a shockwave whenever the player reloads their weapon or resurrecting them from death one time. There isn't a limit to the number of passive items the player can have and none of them are mutually exclusive, which means that the player is encouraged to seek out as many as possible before attempting to beat the boss of a level. Not every item is equal in power, instead being balanced around each having its own rarity, with stronger items being rarer and vice versa. The rarities in KILLBEAT are from the most common to the least: common, rare and epic. Item rarity goes hand in hand with procedural generation and enables strokes of luck and other design goals that come with luck-based progression. The concept of synergies is also deliberately baked into the design of the items, allowing players to pursue specific playstyles.

One of the biggest questions before the work started on these items was how they are actually made in similar games. As was established before, variety in character building is what makes or breaks a roguelite, which means that the more successful ones should have a significantly sized pool of choices. For example, Binding of Isaac: Rebirth (2014) has hundreds of what are basically the equivalent of passive items in KILLBEAT. How does a designer go about designing so many items that are both fun to use and different enough from each other and how does a programmer implement them in a reasonable amount of time?

There were some ideas for passive items in the backlog from the time they were implemented originally. Working on implementing these items first was in order to understand how to work the systems. Upon getting used to the process of

production and completing the backlog it was time to start coming up with new and original ideas. Coming up with ideas for gameplay elements like passive items is mostly a creative process. The goal is to give players interesting and fun ways to alter their gameplay and make them feel powerful, while still having the game retain a sense of challenge.

To come up with items that have a numerical change to the player character, such as increasing the amount of ammo the player can carry, the designer should look at every single type of data in the game that affects the gameplay in some way. Then it should be figured out which ones can be numerically changed to empower the player without breaking the game in the process. As an example, KILLBEAT has multiple items that have a percentage chance of activating their effect, so an item was designed that doubles the chances of all luck-based effects, making activations more frequent and creating a synergy between all other items it affects.

In order to design items with a trigger, a designer has to look for potential triggers in the gameplay. More obvious triggers include actions the player character can take, such as shooting or dashing, but in the end pretty much anything that happens in the game can be used as one, such as the player character entering a new room or an enemy being defeated. Coming up with effects for triggers often requires new functionality. A status effect system had been made for KILLBEAT for the sole purpose of making effects for items and weapons that debilitate enemies in different ways, such as setting them on fire or shocking them. A system like this goes hand in hand with triggers, for example passive items were made for each status effect in the game that inflict their effects on enemies when they are hit by weapons. However, not every passive item with a triggered effect needs new functionality. Finding clever ways to reuse already made functions is an efficient use of time. For example, an item was designed that allows for shots that hit enemies to ricochet to new targets, which uses the same shooting functionality that all weapons already use.

Most of the time when coming up with ideas for passive items was spent on finding triggers and designing different effects for them, as items with numerical changes were already aplenty. Taking inspiration from other games is one way

to come up with different items. For example, chain lightning type effects in other games were used as an inspiration for a similar passive item. This item evolved from a single item to its own status effect, called shock, which has since been used in multiple other passive items and even a couple of weapons. Simply working on the game and testing it is also a way of getting inspired. While programming and creating new systems it's common to think for new applications for what's already been created. Extensively playing your own game lets you see what possible areas aren't covered with sufficient amount of progression mechanics. Gameplay systems like passive items are also an excellent way to promote and encourage players to use systems that they might otherwise not realize or remember even existing. While testing the game, a mechanic called Freestyle Charges was barely used by some players, which are a resource that allows the player to shoot their weapon out of rhythm. The primary goal of designing passive items that made Freestyle Charges stronger and allowing to use them more often was to encourage and ease the players into using the mechanic itself. Most of the time whenever a new idea for a passive item came up, it was while was doing something else, was it programming or working on a different feature, playing and testing the game or even while playing other games. When getting an inspiration, it should be documented as soon as possible as they can be rather easy to forget. In this case a Notepad file was kept updated for all the passive item designs part of this work.

When starting work on a new batch of passive items, or anything else requiring some level of creativity, the ideas would first be listed to the rest of the team. Starting work on new features immediately upon getting new ideas without getting the approval of the team would be rather foolish as they are the client. Consulting the team that built the game from the ground up and designed everything else can only help in overcoming design problems and other challenges. In the end the CEO of the company in KILLBEAT's case has the final say in everything that gets added to the game. When designing gameplay elements in this way it should be assumed that not everything you come up with fits the game's core design or is rejected for some other reason. This why one shouldn't get too invested in their own designs and also why one can't have too many ideas as not all of them will reach the next step of the development process.

From this point on we'll take one passive item that's part of this work, named Reloading Finisher, and go through the process of how it was implemented. For context, Reloading Finisher reloads the player's currently equipped weapon whenever they execute a finisher. Whenever an enemy's current health goes below a certain threshold, it is subsequently stunned and opened up for a finisher. Pressing the input for a finisher while there is an enemy opened up for one on screen makes the player character dash to that enemy and execute it, killing it without the need to reduce the rest of said enemy's health to zero. Finisher is a core mechanic of the game and a very powerful tool on its own, which is why Reloading Finisher has a rarity of rare, as it makes an already strong mechanic even stronger. The goal of the item is to create a satisfying loop of the player first emptying their weapon's magazine, following it up with a finisher and repeat, as the need for reloading is skipped. Weapons with small magazines get more value out of the item, making them more comfortable to use as long as they are able to perform finishers consistently.

KILLBEAT is setup to read data from a Google Sheets file. The file contains tables for different types of gameplay systems with modifiable data, including passive items. Starting work on a new passive item starts from filling out new rows in two different tables, one for identification of the item and the other for modifiable data of said item. Columns in the second table include a checkbox for disabling the item, the item's rarity and name of the script the passive item will be using. There are also columns for data that pertain to the functionality of the item, including common values between all passive items, such as activation chance, effect value and duration, with a custom column for more unique values that can be used as needed. The developers have set up Unity in a way that it can then be prompted to read the file and automatically generate and create files and other basic functionality that is shared between all passive items. For example, whenever a new passive item is generated through the file, it is automatically added to the pool of items that can drop in the game, reducing the amount of busy work that would otherwise be needed. Setting up a game engine to read external files and generate and update content based on them is not only extremely efficient, but it also allows modifying values, such as the rarity of a passive item, of said content without having to touch the code itself. Reloading Finisher is rather light

on modifiable data, with the only one being its rarity. Filling out the columns and activating the generation of passive items automatically adds it to the game as a unique passive item with its own rarity that can be found and picked up by the player, all that is missing is the functionality of the item itself.

As the only functionality is to call the function for reloading of the player's currently equipped weapon, this passive item doesn't actually need any code in a script of its own. In this case, it was needed to find the exact point in the functionality of the player's finisher state script where it would make most sense to trigger the effects of the item. KILLBEAT uses state machines, with the action of the player character dashing towards a target opened up for a finisher and then executing one being its own state. The state script's update function checks constantly for when the player character reaches the target and can confirm the finisher. Adding the functionality precisely to the point where the finisher itself is confirmed is important to prevent the effect of the item from activating when it shouldn't, as the dash portion of the finisher state can be cancelled for a variety of reasons or the target can die from a lingering source of damage, in which case no finisher can be confirmed.

As can be seen from the following picture, most of the functionality here is just calling already existing functions. The finisher state gets references of both the player and the target enemy when the activation of the state is called. Then those references are used to call for appropriate functions for different checks of if-statements, such as confirming the finisher itself. A finisher is guaranteed when it is confirmed that the target enemy of the current finisher state is still valid, meaning that it is both not destroyed and still opened up for a finisher. Upon confirmation, the script has reached the trigger point of the item, meaning it is safe to finally call for the effect itself. Every time the player successfully executes a finisher; the state script first checks if the player has the Reloading Finisher item currently in their possession and then if their currently equipped weapon can be reloaded. If both checks return a value of true, then the player's currently equipped weapon's reload function is called, concluding the effect of the passive item.

```
if (!m_target.IsDestroyed() && m_target.IsStunned())
{
    if (m_player.PassiveItemHandler.HasItem(PassiveItemID.GROUP_FINISHER))...
    else
    {
        DoFinisherToEnemy(m_target);
    }

    if (m_player.PassiveItemHandler.HasItem(PassiveItemID.RELOADING_FINISHER) &&
      m_player.WeaponHandler.CurrentWeapon.CanReload())

        m_player.WeaponHandler.CurrentWeapon.Reload(AccuracyRating.PERFECT);
}
```

Picture 1: The functionality for confirming a finisher with the functionality of the Reloading Finisher passive item inside it.

Since the previously highlighted passive item didn't require a script of its own, one more that does will be featured, in order to showcase more explicitly what generating a passive item from the Google Sheet does in KILLBEAT's case. The second passive item featured is called No Pain No Gain, its effect being upon the player character taking damage from any source they get a fixed amount of currency, called Tickets. Tickets are part of the game's resource progression system, which are used to purchase items, weapons and other resources from a shop that can be found in each level. The purpose of this passive item is to offer some gain for something that is usually only a negative consequence and enable daring players to use their health as a resource.

All passive item scripts in KILLBEAT inherit the same base script that includes all the basic functionality of a passive item. The base script gets any necessary references pertaining to the functionality of the item, such as the player character, sets up the inheriting script to get its corresponding data from the Google Sheets file and calls for the item's enabling and disabling functions as needed.

As the trigger point of the item is whenever the player character takes damage, the activation of the effect can be set to be called by an event in the player character's health script, as the reference of the player character will never change. In short, functions can be set to subscribe to an event and when said event is reached in the code while the game is running, the subscribed functions are then subsequently called. As can be seen from the picture, the activation function of

the item is set to subscribe to an event in the player character's health script, which activates whenever damage is taken.

```
public class MBPassiveNoPainNoGain : MBPassiveItemBase
{
    2 references
    public override void StartPassiveEffect()
    {
        m_player.m_health.m_damageTakenCallback += Activate;
    }

    3 references
    public override void EndPassiveEffect()
    {
        m_player.m_health.m_damageTakenCallback -= Activate;
    }

    2 references
    private void Activate(object _object, Vector3 _direction)
    {
        m_player.m_runData.m_money += (int)m_config.m_effectValue;
        Helpers.CreateFloaterText(m_player.transform, "+" + (int)m_config.m_effectValue);
    }
}
```

Picture 2: The script of the passive item No Pain No Gain.

Creating the effect functionality of the item can be rather simple because of excellent foresight by the developers when originally creating these systems. As the base script already has a reference to the player character, it also has one to the player's currency, which then can be simply increased by the amount set in the Google Sheets file. As the item only has one functionality affecting data point, it can be set to effect value column of the item in the file. Reference to the item's row in the Google Sheets file is named m_config in the script, from which all of its columns can be referenced. As a final touch the item calls for a simple already premade UI text function that shows the amount of currency gained as a way of showing the player that the item's effect has activated.

Using these methods a batch of 32 new passive items were added to the game, almost doubling the number of passive items. The goal was not only to bolster the pool of items but also create new and fun ways to play the game. The variety of each run improved significantly as the player is less likely to find the same items over and over again. There also now exist enough synergies for the player to seek different playstyles, such as focusing on triggering status effects or making a gun shoot as many bullets as possible with one shot. Setting up a pipeline and a process where a lot of the menial parts of adding new mechanical content

to the game is streamlined and automated is how a programmer can produce content quickly. Designing said content requires not only creativity, but also knowledge on how the systems used in the creation process work, as that knowledge allows for clever reuse of work that's already done.

3.1.2 Weapons

Weapons are the second main form of character progression in KILLBEAT. Weapons are what the player mainly uses to defeat enemies. As KILLBEAT is a rhythm-based game, the player has to shoot their weapon to the beat to deal sufficient damage. Unlike passive items, which do not have a limit, the player has to choose which ones they carry with them. Types of weapons in the game include weapons that shoot a single bullet per shot, such as pistols and sniper rifles, weapons that shoot in bursts, such as assault rifles and submachine guns, weapons that shoot multiple bullets at once in an arch, such as shotguns and special weapons with unique firing patters, such as the Explosive Crossbow, which shoots a devastating explosive crossbow bolt. Each weapon has their own statistics, such as damage, range, spread, kickback and magazine size and maximum ammo carried. Much like passive items, weapons are further balanced by rarities, with weapons having better stats the rarer they are. In fact, there exists multiple versions of the same weapons in different rarities, such as there being a common, a rare and an epic version of the pistol. Weapons are found by the same ways as passive items, meaning as room rewards in a randomized selection or in shops in return for currency.

Unlike the work with passive items, the main goal here is not to massively increase the pool of weapons, as there already exists a sufficient number of regular weapons. Instead, what the client wanted was more special weapons, like the Explosive Crossbow, with unique firing patterns. The goal is to make fun and powerful weapons for higher rarities that visibly differentiate from your standard weapon in order to excite the player whenever they stumble upon one.

Designing weapons for a rhythm-based game like KILLBEAT can be quite different from your regular shooter. As the player is required to shoot their weapon to the beat, differentiating fire rates for weapons can't really be done, as each

weapon’s viable fire rate or how often the player can press the input for shooting is whatever the beats per minute currently is. What can be changed however is what happens when the input for shooting is pressed. Before this work the developers had already overcome the burden of not having the ability to modify fire rates by having weapons either shoot multiple bullets at once, such as shotguns, shoot bullets in short bursts, such as submachine guns or singular bullets that pierce enemies, such as sniper rifles. Also, if a weapon should shoot slower, its magazine size can be decreased in order to force the player to reload more often and vice versa.

The work on weapons done as a part of this thesis was mostly finding unique and fun ways to change and modify what happens when the input for shooting is pressed. The design of these special weapons was very similar to passive items, as the act of shooting a shot can be thought of as a trigger and then anything coming out of the gun as its effect. It's because of this that designs for weapons for each status effect in the game came naturally, as passive items with similar effects already existed. How a status effect weapon differentiates from its passive item counterpart is that the activation of its status effect is guaranteed for each hit. Not every status effect has a regular firing pattern either, such as the Icicle Launcher, which shoots piercing projectiles that can hit multiple enemies with one shot and apply the chill status effect for each enemy hit. Before the start of the work the CEO requested weapons with what can be called a combo firing pattern, meaning that each shot of a weapon perfectly timed to the beat builds up an effect of some kind, such as every third shot firing in a circle pattern or with each shot the number of bullets shot per firing of the weapon is increased.

From this point on the process is rather similar to passive items. After the designs are approved by the team, adding the weapons’ parameters and other data to the Google Sheet’s weapons section and hitting generate in Unity lays lots of the groundwork in making a functional weapon. Unlike passive items, almost every data column of a weapon should be filled as their basic functionality requires values for things like stability, distance and spread. While weapons have default values for their statistics, adjusting these values is the main way to balance and distinguish them from each other. As weapons can have different firing modes, they have to inherit different scripts for their basic functionality, such as a shotgun

functionality script for shotguns. As with any special weapon that has a unique firing pattern, a new functionality script has to be created, much like a passive item. All functionality scripts should still inherit the same weapon base script, like it was with passive items.

The weapon featured as part of this thesis is called Railgun. Railgun is a special weapon with a combo firing pattern, that fires an all-piercing bullet every three shots triggering a chaining ricochet shot on all enemies hit. For the regular shots that do not trigger the combo's effect, the weapon fires standard bullets that aren't particularly strong. The weapon's strength lies in rewarding the player for keeping up the three-shot-combo with a devastating shot that first hits all enemies in the bullet's trajectory and then activates a chain reaction of new shots, that target other nearby enemies and ricochet once more from the new hits. It should be noted that a new ricochet instance is created for every enemy hit by the original shot, meaning that while it doesn't make much sense as far as realism is concerned, shooting into a crowd of enemies creates a satisfying and hectic multi-hit chain reaction. Railgun gets custom data from the Google Sheets file for the number of shots required to trigger the combo, damage of the ricocheting shot, how many times the initial ricochets can trigger new ricochets and the increased amount of pierce of the triggered shot.

The railgun's script inherits a combo weapon functionality script that tracks the combo's counter and calls the combo effect's activation function, while still inheriting the base weapon functionality script. The following picture shows the function for the ricochet effect that gets called for each enemy hit by the last shot of the combo pattern. The function needs references to the target that was hit by the original shot, previous target if the shot has ricocheted more than once to prevent it going back and forth between targets and the number of ricochets left. First the script checks whether there are any ricochets left, if not the ricochet loop stops. Then a reference to the position of the initial hit is taken as an origin point for the upcoming ricochet. Following that the script calls a method for getting a list of all enemies in a specific range, which in this instance is the range of the weapon from the origin point. All unnecessary or invalid enemies are then subsequently removed from the list and a random enemy is chosen as a target for the ricochet. Then the trajectory of the ricochet is calculated from the origin point

of the shot to the target and the function for shooting a projectile, that all weapons use, is called with the necessary parameters to launch the ricochet projectile from the origin point towards the target. The shooting function returns the data of all hit enemies that is then used to shoot new ricochets, calling the very same method and creating a loop of ricochets up until they run out.

```
private void ShootRicochet(MBEntityBase _originTarget, MBEntityBase _previousTarget, int _ricochetsLeft)
{
    if (_ricochetsLeft <= 0)
        return;

    Vector3 ricochetPos = _originTarget.transform.position;
    ricochetPos.y = 1;

    List<MBEnemyBase> enemies = new();
    EnemyManager.GetEnemiesInRange(m_player.m_currentRoom, enemies, ricochetPos, m_comboProjectile.GetMaxDistance(m_player.m_statChanges));

    enemies.RemoveAll(e => e == _originTarget || e == _previousTarget || e == null || !e.IsAlive());
    if (enemies.Count <= 0)
        return;

    int rng = Random.Range(0, enemies.Count);
    MBEnemyBase ricochetTarget = enemies[rng];

    if (ricochetTarget == null)
        return;

    Vector3 targetPos = ricochetTarget.transform.position;
    targetPos.y = 1;
    Vector3 dir = (targetPos - ricochetPos).normalized;

    ShootVisualizationData shootVisualizationData = new(EffectType.TRAIL_VFX_SMALL, AccuracyRating.PERFECT, false);
    HitData hitData = CombatHelpers.Shoot(m_player, m_comboProjectile, ricochetPos, ricochetPos, dir, shootVisualizationData, false, m_config.m_id);

    foreach (MBEntityBase entity in hitData.m_entities)
    {
        if (entity is not MBEnemyBase || entity == null)
            continue;

        ShootRicochet(entity, _originTarget, _ricochetsLeft - 1);
    }
}
```

Picture 3: The Railgun weapon functionality script’s ricochet function.

A total of nine unique weapons were designed and implemented as a part of this work. All nine weapons followed a similar process as described above in order to create fun and powerful weapons that are on the rarer side. While not nearly as many weapons were created compared to the batch of passive items, it should be noted that almost all of them required new functionality. As each weapon is unique from one another, they require more visual work, such as new models, effects and animations, from other developers. All in all, the goal here was to add gameplay elements in the form of weapons with their own sense of self, something for the player to discover and even potentially get attached to as they hopefully find at least one they enjoy using. As shooting is the what the player does most of the time in KILLBEAT, creating varied and exciting choices for how to do it is an important priority.

3.1.3 Active items

Active items are the third and final progression system in KILLBEAT. They are actually the main form of meta progression of the game. Active items are acquired by the way of unlocking challenge levels by playing and completing them outside a normal run. Before the start of each run, the player can choose one active item for the whole run. As for how they work in normal gameplay, active items have their own input key for activating their effects, hence the name. After using an active item, the player can't just immediately use it again, instead having to charge it up by fighting enemies.

Because active items can't be used all the time, they can be made very powerful. Other than that, they are very similar to passive items in how they work. The trigger point for active items is their input key, which gives a great deal of control for the player to manage how to use the item. Active items vary, but often do something that can affect multiple enemies in a large area. The purpose of active items is to be used selectively as tools against tough enemies. KILLBEAT is designed in a way that, unlike weapons, active items are in no way required to beat the game, making them excellent for meta progression. As active items are extremely strong, unlocking and using them makes the game easier, but the player can still forgo using them if they want an extra challenge. While active items exist to make the player more powerful, they are also yet another source of variance in a run.

While designing active items is very similar to passive items, a couple of things have to be kept in mind. As active items aren't categorized by rarities like weapons or passive items, they don't have to adhere to a specific tier of power. If an active item is extremely strong it can be put behind a harder challenge level, but other than that they don't have to be in perfect balance. In fact, some of them should be made somewhat overpowered in order to make the player feel like doing the meta progression challenges is worth their time. The other balancing factor of active items is for how long it takes to charge them in order to be used again. Using these factors the power of an active item should walk a tightrope of being very strong, something the player looks forward to using and wants to charge up, while not totally breaking the balance of the game where they are the

only viable path to victory. The goal is to give players an occasional and satisfying burst of power that complements the rest of the gameplay at opportune moments.

The highlighted active item is called Chaos Orb. Activating the item spawns in an object that shoots out beams at random nearby enemies triggering a random status effect for each hit. The item is a guaranteed way to activate and spread status effects in a room filled with enemies. Status effects can be extremely strong, especially if the player gets multiple status effect affecting passive items, meaning that a guaranteed way to spread them enables a build focusing on status effects. Yet an active item shouldn't require the player getting specific items that they aren't even guaranteed to get for it to be viable, however in this case status effects are already strong enough on their own. Status effects are so strong in fact that Chaos Orb had to be limited to one beam and therefore status effect per enemy.

When it comes to using the Google Sheets file, it's pretty much the same deal here as with the previous two progression systems. The one common data point between all active item is their charge rate, meaning how fast the item can be used again. Other than that, the custom data column is used for all necessary data, as usually active item designs are more unique from each other. In Chaos Orb's case values for the duration, the maximum range of the beams and the parameters of the electrify status effect it triggers are needed.

How Chaos Orb differs from everything previously highlighted is the fact that on use it spawns an object into the game world. This means that the script for the active item needs to instantiate a prefab of the object each time it's used. The functionality of the item itself in this case is under the spawned object. Chaos Orb shoots out a beam every half beat, so twice per beat, for a couple of beats. Thankfully KILLBEAT has an excellent rhythm system and timing different events to beats is as easy as calling a couple of methods in the beat manager script to create a sequence that is synced with the beats per minute of the game and then adding rhythm actions such as methods, either in quarter, half or full beats, to said sequence. Each half beat the method for processing a single shot of the item is called, in which a random enemy target in range is chosen, a beam is shot out

towards it and a random status effect is applied to it, rinse and repeat until the maximum duration is reached and the object deletes itself from the game world.

The following picture shows a peek at the various status effects made as part of the work of this thesis, except for burn, that the item can trigger. It should be noted that one status effect was deemed too strong to be triggered by the item, as it increases the amount of damage enemies take. A status effect is randomized while calling for the pictured method that then adds it to the target's list of active status effects. As status effects do different things, some may need references for different values while others work the same way every time upon activation. One status effect, called Death Mark, needed its own manager script as the game needs to keep track of all enemies afflicted with the status effect.

```
private void ApplyStatusEffect(StatusEffectType _status, MBEntityBase _entity)
{
    switch (_status)
    {
        case StatusEffectType.BURN:
            StatusEffectBurn burn = new();
            if (_entity.StatusEffect.TryAdd(burn))
                burn.PassValues(PVConstants.STATUS_EFFECT_BURN_DURATION);
            break;

        case StatusEffectType.CHILL:
            _entity.StatusEffect.TryAdd(new StatusEffectChill());
            break;

        case StatusEffectType.FREEZE:
            _entity.StatusEffect.TryAdd(new StatusEffectFreeze());
            break;

        case StatusEffectType.ELECTRIFY:
            StatusEffectElectrify electrify = new();
            if (_entity.StatusEffect.TryAdd(electrify))
                electrify.PassValues(m_source, m_orbConfig.m_chainCount, m_orbConfig.m_targetCount,
                  m_orbConfig.m_electrifyDamage, m_orbConfig.m_chainRange);
            break;

        case StatusEffectType.DEATHMARK:
            DeathMarkManager.AddDeathMark(_entity);
            break;
    }
}
```

Picture 4: The method for applying a randomized status effect on a target in the script of the Chaos Orb active item.

The number of active items in the game was doubled with four new ones. Essentially, the amount of meta progression in the game has been doubled. Because of this, the variance of the game has significantly increased.

4 CONCLUSION

Variance brought by a plethora of gameplay choices with a mix of randomness is what keeps players engaged and playing roguelites for hours. The integral design elements of the genre which have evolved over 50 years of iteration are what keep players excited to replay what is essentially the same strip of gameplay over and over again. Roguelite design principles are an excellent way to not only enable small teams of developers to create exceptional games but also give a unique and fun twist to larger projects.

One of the main purposes of this thesis was to learn and analyse how progression systems in roguelites are designed and made in order to project that knowledge into the development of new content for the progression systems of KILLBEAT. Important lessons were learned with the help of both the people who pass their knowledge of game development along on the internet and the developers of KILLBEAT. Lessons such as how to set up a development pipeline for creating numerous assets for a character progression system with the help of a spreadsheet application like Google Sheets or how to design synergies to enable fun playstyles for players.

Ultimately the main goal was to increase the amount of variance of a run in KILLBEAT with new passive items, weapons and active items. With the work to be done including both the design and programming of the aforementioned content, the groundwork done in the theoretical part of this thesis was paramount to any kind of success.

The concerns of working on a game with a certain sense of unfamiliarity by the virtue of it being rhythm-based were thankfully overcome. By thinking of the beat as yet another game mechanic for which gameplay elements can be built around, the work became incredibly interesting and fulfilling. With the help of a personal interest in progression systems in games and the experience of countless of hours in roguelite games, the design part of the work came rather naturally. The implementation part of the process could have been harder than it was, were it not for the excellent guidance and previous work done by the KILLBEAT team.

To recap, a total of 32 passive items, nine unique weapons and four active items were designed and implemented into the game. Passive items were almost doubled, meaning the length of a single run can be increased while keeping an ideal amount of variance in what the player can find. While comparatively not as many weapons were added, each new one has a unique firing pattern, making them fitting additions as weapons that are on the rare side but bring a rewarding sense of discovery upon finding one. The number of active items was doubled, meaning the options at the start of a run were also doubled. As active items are used as the meta progression of the game, each new active item not only adds a unique way to play the game, but also something new to unlock in between runs.

As far as the client was concerned, the goals for the work done here were met. The amount of variance in a run was increased sufficiently and new synergies and playstyles were also successfully designed and created as something for the player to pursue. Ultimately one of the worst feelings a player can have while playing a roguelite is the desperation in finding and getting the same rewards over and over again, either because there aren't enough unique options or because the player feels like they are the only viable ones. The work done here was done mainly to combat likelihood of the player getting that feeling, because in the end, player satisfaction and engagement are paramount to a game's success on a fundamental level.

REFERENCES

Albert, M. What Are Roguelites? A Deep Dive. HackerNoon 10.4.2023. Read 10.1.2025. https://hackernoon.com/what-are-roguelites-a-deep-dive

Annand G. By Design: Procedural Generation. Superjump Magazine 15.5.2024. Read 15.1.2025. https://www.superjumpmagazine.com/by-design-procedural-generation/

Bellow P. Permadeath and Procedural Generation: The Heart of Roguelike Games. LitRPG Reads 6.8.2023. Read 16.1.2025. https://litrpgreads.com/blog/permadeath-and-procedural-generation-the-heart-of-roguelike-games

Bellow, P. Permadeath: The Heart of Roguelike Gameplay. LitRPG Reads 29.8.2024. Read 16.1.2025. https://litrpgreads.com/blog/permadeath-the-heart-of-roguelike-gameplay

Bycer, J. The Importance of Progression Gameplay Models. Game Wisdom 29.5.2014. Read 20.1.2025. https://game-wisdom.com/critical/progression-gameplay

Bycer, J. The Roguelike Debate – Roguelikes vs Roguelites. Game Developer 25.11.2019. Read 14.1.2025. https://www.gamedeveloper.com/design/the-roguelike-debate----roguelikes-vs-roguelites

Cartlidge, J. Genre, Prototype Theory and the Berlin Interpretation of Roguelikes. Game Studies 9.2024. Read 13.1.2025. https://gamestudies.org/2403/articles/cartlidge

Eng, D. What are Progression Systems in Games? University XP 16.1.2024. Read 7.1.2025. https://www.universityxp.com/blog/2024/1/16/what-are-progression-systems-in-games

Ritz, E. Thoughts on the Berlin Interpretation. One More Game-Dev and Programming Blog 19.3.2014. Read 13.1.2025. https://ericjmritz.wordpress.com/2014/03/19/thoughts-on-the-berlin-interpretation/

Switzer, E. Uh Oh, Everyone Is Mixing Up Roguelikes And Roguelites Again. The Gamer 26.4.2021. Read 8.1.2025 https://www.thegamer.com/roguelike-roguelite-difference-hades-darkest-dungeon/

Torick, S. Roguelites deconstructed. Linkedin 15.02.2024. Read 21.1.2025. https://www.linkedin.com/pulse/roguelites-deconstructed-svyatoslav-torick-oofgf