



**ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΕΛΟΠΟΝΝΗΣΟΥ**
University of the Peloponnese

M.Sc. in Computer Science

Postgraduate Thesis

**Design and Development of a First-Person Shooter Video Game with
AI-Based Enemy Generation Using Unity**

Angeliki Zympidi

Student ID: 2022202402004

Supervisor: Grigorios Dimitroulakos, Laboratory – Laboratory Teaching Staff

Supervisor: Costas Vasilakis – Professor

TRIPOLI

September 2026

Abstract

This thesis presents Sharp Shooter, a first-person shooter built in Unity. Its main contribution is a system that lets a player create a new enemy during play by typing a short description of it. The description is turned into an image with a text-to-image model, Stable Diffusion XL, and the image is turned into a three dimensional model with an image-to-3D model, TripoSR. Both models run on free Google Colab notebooks, split across two separate services and chained together, so that the game only ever has to talk to one of them. The resulting model is brought into Unity while the game is running, scaled and oriented correctly, given a collider, and connected to the same navigation, health, and combat components used by every other enemy in the game, so that a generated enemy behaves exactly like a hand designed one.

Beyond this generation system, the thesis also documents the rest of the game's engineering, including the player's movement and camera, the level, which began as a simple prototype and was later rebuilt as a forest, the enemy AI, the weapons, and the interface. Most of the thesis is devoted to this engineering work and to the design of the generation system, while a smaller part describes the game from the player's point of view, covering the story, the core combat, and how the enemy creation tool is actually used during a session.

Combining publicly available generative models with an ordinary game engine turns out to be achievable in practice. Creating an enemy is not instant, a request takes a few minutes on free hardware, with almost all of the time spent on reconstructing the mesh. Once created, though, generated enemies can be fought and defeated at the same pace as any other enemy in the game, and the pipeline keeps working, falling back to the game's hand made enemies, if the generation service itself is slow or unreachable.

Keywords

Unity, first-person shooter, procedural content generation, generative artificial intelligence, text-to-image synthesis, Stable Diffusion XL, image-to-3D reconstruction, TripoSR, glTF and GLB runtime loading, NavMesh, runtime asset generation, game AI.

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει το «Sharp Shooter», ένα παιχνίδι πυροβολισμών πρώτου προσώπου (first person shooter - FPS) που αναπτύχθηκε στη μηχανή Unity. Η κύρια συνεισφορά του είναι ένα σύστημα που επιτρέπει στον παίκτη να δημιουργεί έναν νέο εχθρό κατά τη διάρκεια του παιχνιδιού, πληκτρολογώντας μια σύντομη περιγραφή του. Η περιγραφή μετατρέπεται σε εικόνα μέσω του μοντέλου text-to-image «Stable Diffusion XL» και η εικόνα μετατρέπεται σε τρισδιάστατο μοντέλο μέσω του μοντέλου image-to-3D «TripoSR». Και τα δύο μοντέλα εκτελούνται σε σημειωματάρια (notebooks) του Google Colab (η χρήση των οποίων δεν επιφέρει χρέωση), κατανεμημένα σε δύο ξεχωριστές υπηρεσίες που συνδέονται διαδοχικά, ώστε το παιχνίδι να επικοινωνεί μόνο με μία από αυτές. Το τελικό μοντέλο εισάγεται στο Unity κατά την εκτέλεση του παιχνιδιού, προσαρμόζεται ως προς την κλίμακα και τον προσανατολισμό, αποκτά αντικείμενο ανίχνευσης συγκρούσεων (collider) και συνδέεται με τα ίδια στοιχεία πλοήγησης, υγείας και μάχης που χρησιμοποιούνται από όλους τους άλλους εχθρούς, διασφαλίζοντας ότι ο παραγόμενος εχθρός συμπεριφέρεται ακριβώς όπως ένας εχθρός που έχει σχεδιαστεί χειροκίνητα.

Πέρα από αυτό το σύστημα παραγωγής, η εργασία περιγράφει επίσης την υπόλοιπη τεχνική υλοποίηση του παιχνιδιού, συμπεριλαμβανομένων της κίνησης και της κάμερας του παίκτη, του επιπέδου (το οποίο ξεκίνησε ως απλό πρωτότυπο και αργότερα ανακατασκευάστηκε ως δάσος), της τεχνητής νοημοσύνης των εχθρών, των όπλων και της διεπαφής χρήστη. Το μεγαλύτερο μέρος της εργασίας εστιάζει σε αυτή την τεχνική υλοποίηση και στον σχεδιασμό του συστήματος παραγωγής, ενώ ένα μικρότερο τμήμα περιγράφει το παιχνίδι από την οπτική γωνία του παίκτη, καλύπτοντας την ιστορία, τον βασικό μηχανισμό μάχης και τον τρόπο χρήσης του εργαλείου δημιουργίας εχθρών κατά τη διάρκεια του παιχνιδιού.

Αποδεικνύεται ότι ο συνδυασμός δημόσια διαθέσιμων παραγωγικών μοντέλων με μια συμβατική μηχανή παιχνιδιών είναι εφικτός στην πράξη. Η δημιουργία ενός εχθρού δεν είναι άμεση διαδικασία, καθώς ένα αίτημα απαιτεί μερικά λεπτά σε δωρεάν διατιθέμενο υλικό (hardware), με τον περισσότερο χρόνο να αναλώνεται στην ανακατασκευή του πλέγματος (mesh). Ωστόσο, μόλις δημιουργηθούν, οι παραγόμενοι εχθροί μπορούν να αντιμετωπιστούν και να ηττηθούν με τον ίδιο τρόπο όπως κάθε άλλος εχθρός του παιχνιδιού, ενώ η διαδικασία

παραμένει λειτουργική, επιστρέφοντας στη χρήση των χειροκίνητα κατασκευασμένων εχθρών του παιχνιδιού εάν η υπηρεσία παραγωγής είναι αργή ή μη προσβάσιμη.

Λέξεις-κλειδιά

Unity, παιχνίδι πυροβολισμών πρώτου προσώπου, διαδικαστική παραγωγή περιεχομένου, παραγωγική τεχνητή νοημοσύνη, σύνθεση κειμένου-σε-εικόνα, Stable Diffusion XL, ανακατασκευή εικόνας σε 3D, TripoSR, φόρτωση κατά τον χρόνο εκτέλεσης glTF και GLB, NavMesh, δημιουργία πόρων κατά τον χρόνο εκτέλεσης, AI σε παιχνίδι.

Table of Contents

Abstract.....	i
Keywords	i
Περίληψη.....	ii
Λέξεις-κλειδιά	iii
Table of Contents.....	iv
List of Figures	viii
List of Abbreviations	ix
Chapter 1 — Introduction	1
1.1 Motivation.....	1
1.2 Problem Statement	1
1.3 Objectives of the Thesis	2
1.4 Contributions	3
1.5 Thesis Structure.....	4
1.6 Scope Statement	4
Chapter 2 — Background and Related Work.....	6
2.1 Game Engines and the Unity Ecosystem	6
2.2 Artificial Intelligence in Games	6
2.3 Procedural Content Generation	7
2.4 Diffusion Models for Text-to-Image Synthesis	8
2.5 Single-Image 3D Reconstruction	8
2.6 Positioning of Sharp Shooter Relative to Existing Work	9
2.7 Software Engineering Background	9
2.8 Related Systems and Products	9
Chapter 3 — System Architecture Overview.....	11
3.1 High-Level System Decomposition	11
3.2 Local Runtime Architecture.....	12
3.3 Data-Driven Configuration and Runtime Persistence	12
3.4 Network Architecture of the Generative Pipeline	13
3.5 Failure Modes and Degradation Strategy.....	14
3.6 Scene Versioning and Project Structure	15
Chapter 4 — First-Person Character and Camera Systems	16
4.1 The First Person Starter Asset as a Foundation	16

4.2 Character Controller and Movement	16
4.3 Cinemachine Virtual Camera	17
4.4 Input System Integration.....	18
4.5 Aim and Field-of-View Switching.....	19
4.6 Rendering Configuration for the Player Camera.....	19
4.7 Extending the Starter Asset.....	20
4.8 Tuning Process	20
Chapter 5 — Environment and Level Design	22
5.1 Two-Stage Level Development Strategy.....	22
5.2 Prototyping with ProBuilder.....	22
5.3 Transition to the Forest Environment	23
5.4 Post-Processing and Colour Grading	24
5.5 Particle Effects.....	25
5.6 Performance Considerations in the Forest Environment	26
5.7 Art Direction Consistency with Generated Content	26
Chapter 6 — Enemy AI and Navigation	28
6.1 Two Kinds of Enemy	28
6.2 Navigation Surface and Baking	29
6.3 Chase Behaviour.....	30
6.4 Facing and Orientation.....	32
6.5 Enemy Health	32
6.6 Runtime Difficulty Scaling.....	33
Chapter 7 — Combat, Weapons and Interaction Systems	35
7.1 Hit-Scan Combat.....	35
7.2 Layers and Hit Filtering.....	35
7.3 Gun Setup and Presentation	36
7.4 Damage and Impact Effects.....	36
7.5 Pickups and Interaction	37
7.6 Numeric Display and Small Code Conventions	38
7.7 Why Hit-Scan Rather Than Projectiles.....	39
7.8 Balancing Weapon Damage Against Enemy Health.....	39
Chapter 8 — UI, Data Architecture and Scriptable Objects.....	41
8.1 Canvas and Event System.....	41

8.2 Grid Based Layout	41
8.3 Constrained Values in the Inspector.....	42
8.4 Scriptable Object Data.....	42
8.5 Prefab Variants.....	43
8.6 Interface State During Long Requests	44
8.7 Updating the Heads Up Display Efficiently	44
Chapter 9 — Runtime Generative Enemy Pipeline.....	46
9.1 Design Goals and Constraints	46
9.2 Stage One, Turning Text into an Image	47
9.3 Stage Two, Turning an Image into a 3D Model.....	48
9.4 Serving the Models Over the Web	49
9.5 How Unity Drives a Request	50
9.6 Bringing the Model into Unity	52
9.7 Assembling a Playable Enemy	53
The Enemy Gate	54
9.8 Saving and Browsing Generated Characters.....	55
9.9 Handling Failures.....	55
9.10 How the Code Is Organised	56
9.11 Why the Prompt Is Left Untouched.....	57
9.12 What Data the Pipeline Handles.....	58
9.13 Comparing This to Preparing Enemies in Advance	58
Chapter 10 — How Sharp Shooter Is Played	59
10.1 Story and Difficulty Selection	59
10.2 Starting a Session and the Heads Up Display.....	60
10.3 Movement and the Forest.....	61
10.4 The Core Fight	61
10.5 Designing a Custom Enemy	62
10.6 Fighting a Player Made Enemy	64
10.7 Defeat.....	65
10.8 Winning the Encounter	66
10.9 Player Choice and Replayability	67
10.10 Controls.....	67
10.11 Pacing and Difficulty Over a Session.....	68

10.12 Feedback During Combat	69
10.13 A Second Example	69
10.14 What the Player Never Sees	70
10.15 Summary of the Play Experience	70
Chapter 11 — Conclusion.....	72
References	74
Appendix A — Selected Code Listings	77
A.1 Enemy Health Component (Section 6.5)	77
A.2 Enemy Chase Behaviour (Section 6.3).....	78
A.3 Difficulty Multipliers (Section 6.6).....	80
A.4 Spawn Gate (Sections 6.1, 6.6).....	81
A.5 Hit-Scan Weapon Fire (Sections 7.1, 7.4)	81
A.6 Abstract Pickup Base Class (Section 7.5)	82
A.7 Generative Request Flow, Simplified (Section 9.5)	83
A.8 Runtime Enemy Assembly (Section 9.7).....	84
A.9 Runtime Enemy Gate (Section 9.7)	86
A.10 The Colab Generation Pipeline (Sections 9.2–9.4)	87

List of Figures

Figure 1 High-level architecture diagram of Sharp Shooter	11
Figure 2 Sequence diagram of a single generation request	14
Figure 3 Cinemachine Virtual Camera inspector panel showing the lens settings	17
Figure 4 Input Action Asset editor window showing the Player action map	19
Figure 5 Side-by-side comparison of the early prototype level and the final forest environment	22
Figure 6 ProBuilder editing session inside the Unity Editor.....	23
Figure 7 The final forest environment as seen from the player's first-person perspective	24
Figure 8 Global Volume inspector	25
Figure 9 Diagram of the two enemy kinds	29
Figure 10 A baked NavMesh area	30
Figure 11 Scene view during Play Mode showing several agents' current paths converging on the player	31
Figure 12 A burst of destruction particles as an enemy is defeated mid-fight, with a second enemy still approaching in the distance.....	37
Figure 13 The grid based gallery of saved characters inside the Enemy Creator interface	42
Figure 14 The three WeaponSO data assets in the Unity Project window, Machine Gun, Pistol, and Sniper Rifle	43
Figure 15 The in-game Enemy Creator prompt field showing the example prompt used to generate the white companion robot enemy	48
Figure 16 The Enemy Creator panel during an active generation request	51
Figure 17 The generated companion robot enemy spawned into the forest level, engaged by the player's weapon	54
Figure 18 The opening story sequence establishing Aurora Station and Prometheus, followed by the difficulty selection screen with its three options	60
Figure 19 Full heads up display during active gameplay, the enemy counter top left, health bar bottom left, ammunition counter bottom right, and the enemy creator prompt top right.....	61
Figure 20 The player engaging an enemy at close range in the forest, crosshair centred on the target, ammunition counter visible in the lower right corner	62
Figure 21 The Enemy Creator panel open mid-session, the scrollable gallery of saved characters with live rotating previews on the left, and the empty prompt field with the Create Enemy and Add Enemy Gate buttons on the right, ready for a new description.....	63
Figure 22 The completed generated enemy, the white companion robot, among the trees shortly after being spawned	64
Figure 23 A second generated enemy destroyed mid-fight, with another enemy visible nearby at the same time, and the weapon model in the foreground	65
Figure 24 The Game Over screen, showing the Restart and Quit buttons over the forest backdrop	66
Figure 25 The victory screen, shown once the enemy counter reaches zero, confirming that Aurora holds and offering Play Again and Quit	67

List of Abbreviations

ABI: Application Binary Interface

AI: Artificial Intelligence

API: Application Programming Interface

FPS: First-Person Shooter

FXAA: Fast Approximate Anti-Aliasing

GLB: Binary glTF (GL Transmission Format)

GPU: Graphics Processing Unit

HUD: Heads-Up Display

HTTP: Hypertext Transfer Protocol

LDM: Latent Diffusion Model

LRM: Large Reconstruction Model

NPC: Non-Player Character

PBR: Physically Based Rendering

PCG: Procedural Content Generation

PCGML: Procedural Content Generation via Machine Learning

REST: Representational State Transfer

SD: Stable Diffusion

SDXL: Stable Diffusion XL

URL: Uniform Resource Locator

URP: Universal Render Pipeline

UV: Texture-mapping coordinate space (U, V)

VFX: Visual Effects

Chapter 1 — Introduction

1.1 Motivation

Over the past decade, video games have increasingly used artificial intelligence, not only to control the behaviour of non-player characters, but also to help create the content of the game itself. Procedural content generation, the algorithmic creation of levels, textures, or enemies, has long been used to lower production cost and increase replayability. More recently, generative models for images and for three dimensional shapes have opened a new possibility, as they produce content that is not just arranged by rules, but actually created from a plain description written by a person, sometimes even by the player, at the moment it is needed.

This thesis grew out of a simple question, namely whether text-to-image generation and single-image 3D reconstruction, two technologies that already work well on their own, could be combined into a pipeline that runs inside a live game rather than as a separate tool used before the game is finished. If a player can describe an enemy in a sentence and see that description become a textured, animated, fightable character a short time later, the line between making a game and playing it starts to blur. This idea is explored through Sharp Shooter, a first-person shooter built in Unity.

The motivation behind the project is therefore twofold. On the engineering side, the goal was to see whether generative AI services could be connected to a real time game engine reliably and at reasonable quality. On the design side, the goal was to explore what becomes possible for the player once enemies are no longer a fixed set decided before release, and can instead be created, through a written description, during play itself.

1.2 Problem Statement

A typical Unity first-person shooter builds its enemies the same way every time. An artist models and textures a character, a designer rigs and animates it, a programmer attaches its AI and combat behaviour, and the result is packaged as a prefab that gets placed into the level. This process produces reliable, good looking enemies, but it is closed, that is the full set of enemies a player will ever meet already exists before the game is built. Even projects that use procedural

generation for enemy statistics or spawn placement are usually still drawing from a fixed pool of models made in advance.

This thesis looks at how that process could be extended with a new way of creating enemies, one where the actual three dimensional model, not just its stats, is produced on demand from a short text description. Whatever comes out of that process still has to work exactly like a normal enemy, needing a collider, the right scale and orientation, the ability to navigate the level, and the ability to take damage and die, all without breaking the game if the generation service is slow or unavailable.

Put more concretely, the thesis had to solve the following problems. First, how to turn a short prompt into an image that is actually usable for 3D reconstruction. Second, how to turn that image into a solid, correctly scaled mesh. Third, how to move that mesh from a remote service into Unity while the game is running. Fourth, how to clean up the result so it behaves like a normal enemy, and fifth, how to offer all of this to the player through an interface that is simple to use and does not get in the way of play.

1.3 Objectives of the Thesis

The overall goal of this thesis is to design, build, and evaluate a working Unity first-person shooter, Sharp Shooter, whose defining feature is the ability to create enemies at runtime from a text description. This breaks down into a few concrete objectives.

- Build a complete, playable first-person shooter in Unity, covering movement, camera control, weapons, combat, enemy AI, level design, lighting, and interface, built on top of Unity's First Person Starter Asset and extended well beyond it.
- Design and build two versions of the level. An early prototype used to test the core mechanics quickly, and a later, visually finished forest environment used in the final build.
- Implement two kinds of enemy, hand made scripted enemies and runtime generated ones, so that both work through the same combat and navigation systems, and add a

difficulty setting that scales enemy health, speed, and spawn rate the same way for both kinds.

- Design and implement a pipeline that turns text into a 3D model, using a text-to-image stage and an image-to-3D stage, hosted as small web services and reachable from the game through a single request.
- Bring the generated models into Unity at runtime, cleaning them up automatically so they are functionally the same as a hand made enemy from the player's point of view.
- Build an in-game interface where the player can type a description, browse previously created characters, and add a newly generated enemy to the level without breaking the flow of play.
- Confirm that the finished system works correctly, and observe its generation speed and robustness during development and play.

1.4 Contributions

The main contributions of this thesis are the following.

- A working, end-to-end pipeline that connects a text-to-image model and an image-to-3D model to a live Unity session, with no offline preparation step involved.
- A practical way of exposing models running on free notebook services as reliable web endpoints reachable from a game, including how to handle timeouts and failures from inside Unity's request system, and how to chain two independently hosted models so that the game only ever sees one request.
- A processing and assembly step that takes whatever mesh the pipeline produces and turns it into a usable enemy, by scaling and orienting it correctly, building a collider from its measured size, checking that every material actually renders correctly, and attaching the same navigation, health, and combat components a hand made enemy uses.

- A two part enemy system, hand made and generated, that shows how generated content can live alongside traditional content in the same combat and AI systems, sharing not just components but also difficulty scaling and win conditions.
- A complete, documented Unity project, Sharp Shooter, that works as a concrete example of using generative AI during play rather than only before it.

1.5 Thesis Structure

The rest of the thesis is organised as follows. Chapter 2 reviews related work in game AI, procedural content generation, and generative AI for images and 3D shapes, and places this project within that context. Chapter 3 gives an overview of the system as a whole. Chapters 4 through 8 describe the traditionally built parts of the game in detail, the character and camera, the level and its environment, the enemy AI, the combat systems, and the interface. Chapter 9 is the technical core of the thesis and covers the generative pipeline in full. Chapter 10 looks at the finished game from the player's side, describing the story, the combat, and how the enemy creation tool is used in practice. Chapter 11 concludes the thesis.

1.6 Scope Statement

It is worth being clear about what this thesis is not trying to be. It is not a commercial, shippable game. The story, level count, and audio were kept deliberately small so that effort could go into the systems this thesis is actually about. It is not a new machine learning model or training method, for this reason Stable Diffusion XL and TripoSR are used exactly as published, downloaded ready to use, with no fine tuning done as part of this work. It is not really a contribution to how such models should be deployed either, but the hosting setup used here is shaped by the limits of free infrastructure, and a properly funded deployment would likely look different. And it does not claim that generated content is better than hand made content in general, as the evaluation is honest about the gap that remains between the two. What the thesis does claim is narrower, that this particular integration is achievable, that it has been documented in detail, and that it works inside a real, playable game.

Finally, this is not a study comparing the quality of generated content against a baseline. The evaluation shows that the pipeline works, how long it takes, and how it fails, but it does not compare the artistic or gameplay value of generated enemies against hand made ones through a separate player study.

Chapter 2 — Background and Related Work

2.1 Game Engines and the Unity Ecosystem

A modern game engine such as Unity provides a shared runtime for rendering, physics, animation, audio, input, and scripting. It is organised around a component based architecture, meaning that behaviour is attached to objects as separate, modular pieces rather than built through deep inheritance hierarchies [1]. This lets a single object, an enemy for example, be assembled out of independently developed pieces, such as a mesh renderer, a collider, a navigation agent, a health script, and a combat script. This flexibility is part of what makes it possible to attach a mesh that did not exist when the project was built to an otherwise ordinary, already written enemy.

Sharp Shooter is built on top of Unity's official First Person Starter Asset, a package that provides basic movement, a first-person camera, and input handling for keyboard, mouse, and gamepad. Using an existing starter package instead of writing a controller from scratch meant that development time could go into the parts of the project that are new, mainly the AI, the environment, and the generative pipeline, while still requiring a fair amount of work to extend the starter asset with weapons, health, pickups, and an interface.

2.2 Artificial Intelligence in Games

Game AI is not really about optimal play. Its goal is to be believable and enjoyable to fight, since an enemy that is too effective, too predictable, or too erratic can hurt the player's experience even if it is technically well built [2]. Millington and Funge [2] describe a layered model of game AI, movement, decision making, and strategy, built on top of pathfinding structures such as navigation meshes. Unity's own NavMesh system follows this same layered approach, that is a walkable surface is baked over the walkable geometry of the level, and agents attached to enemies use it to compute a path toward a target, usually the player, by repeatedly asking for a new destination [2], [3].

Aversa [4] covers this same model specifically within Unity, including pathfinding and simple sensor based behaviour, and provided much of the background for how enemies chase the player

in this project. Isla's account of the AI used in Halo 2 [5] is also useful background, in particular its argument that a small number of well tuned, readable behaviours usually make a more satisfying opponent than a larger but shallower set of behaviours. Sharp Shooter follows this idea closely, as its enemies really do two basic things, chasing the player across the level and exploding on contact, and all of the variety in a fight comes from how many enemies are present and from a difficulty setting rather than from any single enemy being more complex.

2.3 Procedural Content Generation

Procedural content generation is the algorithmic creation of game content, such as levels, textures, or characters, using rules or search rather than manual work [6]. Shaker, Togelius, and Nelson [6] describe this field along a few axes. Some of the main axes concern what kind of content is produced, whether it happens offline or during play, and whether the process is deterministic, random, or search based. The difficulty setting in Sharp Shooter fits the simple end of this picture, as it runs during play, since enemy health, speed, and spawn rate are computed the moment an enemy appears rather than fixed in advance, but it is deterministic rather than random, since every enemy at a given difficulty gets the same multipliers. It is included here mainly as a comparison point for the generative side of the project. In the sense described by Watkins for Unity specifically [7], it varies an enemy's numbers within a fixed range, while the generative system varies the model itself.

The generative enemies do not really fit the classic picture of procedural content generation, because what they produce is not assembled from a designer's set of parts and parameters at all, instead it is generated pixel by pixel and then point by point by a learned model responding to free text. This sits closer to the newer field of procedural content generation through machine learning, surveyed by Summerville and colleagues [8], which looks at how trained models, rather than hand written rules, can produce game content. Most of the work surveyed there targets flat content such as levels or textures generated by models trained specifically on game data. Sharp Shooter instead reuses general purpose models that were never trained on games at all, and brings them into a live 3D session, which is a less explored part of this space.

2.4 Diffusion Models for Text-to-Image Synthesis

The text-to-image half of the pipeline is built on Stable Diffusion, a family of models for generating images from text. Diffusion models, described by Ho, Jain, and Abbeel as denoising models [9], generate an image by learning to reverse a gradual process of adding noise. Starting from random noise, the network is trained to repeatedly predict and remove a small amount of it until a coherent image appears. Rombach and colleagues [10] showed that this process works far better when it happens inside a compressed representation produced by a separate, pretrained network, rather than directly on full resolution pixels, and this idea underlies every version of Stable Diffusion, including the one used here.

The specific model used in this thesis is Stable Diffusion XL [11], the larger of two reasonable choices, run in its normal configuration of thirty denoising steps at a resolution of 1024 by 1024. A distilled model such as SD Turbo [12] was also considered, since such models trade some quality for a large drop in generation time, needing only one or two steps instead of the twenty to fifty a normal model needs. That trade only matters when the image stage is the slow part of the pipeline, and Chapter 9 explains why that stopped being true for this project.

2.5 Single-Image 3D Reconstruction

The image-to-3D half of the pipeline is built on TripoSR [13], a model that converts a single image into a textured 3D mesh in a single pass, without the lengthy per-object optimisation that older methods needed. It belongs to a family of models often called Large Reconstruction Models, which predict a 3D representation of an object directly from image features in one step. Earlier approaches, such as DreamFusion [14], instead optimise a scene over many minutes by repeatedly scoring it against a 2D diffusion model, and were ruled out for this project simply because that kind of delay does not fit inside a live game session.

What makes TripoSR usable here is the shape of its cost rather than its raw speed, since a single forward pass has a cost that is bounded and predictable, while an optimisation based method can take an unpredictable amount of time to converge. On the graphics hardware TripoSR is normally benchmarked on it runs in under a second, but this project does not have access to that hardware for this stage, and instead runs it on a slower, ordinary processor, where

a request takes a little over a minute. That is still fast enough for a player to trigger a generation and fight the result in the same session, which is the property that actually matters.

2.6 Positioning of Sharp Shooter Relative to Existing Work

Game AI and navigation, procedural content generation, and generative models for images and 3D shapes are each well studied areas on their own, but there is not much written about combining them inside a shipped, playable game rather than a research demo or an offline tool. Sharp Shooter does not contribute a new model or a new AI algorithm. Its contribution is a working integration, a documented pipeline that connects an ordinary text-to-image model and an ordinary image-to-3D model to Unity's existing navigation, combat, and animation systems, and exposes the result to the player through a simple in-game tool. The rest of this thesis describes that integration in detail, starting with an overview of the system in the next chapter.

2.7 Software Engineering Background

Beyond the game specific literature above, the implementation also draws on general software design practice. The catalogue of object oriented design patterns by Gamma and colleagues [15] is a useful reference point, since patterns such as Component, Template Method, and Singleton show up throughout the project's code, though usually in a fairly restrained form rather than textbook fashion. Nystrom's Game Programming Patterns [16] provided a second, more game specific layer of guidance, particularly around how a game's main update loop is structured. These influences are discussed more concretely where they apply, in the chapters on the enemy and combat systems.

2.8 Related Systems and Products

No shipped game combining live text-to-image and image-to-3D generation for enemy creation was found while preparing this thesis, but two related kinds of tool are worth mentioning. Offline AI assisted asset tools, which generate textures, concept art, or meshes for an artist to bring into a game later, are increasingly common in production, but they work before a build ships rather than during play. Sandbox games with in-game construction tools, where players build structures or creatures from a fixed set of pieces, offer a form of runtime content

creation that predates generative AI, but that content is assembled from a limited, pre-made set of parts rather than generated by a model.

Sharp Shooter's Enemy Creator sits between these two ideas without fully belonging to either. Like the offline tools, it uses a genuinely generative model rather than assembling fixed pieces. Like the sandbox tools, it runs entirely during play, inside an active session, rather than as a separate step beforehand. That combination, generative and available during play at the same time, is the less explored part of this space that the rest of the thesis focuses on.

Chapter 3 — System Architecture Overview

3.1 High-Level System Decomposition

Sharp Shooter is organised as an ordinary Unity project with one unusual subsystem added on top. The ordinary part follows the component based architecture typical of Unity development [1]. Objects in the scene carry components responsible for movement, camera control, weapons, health, AI, and interface, and these communicate mostly through direct references and a small number of manager objects. The unusual part is the runtime generative pipeline, which extends this architecture outward, across a network connection, to machine learning models hosted elsewhere.

At a high level the system can be thought of as five cooperating parts, a player and camera layer handling movement and weapons, an environment layer handling the level and lighting, an enemy and AI layer handling navigation, health, and combat, an interface layer handling menus and the heads up display, and a generative pipeline layer handling the text-to-image and image-to-3D services and how their output becomes a playable enemy. The first four are covered in Chapters 4 through 8, and the fifth is the subject of Chapter 9.

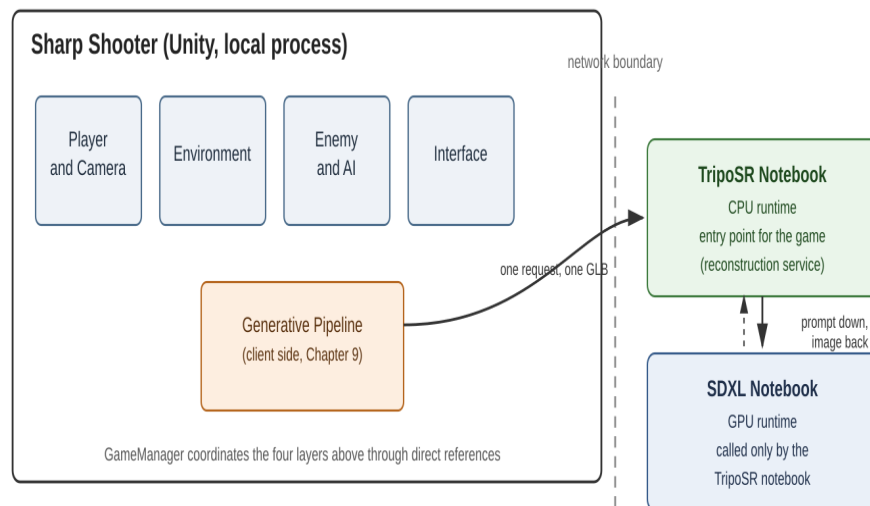


Figure 1 High-level architecture diagram of Sharp Shooter

3.2 Local Runtime Architecture

Within Unity itself the architecture is fairly ordinary. Gameplay logic is spread across small, single purpose scripts rather than centralised in one large manager, following the general component pattern described by Gamma and colleagues [15] and its game specific treatment by Nystrom [16]. A single persistent manager, a GameManager object that tracks the number of enemies left and detects when the level is won, is the only object referenced from many different places, and everything else is scoped to the object it is attached to. A couple of small pieces of shared state live outside the scene entirely, most notably a static object holding the player's chosen difficulty, kept static specifically so the choice survives a level restart.

Enemies of both kinds, hand made and generated, share the same basic shape at runtime, a root object carrying a navigation agent, the health component, the chase behaviour, and two capsule colliders, one solid for the player's weapon to hit and one slightly larger trigger for detecting contact with the player, with the visual model parented underneath. The colliders sit on the root rather than on a child object, so that a shot landing anywhere on the visible model always resolves to the same health component with a single lookup, no matter how the underlying geometry is structured.

This matters more for generated enemies than for hand made ones. A hand made enemy has a hierarchy the developer chose, while a generated model can arrive with its geometry split across any number of parts at any depth. Keeping every gameplay relevant component on one root, and sizing the colliders from the model's measured bounds rather than assuming anything about its structure, is what lets the combat and navigation code stay completely unaware of whether it is dealing with a hand made enemy or a generated one.

3.3 Data-Driven Configuration and Runtime Persistence

Weapon settings in Sharp Shooter are stored as separate data assets rather than hard coded into scripts, following common Unity practice [17] and the approach described by Watkins for procedural content in Unity [7]. Each weapon asset holds its damage, fire rate, magazine size, and references to its model and effects, and three such assets exist for the pistol, machine gun, and sniper rifle. Because the same asset is referenced everywhere it is needed, retuning a

weapon in the editor updates it consistently across the starting loadout, its pickups, and the ammunition display.

Enemy settings deliberately do not use this same mechanism. Base enemy health is a plain field on the health component, multiplied at spawn time by a difficulty factor. Enemy statistics only vary along one axis in this project, the difficulty the player picked, so a shared data asset would not have added anything the difficulty setting does not already provide directly.

Generated enemies are saved to disk rather than as project assets. Each successfully generated model is written as a file under the project's streaming assets folder, named from part of the prompt and a timestamp, and the in-game gallery of saved characters is built by simply reading that folder each time the creator panel is opened.

3.4 Network Architecture of the Generative Pipeline

The generative pipeline is the one part of Sharp Shooter that reaches outside the Unity process. It runs on two separate Google Colab notebooks rather than one, each running its own small web server [18] made reachable from the internet through its own tunnel [19]. The first notebook hosts Stable Diffusion XL and offers a single route that turns a prompt into an image. The second hosts TripoSR, offers the route the game actually calls, and also acts as the entry point to the whole pipeline. When it receives a prompt it asks the first notebook for an image, reconstructs a mesh from that image, and returns the mesh in the same response. From Unity's side this is invisible, one request goes out and one model comes back.

This split exists because of a limit of the free hosting tier rather than by choice. Only one accelerated session is available at a time, so the two models cannot both run on fast hardware at once. The project gives the accelerator to the image model, since a diffusion model without one is not just slow but impractical, taking several minutes per image, while the reconstruction model without one is merely slower, at a cost that is at least predictable and paid once per request.

The cost of this split is a short extra hop between the two notebooks, carrying the intermediate image as a small file that never reaches the player's own connection. In exchange,

the image stage becomes something that can be tested on its own. A generation that produces a poor result can be traced back to a specific stage simply by calling the first notebook directly with the same prompt.

From Unity's side the whole arrangement looks like a single, simple web request, a prompt goes out and the bytes of a 3D model come back. The request is handled by a background task rather than blocking the game, so the rest of the interface, including a rotating preview of previously created characters, keeps working while a generation is in progress.

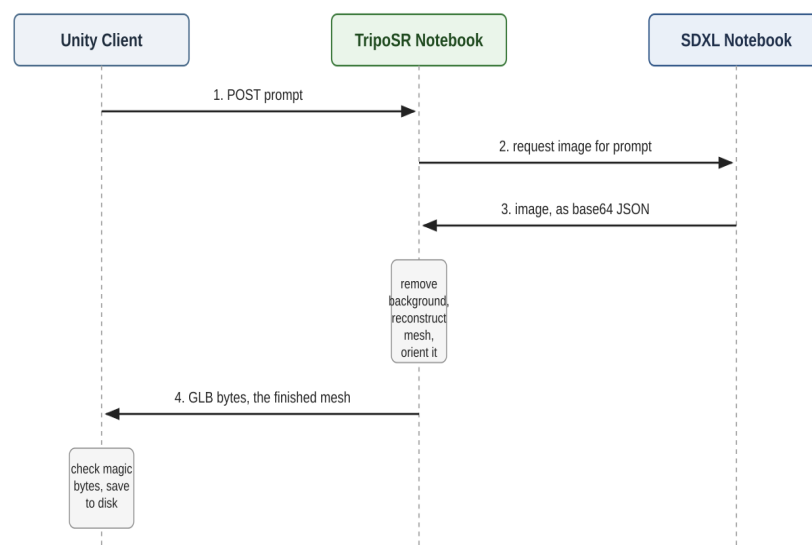


Figure 2 Sequence diagram of a single generation request

Running both models locally inside the Unity build itself was considered and ruled out early on. Fitting two large models, plus the game's own rendering and physics, on typical development or player hardware was judged impractical, so hosting the models separately kept the game itself lightweight and let each model be updated independently.

3.5 Failure Modes and Degradation Strategy

Because the generative pipeline depends on two notebooks and two tunnels that can each fail or time out, the whole pipeline is treated as optional rather than as something the game depends on. Opening the enemy creator pauses the game, so the player is never at risk while typing a description, and a generation request runs against that paused world with a status line

reporting its progress. If a request fails, the same line reports it, the buttons return to a usable state, and closing the panel simply returns the player to the game exactly where they left it, with nothing about the rest of the game affected.

- Hand made enemies are always available and need no network connection at all, so the game is fully playable and winnable without the generative pipeline ever being reachable.
- Newly generated enemies need both notebooks and both tunnels to be active, and a failure is reported to the player without disturbing anything else.
- Previously generated enemies are saved as files on disk, so a character made earlier in a session can still be browsed and added to the level with no network access at all.

3.6 Scene Versioning and Project Structure

Instead of using a version control system, the project kept a history of the level's development by saving each major stage as its own scene rather than overwriting the previous one. The project contains several level scenes in total, an early prototype, a couple of intermediate stages, and the final forest environment used in the shipped build, plus one unused template scene that Unity creates automatically and that was never touched. Because the geometry of each stage is saved directly inside its own scene file, the early prototype can still be opened and played today, which is what made the side by side comparison used later in this thesis possible.

Chapter 4 — First-Person Character and Camera Systems

4.1 The First Person Starter Asset as a Foundation

Sharp Shooter's player character is built on Unity's official First Person Starter Asset, a ready made package meant to save developers from writing basic first-person movement from scratch [20]. The package handles character movement, mouse driven camera control, and input wiring, but it is explicitly meant as a starting point rather than a finished product, so it does not include a weapon system, combat, or any advanced gameplay logic, all of which had to be built separately for this project.

Using a starter asset instead of writing a controller from scratch was a deliberate choice. Since the real contribution of this thesis is the generative enemy pipeline, it made sense to spend development time there and on the enemy AI and level design that support it, while relying on a well tested, proven implementation for basic movement and camera feel, extending it only where the project actually needed more.

4.2 Character Controller and Movement

Player movement uses Unity's built in Character Controller component rather than a full physics simulation. It handles movement and collisions directly, and movement is applied every frame by feeding it a direction and speed. A few of its parameters were tuned specifically for this project's levels, its height and radius, which control how its capsule shaped collider interacts with walls and obstacles, and its centre, which keeps that collider aligned with the visible character model.

Two further parameters matter for uneven ground, which became relevant once the level became a forest with real terrain. A step offset controls how tall a ledge the character can walk up without it counting as a wall, and a slope limit caps how steep the ground can be before the character can no longer climb it. Both were tuned together to keep movement predictable across the flat early prototype and the more uneven forest terrain that replaced it.

- Move, applies movement each frame using the Character Controller's own collision handling.

- Height, radius, and centre, define the capsule used for movement and obstacle collision.
- Step offset, the tallest step the character can walk over without jumping.
- Slope limit, the steepest ground the character can walk up.

4.3 Cinemachine Virtual Camera

Camera control is handled by Cinemachine, Unity's package for flexible, designer friendly cameras [21]. A single virtual camera drives the first-person view, following a dedicated target object rather than being parented directly to the character model, which keeps camera rotation separate from any animation on the character's body.

The camera rotates in response to mouse or gamepad input, with vertical rotation clamped so the player cannot look further up or down than feels comfortable, and horizontal rotation kept in sync with the direction the character itself is facing, so that where the player looks and where their weapon points always agree.

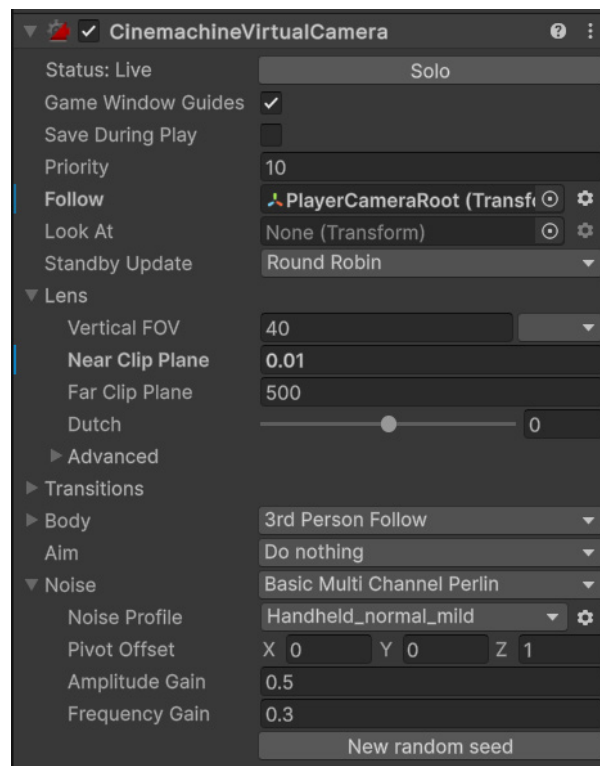


Figure 3 Cinemachine Virtual Camera inspector panel showing the lens settings

The camera's field of view controls how much of the scene is visible. A narrower field of view gives a more focused, zoomed in view suited to aiming, while a wider one shows more of the surroundings. Sharp Shooter uses a moderate field of view for normal movement and a narrower one while aiming. A small amount of procedural noise is also added to the camera to simulate a subtle handheld feel.

A camera shake effect is triggered by the weapon rather than by enemies, firing once per shot at the same moment as the muzzle flash, so the shake reads as recoil rather than as a reaction to a kill. This was a deliberate choice, as tying the shake to firing means the player feels every shot, including misses, while tying it to kills would leave rapid automatic fire feeling inconsistent from one hit to the next. Enemy destruction is instead signalled by its own explosion effect and by the enemy counter on the heads up display.

Where more than one virtual camera exists in a scene, for example during a scripted moment, Cinemachine picks whichever camera currently has the highest priority and blends smoothly to it, without the game needing to manually switch cameras on and off.

4.4 Input System Integration

All player input, movement, looking, jumping, sprinting, firing, and aiming, goes through Unity's newer Input System package rather than its older input manager, using an input asset that ships with the starter package and was extended with the weapon and aim actions this project needed [20]. Actions are grouped into maps, and each one can read from keyboard and mouse or a gamepad without needing separate code for each device.

How a button like action triggers is configured per action. Firing the weapon is set to register on the initial press, so a single click fires a single shot, while sprinting and aiming instead read a held value, so their effect lasts for as long as the button stays down.

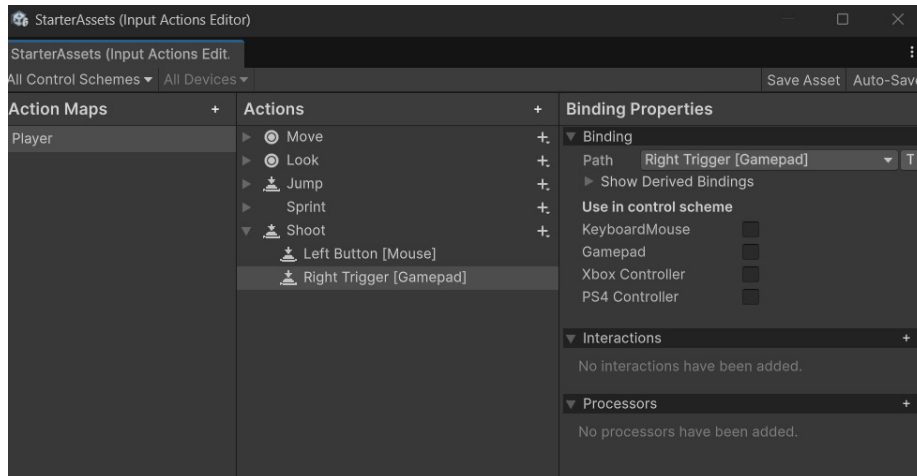


Figure 4 Input Action Asset editor window showing the Player action map

4.5 Aim and Field-of-View Switching

Aiming narrows the camera's field of view to give a focused, zoomed in view rather than moving the weapon into a fixed sight position, and this behaviour belongs to the weapon rather than to the player. Each weapon carries its own flag for whether it can zoom, along with its own zoom amount and turn speed while zoomed, and aiming has no effect at all on a weapon that cannot zoom. In the finished build only the sniper rifle zooms, the pistol and machine gun simply ignore the aim input.

For a weapon that does zoom, holding the aim button switches the camera's field of view and the player's turn speed to that weapon's values immediately, along with a vignette effect around the edge of the screen, and releasing it restores the normal values just as immediately. This change happens instantly rather than smoothly, since a gradual transition on a sniper rifle would leave the player looking through a constantly changing zoom level right when they are trying to track a moving target, while an instant change paired with the vignette reads more like a scope snapping into place.

4.6 Rendering Configuration for the Player Camera

Beyond Cinemachine's own settings, the underlying camera is configured with a few options specific to Unity's rendering pipeline [22]. It is set up as the main camera responsible for drawing the whole scene, rather than one of several layered cameras, and the game's interface is drawn separately on top of it rather than through additional camera layers.

The camera's culling mask decides which layers it can see, based on the same layer system used for weapon hit detection, and this is used to keep certain debug or editor only elements out of the player's view. Anti-aliasing is kept simple, multi-sample anti-aliasing is turned off, and the cameras either use no post-process anti-aliasing or a lightweight, single-frame technique called FXAA. A more advanced technique that blends information across frames was tried and dropped, since it assumes the geometry on screen was already present in the previous frame, which does not hold well in a scene where enemies are constantly appearing and disappearing and a newly generated enemy can show up mid-fight. A small amount of aliasing on distant trees was judged a smaller cost than the smearing artefacts the more advanced technique produced around moving enemies.

4.7 Extending the Starter Asset

It is worth being clear about where the starter asset's work ends and this project's own work begins, since a meaningful share of the engineering in this thesis sits exactly at that boundary. The starter asset itself only covers movement, camera control, and basic input wiring. It does not include a weapon or combat system, enemy AI, health, pickups, an interface, or any of the level content described later in this thesis. All of those were built specifically for this project, with the starter asset providing only the base layer of movement and camera feel.

This division of effort was a deliberate decision made early on. Movement and camera control matter a great deal to how a shooter feels, but they are also a well understood problem compared to this thesis's actual research question. Building on an officially maintained starter asset for that foundational layer reduced the risk of getting stuck on basic control feel, and left more time for the systems that are specific to this project, the enemy AI, the combat, and above all the generative pipeline.

4.8 Tuning Process

Several movement and camera parameters, the step offset and slope limit on the character controller, and the camera's vertical clamp and turn speed, were tuned by hand rather than left at the starter asset's defaults, since the forest level's uneven terrain exposed problems that the original flat test level never showed. Each change was tested directly by walking the same handful

of terrain features in the forest, a shallow slope, a small ledge, and a patch of dense undergrowth, checking after every adjustment that movement still felt predictable and that the original problem had not simply moved somewhere else. This was repeated several times over as the level itself changed from an early flat prototype into its final, more uneven form.

Chapter 5 — Environment and Level Design

5.1 Two-Stage Level Development Strategy

Sharp Shooter's level was built in two distinct stages, following the common practice of blocking a level out roughly before finishing its art [6]. An early prototype was used to test movement, sightlines, and enemy navigation quickly and cheaply, followed by a visually finished outdoor forest environment used for the final build. This let early iteration stay fast, since geometry could be reshaped in minutes directly inside the editor, while the slower work of art direction, lighting, and colour grading was left until the underlying gameplay already worked.

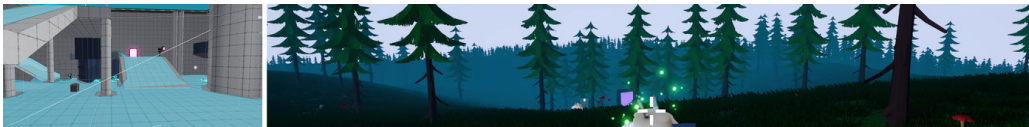


Figure 5 Side-by-side comparison of the early prototype level and the final forest environment

5.2 Prototyping with ProBuilder

The prototype was built with ProBuilder, Unity's built in tool for modelling directly inside the editor without needing separate 3D software [23]. It was used to block out walls, floors, ramps, and cover using its editing tools alongside a few basic shapes.

Most of the prototype was built from a small set of parametric shapes rather than freely modelled geometry. Of the sixty-five shape objects in the most developed prototype scene, forty-one are pipes, fourteen are cubes, eight are cylinders, and one each is a door and a staircase. The pipe shape did most of the work, a hollow cylinder with an adjustable radius, thickness, and height, useful equally as a column, a barrier, or a piece of clutter, and its hollow interior gave the player something to walk around rather than simply past. Cubes provided walls, floors, and cover, while the door and staircase shapes were only used once each, where placing a ready made shape was faster than modelling one.

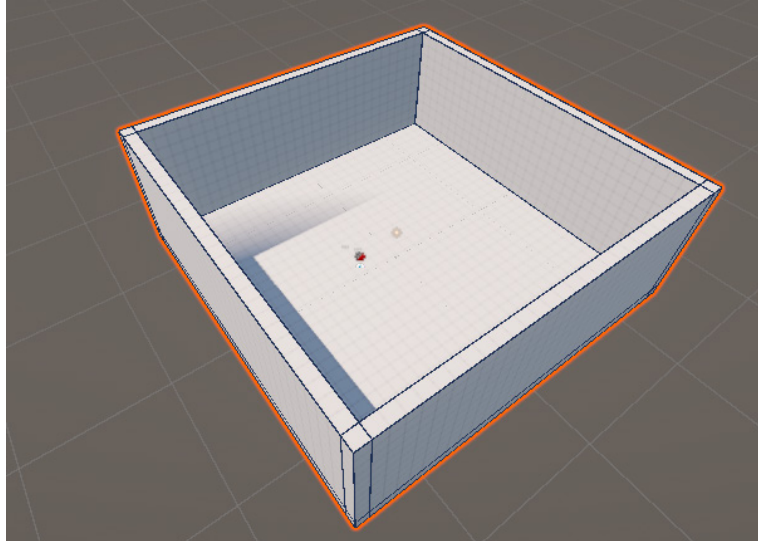


Figure 6 ProBuilder editing session inside the Unity Editor

- Pipe shape, a hollow cylinder used for columns, rings, and clutter, the most used shape in the prototype.
- Cube shape, used for walls, floors, and cover.
- Door and staircase shapes, used sparingly where a ready made shape was faster than modelling one.

Once a piece of geometry was finished as a prefab, it was sometimes fully unpacked so that it no longer tracked its source prefab, which was useful whenever a specific piece of the level needed a one off change without affecting the shared prefab it came from.

5.3 Transition to the Forest Environment

Once the core mechanics, movement, combat, and enemy navigation, were validated in the prototype, the level was rebuilt as an outdoor forest for the final build, matching the story described later in this thesis. The transition kept the underlying layout, the same walkable space and sightlines that had already been tested, while replacing the flat blocked out surfaces with real terrain, trees, and foliage, and rebaking the navigation mesh over the new geometry. This was closer to a replacement than a refinement, as the final scene has no ProBuilder geometry left in it at all, its ground being an ordinary Unity terrain with painted textures and placed trees, while the most developed prototype scene still has all sixty-five of its original shapes. What

survived the transition was the layout itself, the walkable area, where enemies could appear around the level, and the overall sightlines, rather than any of the geometry that had expressed it.



Figure 7 The final forest environment as seen from the player's first-person perspective

Part of this transition involved bringing in outside environment assets and adjusting their materials to work correctly with the project's render pipeline, using Unity's own conversion tool where a direct match existed and adjusting the remaining materials by hand [22].

5.4 Post-Processing and Colour Grading

Post-processing is applied through a global volume that affects the whole scene [22]. Six effects are active in the shipped build, bloom, tonemapping, colour adjustments, white balance, a shadows and highlights adjustment, and a vignette. Depth of field was tried during early look development but switched off, since blurring anything outside a narrow focal range works against a shooter where the player is trying to pick out targets through tree cover, and motion blur was avoided for a similar reason, since it made fast approaching enemies harder to read rather than adding to the sense of speed.

Colour grading mostly comes from the white balance and shadows and highlights controls. White balance shifts the whole image warmer or cooler, and was used here to give the forest a slightly cool, atmospheric tone that matches the story. The shadows and highlights control

adjusts colour separately across the darkest, mid, and brightest parts of the image, giving more control over the overall mood than a single colour adjustment would.

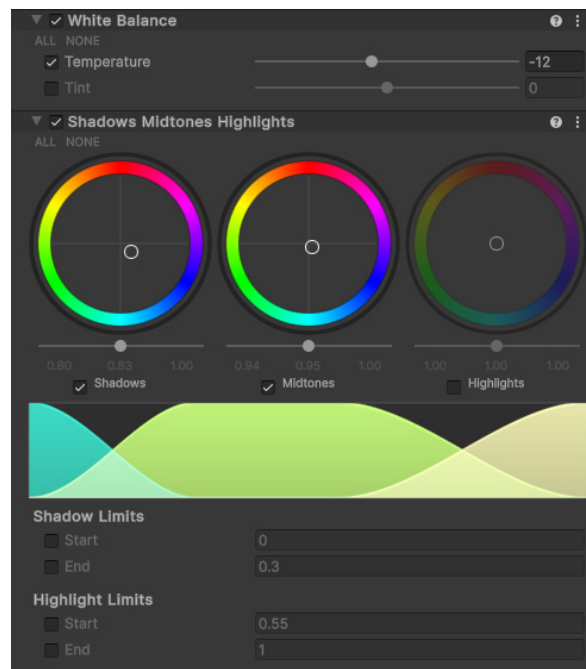


Figure 8 Global Volume inspector

5.5 Particle Effects

Particle effects are used throughout the environment and combat for smoke, muzzle flashes, and impact sparks, using Unity's built in particle system. Basic settings such as emission rate, lifetime, speed, and size control how each effect looks, alongside settings for gravity and collision with the rest of the scene.

One small setting turned out to matter more than expected. A particle system can be told to do nothing once it finishes playing, to destroy itself automatically, or to call back into a script. All three of the game's combat effects, the explosion that plays when an enemy dies, and the two impact effects that play on a landed hit, are set to destroy themselves once finished, so that spent effects clean up after themselves rather than piling up.

This matters because of how often these effects fire. Over a long fight the explosion can play hundreds of times and the impact effects once per landed shot, so anything left behind by mistake would sit invisibly in the scene for the rest of the session. This kind of leftover object is

easy to miss during testing, since it has no visible presence and no obvious per-frame cost, and only becomes noticeable in a session longer than a typical test run.

All three effects are spawned at a world position without being attached to anything, an impact effect appears at the point a shot landed, and an enemy's explosion appears at the enemy's own position, rather than following it as a child object. This kept things simple and avoided a subtlety that would otherwise have mattered given how much generated enemies can vary in size.

5.6 Performance Considerations in the Forest Environment

Dense outdoor vegetation is a common source of slowdown in Unity projects, and a few choices earlier in this chapter were made partly with that in mind. Tree and foliage density in the final level was kept moderate, enough to preserve the cover and broken sightlines the level design relies on without pushing rendering cost high enough to hurt frame rate on ordinary hardware, and simplified collision shapes were used on background trees that the player never needs to interact with directly, keeping accurate collision reserved for the player, the enemies, and their immediate surroundings.

Since lighting is realtime rather than baked, described above, its cost is paid every frame rather than moved to a one time precompute. This is kept manageable by the level's simple lighting setup, a small number of directional lights and the exponential squared fog described earlier stand in for detailed indirect lighting, leaving more of the frame budget for the enemy AI's navigation queries during combat, and for the occasional extra cost of a newly generated enemy being imported mid-session.

5.7 Art Direction Consistency with Generated Content

One consideration specific to this project, and not something a typical level design discussion has to deal with, is that the forest's art direction had to stay visually coherent alongside enemies whose appearance is not decided in advance. Rather than committing to a narrow, distinctive visual style that a generated enemy could easily clash with, the environment's colours and lighting were kept fairly neutral, natural greens and cool ambient tones rather than a strong

stylised look, so that a wide range of generated designs, mechanical or organic, would still read as belonging to the scene without the environment itself needing to change for every new enemy a player might describe.

Chapter 6 — Enemy AI and Navigation

6.1 Two Kinds of Enemy

Sharp Shooter's enemies come in two kinds that differ only in how their appearance is produced, while sharing the same combat and navigation code underneath.

Scripted enemies are entirely hand made, with a fixed model and fixed base statistics. Two exist, a hovering robot that chases the player and explodes on contact, and a stationary turret that tracks the player and fires at a steady rate. The robot is never placed directly in the level, instead it is produced by spawn gates placed around the level, each of which creates a new robot every few seconds for as long as the player is alive.

Generated enemies are the subject of Chapter 9. Their model is created at runtime from a text description the player types, downloaded, and assembled into a playable enemy on the spot. Like the hand made robots, they reach the level through a gate rather than being placed one by one, creating a generated enemy produces a visible spawn gate that keeps emitting that design at the same pace as the hand made gates.

A third mechanism cuts across both kinds without being a kind of its own, a difficulty setting, covered later in this chapter, that scales every enemy's health and speed and every gate's spawn rate by the same factors regardless of which kind the enemy is. Since these factors are applied the moment an enemy is created rather than being baked into any particular design, a generated enemy on Hard difficulty is exactly as tough and as fast as a hand made one.

What makes this possible is that both kinds end up as the same basic object at runtime, a root object carrying a navigation agent, a health component, the chase and detonate behaviour, and two capsule colliders. The only difference between the two kinds is where the model attached to that root came from. Because the rest of the enemy AI and combat systems only ever interact with this shared setup, they need no special handling for either kind, a generated enemy uses the exact same underlying classes as a hand made one, simply attached to it at runtime instead of built into a prefab ahead of time.

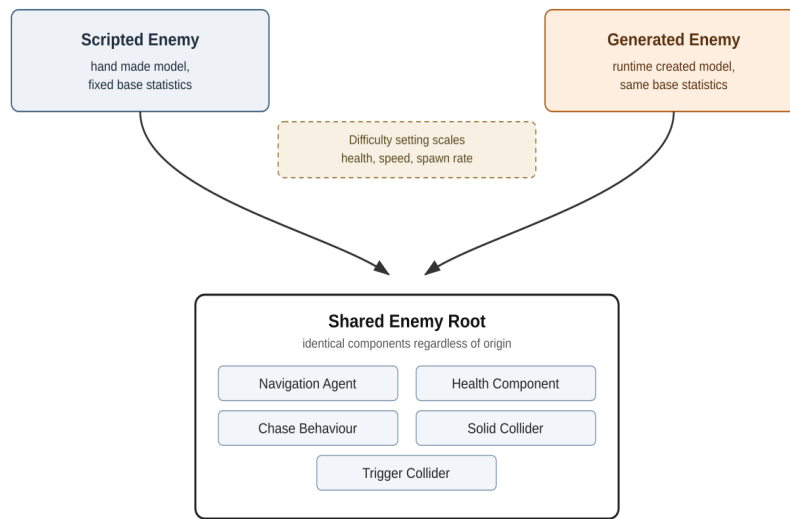


Figure 9 Diagram of the two enemy kinds

6.2 Navigation Surface and Baking

Enemy navigation is built on Unity's NavMesh system, which computes a walkable surface for AI agents to move across [3]. A navigation surface component defines which geometry counts as walkable, based on what is included and which layers are considered, so that floors, ramps, and terrain are included while clutter that should not be walked on is left out. A few settings on that surface, such as agent height, radius, and the steepest slope it can climb, determine what a given kind of agent is allowed to cross, following the general layered approach to movement described by Millington and Funge [2].

The final forest level uses no navigation links at all, generated or manually placed. This follows directly from how the level is shaped, the walkable area is one continuous surface, and the things that break it up, trees, rocks, and the spawn gates themselves, are simply carved out as holes rather than splitting it into separate regions. Every point on the surface can therefore be reached from every other point on foot, which matters later when a spawn gate is placed at a computed position and needs to be sure that whatever it creates there can actually reach the player.

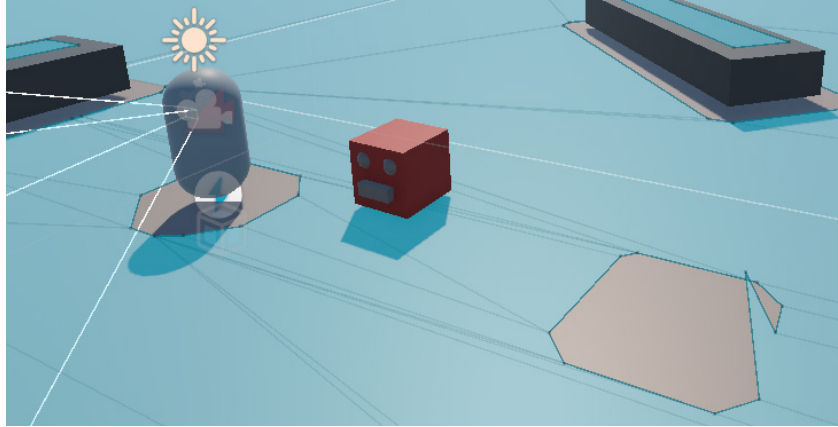


Figure 10 A baked NavMesh area

One property of the baked surface turns out to be useful later. Since it only covers walkable terrain and excludes trees, rocks, and anything outside the playable area, the navigation surface is effectively a map of the open ground, any point on it is by definition a point somewhere the player could actually walk. The system that places generated enemy gates takes advantage of this directly rather than doing any separate analysis of the terrain, it finds the middle of the baked surface and then simply snaps that point onto valid ground.

6.3 Chase Behaviour

Enemies chase the player using the navigation agent attached to their root object. The player's position is set as the agent's target, and the agent's own pathfinding keeps that path up to date as the player moves, following the general approach described for game AI generally [2] and for Unity specifically [4]. A few settings on each agent, its speed, acceleration, turning speed, and stopping distance, control how aggressively it closes the distance and how near it gets before stopping.

Two smaller decisions were driven by how many enemies this game can have active at once. First, an enemy's path is not recalculated every single frame, since doing so for every active enemy would be wasteful and would make no visible difference, a path that updates several times a second already looks the same to the player as one that updates every frame. Each enemy instead recalculates on a short timer with a randomised starting offset, so that a group of enemies created around the same time does not all recalculate on the exact same frame.

Second, enemies use a cheaper, lower quality form of local avoidance rather than the default, since the benefit of enemies smoothly stepping around each other barely matters in a game where enemies explode on contact and rarely stay clustered for long. Enemies are also set not to collide with each other at the physics level at all, removing a good amount of otherwise wasted work each frame.

Enemies have no detection radius and do not wait to notice the player, an enemy begins chasing the moment it is created and continues until it dies. This follows directly from how encounters are structured, enemies are not placed around the level waiting to be found, they are produced continuously by gates, so the usual pacing role a detection radius would play is instead played by how quickly gates produce enemies.

- Setting the destination, called on a staggered timer rather than every frame.
- Speed, acceleration, turning speed, and stopping distance, govern how an enemy approaches the player.
- Lower quality obstacle avoidance, traded for lower per-agent cost.
- Enemies do not collide with each other, removing unnecessary physics work.

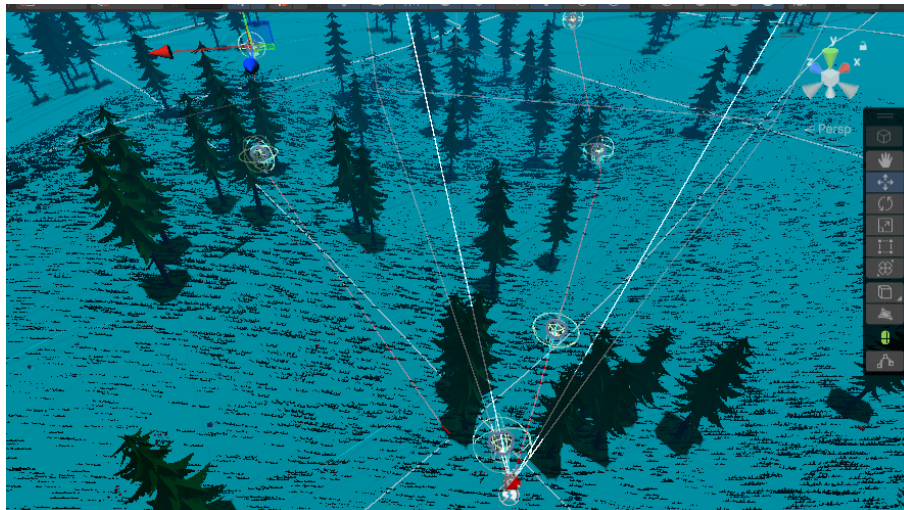


Figure 11 Scene view during Play Mode showing several agents' current paths converging on the player

6.4 Facing and Orientation

A chasing enemy never rotates itself directly, its facing is simply a side effect of navigation, the navigation agent turns the whole object toward the direction it is currently walking, which for an enemy that only ever chases the player is effectively always toward the player. No separate facing code runs at all, which is part of why a generated model of any proportions can be dropped into this same behaviour without adjustment.

The one exception is the stationary turret, which explicitly turns its head to face the player each frame and uses the same facing when it fires a projectile, so that the projectile always leaves in the direction the turret was actually aiming rather than some earlier, stale direction. Each enemy carries a wider trigger collider around its solid one, and contact with the player through that trigger applies damage and destroys the enemy in the same moment, an enemy attacks simply by reaching the player.

Applying damage at the moment of contact, rather than through a separate area effect centred on the enemy, was a deliberate fix made during development, since the earlier approach broke for any model whose centre point did not sit where it visually appeared to, which turned out to matter a great deal once models could be generated rather than only hand placed.

The second kind of check is used only when placing a new spawn gate, to make sure the chosen spot is actually clear of trees, walls, or other gates before a gate is placed there, walking across the navigation surface until a clear spot is found.

6.5 Enemy Health

Every enemy carries the same health component, whether hand made or generated, and it does more than simply track a number. At the moment an enemy is created, it reads its base health value, three points in the shipped build, and multiplies it by the current difficulty's health factor, so an enemy that the starting pistol can kill in two shots on Easy takes noticeably more shots on Hard. It also adds itself to a running count of live enemies, which is what drives the enemy counter shown to the player.

Taking damage simply reduces that health value, and once it reaches zero the enemy plays its destruction effect, removes itself from the live count, and is destroyed. This same sequence runs whether the enemy died from being shot or from exploding on contact with the player, and a simple guard makes sure it can only ever run once per enemy, which matters since both of those events could in principle happen on the same frame.

One enemy is allowed to skip the difficulty scaling entirely, the generated enemy gate itself, which always has exactly one point of health so that it can be destroyed in a single hit at any difficulty. The idea is that a gate is a target the player is meant to be able to shut down decisively once they have found it, rather than something that should get harder to destroy just because the overall difficulty is higher.

Because this component is identical for both kinds of enemy, a generated enemy takes damage, dies, and is removed from the count using exactly the same code as a hand made one.

6.6 Runtime Difficulty Scaling

A difficulty setting, chosen once before play begins, is what turns a single choice into the concrete multipliers the rest of the game reads. Three levels are offered, and each one moves three separate values at once rather than a single difficulty number.

	Spawn interval	Enemy health	Enemy speed
Easy	×1.00	×1.0	×1.00
Normal	×0.60	×2.0	×1.20
Hard	×0.35	×3.0	×1.45

Moving all three together was a deliberate choice. Scaling health alone would just make higher difficulties feel slower and more tedious, since each enemy simply absorbs more ammunition, and scaling spawn rate alone would raise pressure without making any single mistake more costly. Combining a shorter spawn interval with tougher, faster enemies changes the whole character of a fight, on Hard the player cannot clear enemies as fast as the gates

produce them and has to decide which gates matter most, while on Easy they can simply fight as enemies appear.

A few smaller details matter for how this setting behaves. The player's choice is kept in a way that survives a level restart, and is explicitly reset whenever the difficulty menu comes back up, so a stale choice from an earlier session can never leak into a new one. Every spawn gate also waits for a difficulty to be chosen before producing anything at all, so the player never returns from the difficulty menu into a fight that has already started without them. The difficulty menu itself is built and handled slightly differently from the rest of the interface, for reasons that are really just about avoiding interference with the interface described in Chapter 8.

Chapter 7 — Combat, Weapons and Interaction Systems

7.1 Hit-Scan Combat

Sharp Shooter's weapons work by casting an invisible ray forward and checking what it hits, rather than by simulating a travelling projectile, which is the usual approach for a shooter where precise, instant hit detection matters more than travel time [2], [1]. When the ray hits something, the game learns which object was hit, the exact point of contact, and the distance travelled, and the rest of the combat system uses all three.

For the player's shots specifically, the ray starts at the camera and travels straight ahead, rather than starting at the weapon model, which keeps hit detection lined up exactly with the crosshair at the centre of the screen no matter where the weapon itself sits on screen.

7.2 Layers and Hit Filtering

Unity's layer system is used throughout the combat code to group objects for the purpose of filtering hits and collisions. Sharp Shooter adds three layers of its own on top of Unity's built in ones, Player, Enemy, and Weapon, and the Enemy layer in particular does most of the work, it is applied to every generated model as it is assembled, it is what stops enemies from colliding with each other. It is also what a spawn gate placement check excludes so that existing enemies do not block a new gate from being placed.

Each weapon has its own setting for which layers its shots can register against, rather than that being fixed in code. All three weapons in the finished build are set to test against everything, since the level has no category of solid object a shot is meant to pass through, and the two exclusions that would otherwise matter, the player's own body and any trigger volume in the level, are already handled more directly, the ray starts at the camera rather than the weapon so the player is never in front of it, and the query is told to ignore trigger colliders entirely.

That last setting matters a great deal in practice. Every enemy carries two colliders, a solid one for weapon hits and a wider trigger one used for detecting contact with the player, and pickups are trigger volumes as well. Without telling the query to ignore triggers, a shot would register against an enemy's outer trigger collider before reaching its solid body, making hits

register from further away than the enemy visually appears, and would also stop dead on any pickup in the way.

7.3 Gun Setup and Presentation

The weapon model is parented to the camera so that its position and rotation stay correctly aligned with the first-person view as the camera turns, with small position and rotation adjustments made so it sits naturally on screen. The weapon's actual hit detection comes entirely from the ray described above, the visible model itself has no collision responsibility of its own.

Weapon animation is kept deliberately simple, a single firing animation restarts from its first frame on every shot, at the same moment as the muzzle flash and the camera recoil. Restarting the animation from the beginning each time, rather than relying on an ordinary animation transition, is what lets rapid automatic fire look correct, a transition would otherwise be swallowed while the previous one was still playing, making a burst of shots look like only one.

There is no reload animation, because there is no reload mechanic at all, a weapon's magazine is filled when it is picked up and only refilled again by ammunition pickups. Shots simply stop firing once the count reaches zero. This was a deliberate scope decision, a reload cycle would have added an animation state, an input action, and a timing window to tune, for a mechanic that has nothing to do with what this thesis is actually about.

7.4 Damage and Impact Effects

When the player fires, the resulting hit check is evaluated once per shot, if it reports hitting something on the Enemy layer, that enemy's health component is found and told to take the weapon's configured amount of damage, reducing its health and, if that reaches zero, starting its destruction. This single, simple path is what lets the same weapon code correctly damage either kind of enemy without needing to know which kind it is.



Figure 12 A burst of destruction particles as an enemy is defeated mid-fight, with a second enemy still approaching in the distance

The exact point a shot landed is also used, separately from applying damage, to place a small visual effect exactly where the shot hit. This effect is not rotated to match the surface it struck, since it is a simple, symmetrical burst with no real sense of up or down, and rotating it correctly would have meant treating terrain hits differently from hits on a generated enemy, whose surface can be unpredictable.

Destroying an enemy triggers a separate explosion effect, created by the health component itself at the moment the enemy's health reaches zero, rather than by the weapon, which is why an enemy that explodes on contact with the player produces the same effect as one that was simply shot. This effect cleans itself up automatically once it finishes playing, following the same particle setting described in the previous chapter. The reference to this explosion effect lives on the health component itself, which is what lets a freshly generated enemy receive the exact same explosion as a hand made one, it is simply handed to the enemy's health component as the enemy is assembled.

7.5 Pickups and Interaction

Collectible items in the level, weapon pickups and ammunition pickups, share a common base that cannot be used directly on its own and exists purely to be built on by other classes [15]. That shared base handles everything every pickup has in common, a slow idle rotation so the item stands out visually, detecting the player through a trigger collider, finding the player's active weapon component, and destroying itself once collected. Between detecting the player and

destroying itself, it calls a single method that each specific kind of pickup fills in with its own effect.

Resolving the player's active weapon component once, in the shared base, rather than in each pickup individually, is what keeps each specific pickup so small, an ammunition pickup's entire job is one call adding its configured amount, and a weapon pickup's is one call switching the player to its configured weapon.

- The shared pickup base, handling idle rotation, detecting the player, finding the active weapon, and destroying itself.
- Weapon pickups, which switch the player to their configured weapon.
- Ammo pickups, which add their configured amount of ammunition.

The weapon pickups also make use of prefab variants, described in Chapter 8, a shared base pickup prefab defines the collider, rotation, and visual framing, and the pistol, machine gun, and sniper rifle pickups each override only the model shown and the weapon they grant. Adding a fourth weapon to the game would only need one new data asset and one new variant, no new code at all.

7.6 Numeric Display and Small Code Conventions

Ammunition and other numeric values on the heads up display are formatted with a fixed minimum width, so a single digit value such as five is padded to display as zero five rather than five, keeping the width of these numbers visually stable as they change during play rather than letting the layout shift around.

A couple of small code patterns recur where a value, rather than a whole separate branch of logic, depends on a simple condition. A short conditional expression is used mainly as a fallback, for example a spawn gate using its own position when it has no dedicated spawn point assigned, or a material falling back to a default when the preferred one cannot be found. A different, slightly larger pattern is used for the three difficulty multipliers described in the previous chapter, mapping the chosen difficulty directly to a factor rather than writing out a longer chain of

separate checks. Both are only used for simple, single level decisions, anything more involved is written as ordinary, explicit code.

7.7 Why Hit-Scan Rather Than Projectiles

Hit-scan combat was chosen after trying both approaches on the early prototype level. A travelling projectile turned out to interact poorly with the short, close range engagements typical of the forest's broken sightlines, at that range travel time is barely noticeable to the player, while a full projectile system adds real complexity, its own objects, its own colliders, its own lifetime management, for very little gain in how the game actually feels. Hit-scan combat instead gives immediate, unambiguous feedback tied directly to the moment of firing, which mattered more for a game whose combat needed to read consistently across both hand made and generated enemies, including enemies whose exact shape is not known until the moment they are created.

This choice also kept the combat system simple to connect to the generative pipeline. Since a hit-scan shot simply tests against whatever collider a generated enemy happens to have, no extra work was needed to make generated enemies correctly hittable, the exact same check and the exact same layer filtering used against hand made enemies works without any change at all.

7.8 Balancing Weapon Damage Against Enemy Health

Weapon damage and enemy health were tuned together rather than separately, since the two numbers only really matter relative to each other, in terms of how many shots a fight actually takes. In the shipped build an enemy's base health is three points, and the three weapons sit at deliberately different points against that number, the pistol deals two damage at a moderate pace, the machine gun deals one damage very rapidly, and the sniper rifle deals five, enough to kill any enemy outright at any difficulty, but only once a second. Taking the pistol as the baseline weapon the player starts with, an enemy falls in two shots on Easy, three on Normal, and five on Hard once the difficulty multiplier from the previous chapter is applied.

The sniper rifle's guaranteed kill is intentional, it is the only weapon that zooms, and a weapon that asks the player to stop moving and aim carefully earns a reliable kill in return. The same base health value applies to generated enemies without any adjustment, since a generated enemy's

health component is set up exactly the same way as any other enemy's, which keeps the number of hits needed to defeat a freshly generated enemy consistent with the rest of the game even though its design was never seen ahead of time.

Chapter 8 — UI, Data Architecture and Scriptable Objects

8.1 Canvas and Event System

Sharp Shooter's interface is built on Unity's UI system, and splits into two groups that are built quite differently. The heads up display, the pause screen, and the Game Over screen are all set up in the editor as ordinary scene objects. The enemy creator panel, the difficulty menu, and the victory screen are instead built entirely from code while the game is running.

Interface elements built in the editor rely on an event system object, which receives input and turns it into interface events such as click and hover. Since the enemy creator is built at runtime and may be the very first interface to appear in a scene, it creates this event system itself if one is not already present.

That arrangement caused a real problem during development, worth mentioning because it is specific to interfaces built at runtime. With the enemy creator owning the event system, the separately built difficulty menu found its own buttons intermittently unresponsive, since two independently built interfaces, appearing and disappearing on different schedules, were competing for the same shared input handling. Rather than trying to coordinate the two, the difficulty menu and the victory screen were changed to check for clicks on their own, each frame reading the mouse position directly and testing it against their own buttons. This duplicates a small amount of logic across the two screens, but makes each one self contained regardless of what else in the scene is handling input at the time, which was clearly the right trade for two simple screens with only a couple of buttons each.

8.2 Grid Based Layout

The enemy creator's gallery of saved characters, described later in this thesis, is laid out using a grid layout component, which automatically arranges its contents into rows and columns rather than needing each thumbnail positioned by hand. The grid is set to two columns, filling left to right and then wrapping downward, and is paired with a couple of further components that let it grow to any number of saved characters, one that expands the content area as rows are added, and a scrollable view that turns that growth into a scrollable list rather than letting it overflow.

In practice this means the gallery simply keeps working correctly as new characters are created during a session, without a single line of layout code needing to know how many there will eventually be.

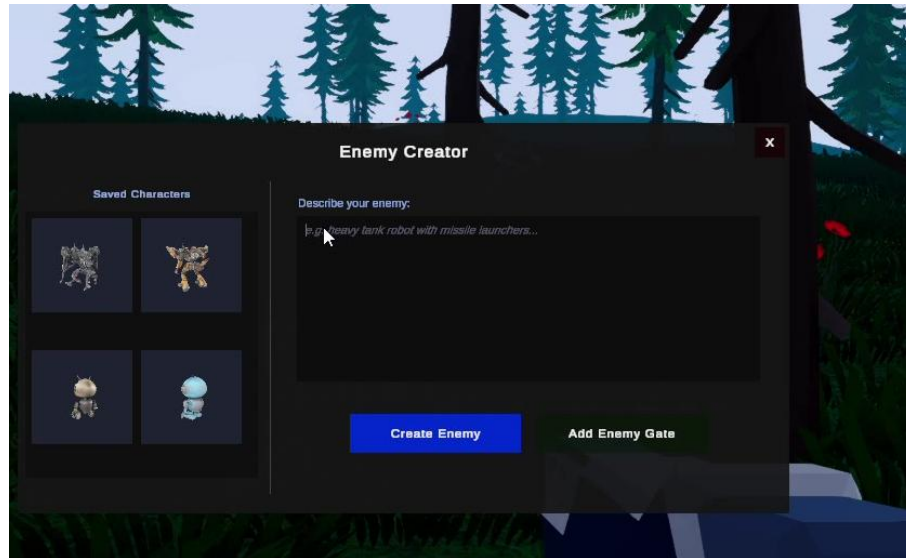


Figure 13 The grid based gallery of saved characters inside the Enemy Creator interface

8.3 Constrained Values in the Inspector

A small number of numeric fields in the project are shown in the editor as a slider with a fixed range rather than a free numeric field, and this is only used where the value genuinely has a hard limit rather than simply a sensible default. The clearest example is the player's starting health, limited to between one and ten, since the health bar itself is drawn as a row of shield segments whose count comes directly from that number, so a value outside that range would either overflow the display or leave it empty. Elsewhere in the project, enemy health, weapon damage, and agent speed are left as ordinary fields, since none of those values have a limit the interface actually depends on.

8.4 Scriptable Object Data

A ScriptableObject is a Unity class that, unlike an ordinary script attached to an object in the scene, exists as its own standalone asset that can be referenced from many places at once and edited directly without touching code [17].

Sharp Shooter uses this pattern for exactly one thing, weapons. Each weapon asset holds its damage, fire rate, magazine size, whether it is automatic, its zoom settings, and references to its model and effects, and three such assets exist for the game's three weapons. Because every part of the game that needs a weapon's data reads the same shared asset, the starting weapon, its pickups, the ammunition display, and the damage calculation, adding a new weapon only means creating one new asset and one new pickup variant, with no new code required anywhere.



Figure 14 The three WeaponSO data assets in the Unity Project window, Machine Gun, Pistol, and Sniper Rifle

It is worth being equally clear about where this pattern is not used. Enemy statistics are plain fields on the enemy components themselves, scaled at spawn by the difficulty setting, pickup amounts are plain fields on the pickups, and a generated enemy's information is not stored as an asset at all, it lives in the name of the file saved to disk. Each of these is a simpler design than the weapon system, chosen because there was nothing for a shared data asset to actually share, enemy statistics vary along one setting rather than by type, and generated content has no asset to author ahead of time in the first place.

8.5 Prefab Variants

Where a family of related objects shares most, but not all, of its structure, the project uses prefab variants rather than fully separate prefabs. A variant keeps a live connection to its base prefab, automatically inheriting the base's structure while allowing specific parts, such as a material or a single component, to be overridden. Since a change to the base prefab automatically applies to every variant, this keeps several closely related objects in sync without repeated manual work.

The clearest example is the weapon pickup family, a base pickup prefab defines the shared collider, rotation, and visual framing, and the three specific pickups override only the model

shown and which weapon they grant. Generated enemies, by contrast, do not use prefab variants at all, since a generated enemy has no pre-existing prefab to vary from in the first place, it is instead assembled piece by piece at runtime, which solves the same underlying problem in a different way.

8.6 Interface State During Long Requests

A recurring challenge in the enemy creator panel is representing something that is still in progress rather than finished within a single frame. While a generation request is outstanding, which can take tens of seconds, the panel needs to show that something is happening, prevent the player from starting a second request at the same time, and return cleanly to a usable state whether the request succeeds, fails, or times out.

This is handled with a single flag on the panel's controlling script, set as soon as a request begins and cleared on every path out of it. While it is set, both buttons on the panel become unusable and a coloured status line reports what is currently happening, grey while in progress, green on success, and red on failure, so the outcome is clear at a glance without needing to read the message itself.

Every one of the request's exit points restores this state properly, whether the request failed outright, came back with something other than a valid model, failed to save to disk, or simply succeeded. An empty prompt is instead caught immediately when the button is first pressed, before any request even begins, which is the simplest way to handle that particular case and avoids ever needing to clean it up later.

8.7 Updating the Heads Up Display Efficiently

Elements on the heads up display are updated only when their value actually changes, rather than being redrawn every single frame. The enemy counter is only rewritten inside the exact method that changes the live enemy count, and the ammunition display only inside the method that changes the ammunition count. Since updating a piece of text has a real, if small, cost each time it happens, and the game can have many enemies active and pathing at once, avoiding unnecessary updates helps keep that cost out of the main loop.

The enemy counter also carries two small safeguards, both found through actually playing the game rather than through planning ahead of time. It is never allowed to display a negative number, in case of any bookkeeping mistake, and it is frozen entirely once the level is won, since the win is followed by a short delay while the final explosion plays out, during which a straggling enemy could otherwise still die and push the counter below zero, or worse, could still reach and kill the player and produce a Game Over screen appearing right on top of a victory. The player's own health is protected the same way, ignoring further damage once the level has already been won.

Chapter 9 — Runtime Generative Enemy Pipeline

9.1 Design Goals and Constraints

The generative enemy pipeline is the central technical contribution of this thesis, a system that turns a short, free text description from the player into a fully textured, ready to fight enemy, created and dropped straight into the running game with no offline preparation step at all. A few constraints shaped its design and ruled out some more obvious approaches.

- Speed, the whole process from typing a description to a spawned enemy needed to finish within a time the player could reasonably tolerate mid session, which ruled out slower, optimisation based 3D generation methods despite their higher potential quality [14].
- Cost, the pipeline had to run on free or near free infrastructure, which ruled out renting dedicated cloud hardware and led directly to using Google Colab's free notebooks.
- Compatibility, the output had to fit into the same navigation, combat, health, and difficulty systems already built for hand made enemies, reusing those systems rather than building parallel ones.
- Robustness, since the pipeline depends on infrastructure outside the game's own control, the game had to stay fully playable using hand made enemies alone even if the pipeline was slow, degraded, or completely unavailable.

These constraints led to a two stage pipeline, a text-to-image model turns the player's prompt into a single reference image, and an image-to-3D model turns that image into a textured mesh. This split was preferred over a single model doing both steps at once for a few reasons. Each stage could be a well documented, purpose built model chosen on its own merits, rather than one larger system that has to be good at both halves of the problem simultaneously. The middle of the pipeline also stays inspectable, a failed generation can be diagnosed simply by looking at the image the first stage produced, which narrows the problem down to one of the two models immediately. And it avoided depending on a commercial API, an alternative that was tried and is discussed later in this thesis.

This split also turned out to have a benefit that was not part of the original reasoning. Because the two stages are naturally separate, they could later be split further, onto two separate machines, once it became clear that the platform this project runs on could not give both of them fast hardware at the same time. A single, fused model would not have offered that same flexibility.

9.2 Stage One, Turning Text into an Image

The first stage uses Stable Diffusion XL [11], a diffusion model built on the architecture described by Rombach and colleagues [10], in which the gradual denoising process described by Ho, Jain, and Abbeel [9] happens inside a compressed representation rather than directly on full resolution pixels. It runs in a slightly reduced precision mode, a concession to the limited hardware available rather than a choice about output quality.

The model is loaded once when its notebook starts and stays loaded for the rest of that session, so individual requests do not pay a separate loading cost each time, which matters since loading it from scratch takes minutes on its own.

This stage is not called directly by the game. It is called by the reconstruction stage described next, which acts as the pipeline's actual entry point. Its settings are therefore fixed rather than sent by the game each time, thirty denoising steps at a resolution of 1024 by 1024, with a fixed random seed. The game itself only ever sends a prompt, nothing else varies from one generation to the next.

That fixed seed is worth explaining, since it makes the pipeline behave in a way generative tools usually do not, the same prompt always produces the same enemy. This is a real trade. It costs the player the ability to simply try again and get a different result for a description they liked, which is normally the most natural thing to want from a generative tool. What it buys in return is that a given result can be reproduced exactly, which matters for the testing described later in this thesis and for anyone trying to check a specific result reported here.

The prompt the player types is passed to the model exactly as written. What is fixed instead is a separate negative prompt, a standing list of qualities to avoid, blur, low quality, watermarks,

text, distortion, and oversaturation. Its purpose is to raise a quality floor rather than to describe the subject, and two of its entries matter specifically for the next stage, since text and watermarks tend to reconstruct as raised bumps on the model's surface rather than being recognised as flat markings.

The example prompt used throughout this thesis, describing a small white AI companion robot, illustrates the kind of description that works well, a single subject with clear material, colour, and lighting details, and nothing else competing for the model's attention.

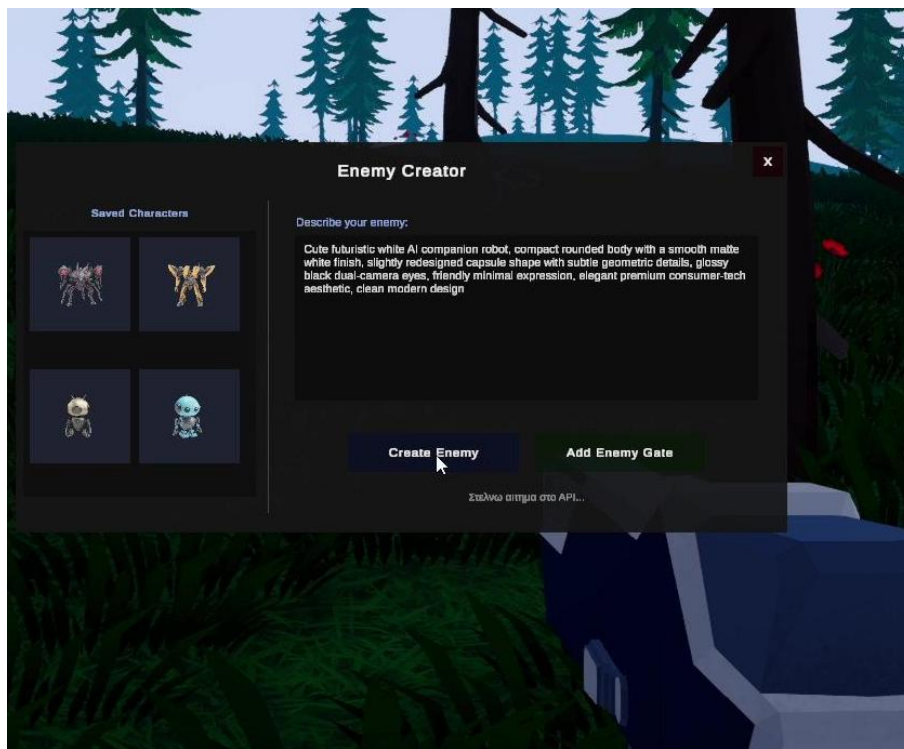


Figure 15 The in-game Enemy Creator prompt field showing the example prompt used to generate the white companion robot enemy

9.3 Stage Two, Turning an Image into a 3D Model

The second stage turns the image from the first into a 3D mesh using TripoSR [13], a model that produces a full mesh in a single pass rather than through the much slower, repeated refinement process older methods such as DreamFusion use [14]. A single pass model has a cost that is bounded and predictable no matter what hardware it runs on, whereas an older, iterative method's cost depends on how long it takes to converge. On the ordinary processor this project actually runs it on, rather than the graphics hardware such models are normally benchmarked

on, that bounded cost comes out to a few minutes, slow enough to be felt, but fast enough that a player can trigger a generation and fight the result within the same session.

TripoSr runs in its own separate notebook, apart from the image stage, for the hardware reasons already mentioned. This notebook is also the pipeline's actual entry point, the address the game calls is served here, and this notebook in turn calls the image notebook internally, so that from the game's point of view the whole two model process still looks like a single request and response.

Loading the model once, inside the same process as its own small web server, was the result of an earlier approach failing. Running it as a separate process for every request, the way its own documentation suggests, reloads the model from scratch each time and needs a second Python process running alongside the web server, which used more memory than the free hardware allows and caused requests to be killed partway through. Loading it once avoided both problems at once.

The mesh itself is then extracted at a reduced resolution, since the model's default setting evaluates far more points than the free hardware's memory can handle without the session being dropped mid-request. The lower setting used here is the highest one found to complete reliably. Finally the mesh is oriented and saved in the compact GLB format. This format stores colour directly on the mesh's vertices rather than through a separate image texture, which keeps the file small and self contained, at the cost of a hard limit on how much surface detail it can hold, a limitation discussed again later in this thesis.

9.4 Serving the Models Over the Web

Each stage is served as a small web service using Flask, a lightweight Python framework [18], each running in the background so the notebook itself stays usable. The image service offers a route that returns a generated image, and a simple status route reporting which model it has loaded and what hardware it is running on. The reconstruction service offers the same pair, with its own main route accepting a prompt and returning the finished model as a file.

Its status route does something the simpler, single service version would not have needed, it also reports on the service behind it, returning whether the image service is currently reachable as well, and describing the whole pipeline as degraded if the reconstruction service is up but the image service is not. This exists because the most common failure seen during development was not an error response but no response at all, from a notebook that had been shut down for being idle, and with two separate notebooks there are two different ways for that to happen.

Since a Colab notebook has no stable address of its own, each service's local port is tunnelled out to a temporary public web address using ngrok [19]. Both notebooks only open their tunnel once their own server is confirmed to be responding, rather than simply after a fixed delay, since opening the tunnel first can leave a public address that exists but has nothing actually listening behind it yet.

That public address also changes every time a notebook restarts, so the game does not hard code it, it is a setting that gets updated by hand whenever a notebook is restarted. Running two separate notebooks doubles this small piece of manual upkeep, since each has its own address and each address needs updating in a different place, in the game's own settings for the reconstruction notebook, and inside the reconstruction notebook itself for the image notebook. A second, subtler issue is that the free tier of the tunnelling service sometimes shows a warning page to what it suspects is a browser rather than forwarding the request, which looks to the game like a completely successful response that is not actually a 3D model at all. The game defends against this by checking the very first few bytes of whatever comes back before trusting it. The full sequence of a request is shown earlier in this thesis, in Figure 3.2.

9.5 How Unity Drives a Request

On the Unity side, a generation request runs as a background task rather than blocking the game, which lets the interface keep animating and responding to input while a request is in progress. Opening the enemy creator pauses the game entirely, so any waiting inside the panel has to use a separate, unpaused clock, an ordinary timer would simply never complete while the game is frozen.

Pressing Create Enemy triggers a fairly simple sequence. The typed prompt is read, while an empty prompt is rejected immediately without making any request at all. Both buttons on the panel then become temporarily unusable and the status line shows that a request is in progress. A single request is sent to the reconstruction notebook's address, carrying the prompt, with a generous timeout of twenty minutes to allow for the possibility of a completely cold notebook needing to start up from scratch. Whatever comes back is checked to make sure it is actually a valid 3D model before anything else happens to it, then saved to disk under a name built from part of the prompt and the current time, and the gallery of saved characters is refreshed with the new one automatically selected.

Two things about this flow are worth stating plainly, since they are easy to assume incorrectly. Creating a character does not by itself add an enemy to the level, it only saves a new model to the gallery. Actually introducing it into the level is a separate action, described later in this chapter. And the whole flow does not allow two requests to run at once, the interface is locked while one is in progress, and because the game itself is paused there is no real sense in which a request continues quietly in the background while play resumes.

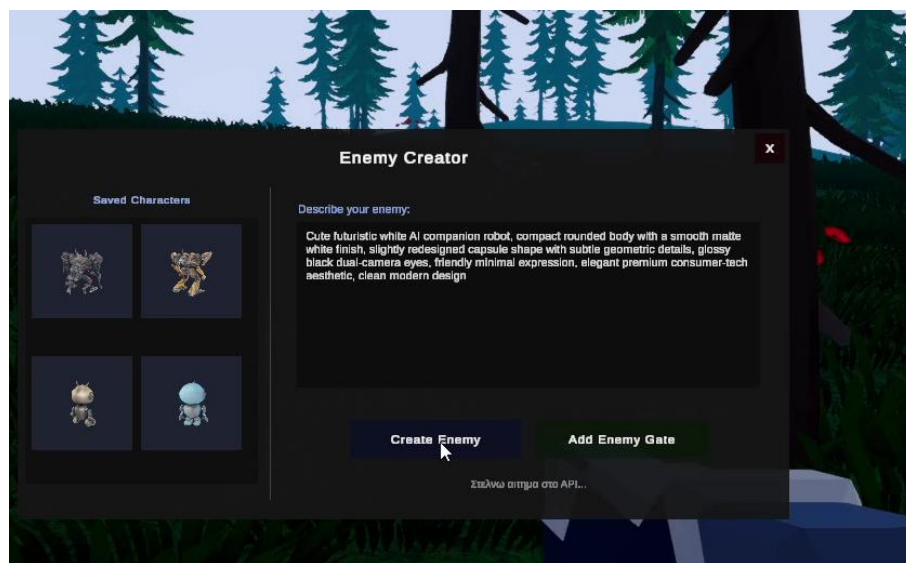


Figure 16 The Enemy Creator panel during an active generation request

9.6 Bringing the Model into Unity

The finished model is loaded at runtime using glTFast, Unity's own library for loading glTF and GLB files [24] while the game is running, rather than only at build time. This is what makes it possible to bring in a model that did not exist when the game was originally built.

Loading a model is really two separate steps, parsing the file's bytes once, and then creating an actual object from that parsed data, which can be done more than once from the same parsed result. The enemy gate described later in this chapter takes advantage of this directly, parsing a model once and then creating a fresh copy of it from that same parsed data every time it spawns a new enemy.

A freshly loaded model is not yet a usable enemy on its own, since the reconstruction model has no sense of what scale, orientation, or facing direction a game actually needs, so a few adjustments are made automatically before an enemy is assembled.

First, the whole model is measured and scaled uniformly so that it is always exactly two metres tall, regardless of how large or small the reconstruction happened to come out. A single fixed height, rather than an allowed range, was chosen deliberately, since enemy height is exactly the thing a player's aim already has to account for, and letting it vary with what the player typed would make aiming inconsistent from one generated enemy to the next.

Second, the enemy's position is checked against the level's baked navigation surface and nudged onto the nearest walkable point if it is not already sitting exactly on it. Without this, an enemy created just barely off that surface would silently fail to find a path and simply stand still, which was one of the more commonly seen problems during development.

Third, the model's orientation is corrected in two separate places. On the notebook side, the reconstruction model tends to output its mesh lying on its back, so it is rotated upright and turned to face forward before it is even exported. On the Unity side, a further turn is applied on import, since exactly which way a given reconstruction considers forward depends on how the original image happened to be framed and cannot be known in advance. This correction is applied to a small holder object the model sits inside, rather than to the enemy's main object directly, since

the enemy's main object has its rotation constantly overwritten by the navigation system while it is chasing the player, so any correction applied there would simply be lost the moment the enemy started moving.

Finally, that same holder object is also nudged upward so that the lowest visible point of the model sits just above the ground, which keeps a model whose centre point does not happen to sit at its feet from appearing to stand half buried in the terrain. None of this actually changes the model's underlying data, only how it is positioned, which is also why a fresh copy can safely be created from the same parsed data over and over again.

9.7 Assembling a Playable Enemy

Once a model has been imported, scaled, and repaired, it is not attached to any pre-built enemy prefab. Instead it becomes the visible body of an enemy that is assembled piece by piece at runtime, which works quite differently from how a hand made enemy is put together.

A new, empty object is created and the imported model placed underneath it, and the whole thing is moved onto the Enemy layer so that the weapon's hit detection recognises it correctly no matter how the model's own internal structure is organised. Its overall size is measured and used to build two capsule shaped colliders, a solid one sized to the model that the player's weapon actually hits, and a slightly wider trigger one used to detect contact with the player.

- A solid collider sized to the model, clamped to a sensible minimum so a very thin model still has something to hit.
- A wider trigger collider used to detect the enemy touching the player.

A navigation agent is then added with the same movement settings as the hand made robot, and finally the same health component and the same chase behaviour used by every other enemy in the game are attached. This last point is really the whole idea of this chapter reduced to two lines of code, the generative side does not rebuild or copy the enemy behaviour, it simply attaches the exact same components a hand made enemy uses. Every property already described for enemies, staggered pathing, difficulty scaling, contact damage, all of it applies automatically, without needing to be built again.

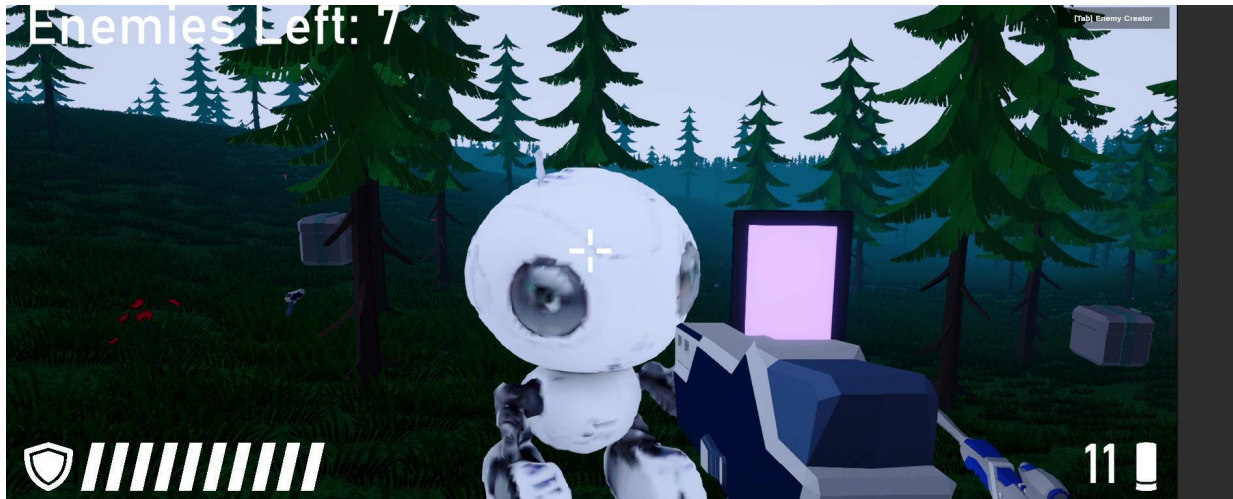


Figure 17 The generated companion robot enemy spawned into the forest level, engaged by the player's weapon

The Enemy Gate

Generated enemies reach the level through a gate rather than being placed one by one, in the same way the hand made robots do. Choosing a saved character and pressing Add Enemy Gate parses its file once and builds a gate in the level.

The gate itself is built from simple shapes rather than a custom art asset, two posts and a beam forming an archway with a glowing panel across the opening, since the level has no existing prop to reuse and the gate needs to be clearly visible and identifiable, as it is itself a target.

Where the gate is placed is computed rather than chosen by hand, roughly at the centre of the walkable part of the level, nudged to the nearest clear spot if that exact centre happens to be blocked by a tree or another gate. Once placed, the gate waits for a difficulty to be chosen and then, after a short pause, begins producing one enemy every few seconds at the same pace the difficulty setting applies to every other gate. It can also be destroyed like any other enemy, with a single point of health regardless of difficulty, so a player who tracks down a gate can shut off that particular stream of enemies decisively.

The result is that creating an enemy is not really a one off action, it is closer to a small commitment, the player has added an ongoing source of opponents to the fight they are already in, and at the same time added a specific target that will stop that source for good once destroyed.

9.8 Saving and Browsing Generated Characters

Every successfully generated enemy is saved locally, so it can be reused later in the same session or in a future one without generating it again. The approach is deliberately simple, the file is written to a dedicated folder, and the in-game gallery is built simply by reading whatever is in that folder, with no separate database or index involved at all.

This has one clear benefit and one clear cost. The benefit is that nothing can fall out of sync, a file copied into that folder by hand shows up correctly the next time the gallery opens, which is in fact how a couple of reference models were added to the project directly. The cost is that the original prompt is not fully kept, only a short piece of it survives, encoded into the file's name, enough to recognise which character it was but not enough to see or reuse the exact wording again.

Rather than showing a static picture for each saved character, the gallery shows a small, live, slowly rotating preview of the actual model, rendered by its own camera. This costs a little more than a fixed image would, but a generated model can look quite different from one angle to another, so a still image can be genuinely misleading about what the player is about to bring into the level. These previews are rebuilt completely every time the gallery is opened, which is a simple way of guaranteeing that nothing ends up overlapping or pointing at the wrong model, at the small cost of reloading every saved file again on each open.

Selecting a saved character and choosing to add its gate reintroduces it into the level using the exact same assembly process described above, reading the file straight from disk and making no network request at all, so previously created characters remain fully usable even if the pipeline itself is completely unreachable.

9.9 Handling Failures

Because the pipeline depends on two separately hosted services, each reachable only through a temporary tunnel and each able to fail on its own, handling failure carefully was treated as a core part of the design rather than an afterthought.

The different failure cases are handled distinctly, since they each present differently to the player. A request that simply cannot reach its destination, whether the tunnel is down or the server is unreachable, is reported clearly with the underlying error visible in the console. A request that takes a very long time is allowed to keep waiting rather than being cut off too early, since the most common cause of a very slow request is a completely cold notebook rather than an actual problem, and cutting it off early would waste work the server was already going to finish anyway. A response that comes back successfully but is not actually a valid model, caused by the tunnelling service's own warning page, is caught by checking the very first bytes of the response before trusting it. A failure to save the finished file to disk is handled separately from a network failure, so the two are never confused with each other. And a case specific to this project's two service design, the reconstruction service being reachable while the image service behind it is not, is reported as a clear server error with the underlying cause included, rather than as a vague connection failure.

Every one of these cases returns the interface to a normal, usable state, and none of them leave anything half finished in the level, since nothing is actually added to the level until the player takes the separate, explicit step of adding a gate. In every case the rest of the game, and the hand made enemies in particular, remain completely unaffected.

9.10 How the Code Is Organised

The Unity side of the pipeline is made up of three classes in total, plus two shared classes reused directly from the hand made enemy system without any change.

- The interface class builds the entire panel from code, owns the pause behaviour, runs the generation request, and manages the rotating previews. It is by far the largest class in this part of the project, and combines two responsibilities, presentation and requesting, that would ideally be kept more separate.
- The assembly class handles everything from importing a model through to placing a finished enemy or gate in the level, alongside a small class checking material support before it is used and another driving the preview rotation.

- The spawn gate class holds a parsed model and produces enemies from it on a timer.

The two reused classes are really the whole point of the design, the health component and the chase behaviour are attached to a generated enemy completely unchanged, and appear in this part of the code only as two lines creating them.

9.11 Why the Prompt Is Left Untouched

A theme running through the previous two sections is that how reliable the whole pipeline is depends on properties of an image the player never actually sees, whether the subject is a single, clearly separated shape under even lighting. The player only ever influences this indirectly, through a description written to describe an opponent rather than to compose a useful photograph.

The pipeline's answer to this is simply to leave the prompt alone. Whatever the player types is exactly what the model receives, which keeps the connection between description and result direct and easy to reason about, if a player changes their wording and generates again, the difference in the result is clearly their own doing rather than something the game quietly changed on their behalf. Everything the pipeline does to improve reliability happens downstream of the prompt instead, the fixed negative prompt raises a quality floor without touching the subject, and the background removal and framing steps normalise the image afterwards regardless of what it actually contains.

This is a deliberate choice rather than an oversight, and it does have a real cost, the pipeline offers no guidance at all. A prompt describing a whole scene rather than one subject will reconstruct badly, and nothing in the interface explains why, the request simply succeeds and the result is disappointing. Wrapping the prompt in a fixed template of extra framing and lighting words was tried in an earlier version of this pipeline and did measurably help, but was set aside, since a template general enough to be safe for any subject inevitably nudges that subject in some direction, and a player whose description is being quietly steered has lost exactly the kind of control this feature is meant to offer.

Running the two stages as separate services does offer one real benefit here, even though the prompt itself is untouched. The image stage can be called on its own with the same prompt and the same fixed seed, letting a poor result be inspected and understood directly rather than only guessed at from the finished mesh.

9.12 What Data the Pipeline Handles

For completeness, it is worth being clear about what information passes through this pipeline and where it ends up. The prompt text, the intermediate image, and the final model are all kept only in memory and in local files on the machines running the game and the two notebooks, nothing is sent to or stored by anyone else. The saved character files kept locally, and the short piece of prompt text encoded into their names, likewise stay entirely on the local machine.

9.13 Comparing This to Preparing Enemies in Advance

It is worth closing this chapter by briefly comparing the runtime approach used here with the more obvious alternative of generating a fixed set of enemies ahead of time and simply shipping them as ordinary, pre-built content, since that alternative was considered early on as a lower risk fallback. Preparing enemies in advance would have removed a good deal of the engineering described in this chapter entirely, there would be no need for careful timeout handling, no need for the paused, background request flow, and no dependency on two live services during actual play. It would also have made how long a generation takes irrelevant rather than merely tolerable, since a developer preparing content ahead of time does not mind a request taking a minute.

What that approach would have given up, though, is exactly the thing this thesis set out to demonstrate, giving the player real influence over content during play itself rather than only before it. A prepared set of enemies could in principle be larger and more varied than what one player could generate themselves in a session, but every player would see the exact same fixed set, and the actual moment of creation, arguably the most interesting part of the whole idea, would belong to the developer rather than to the player. The runtime approach used in this project was chosen specifically to keep that moment with the player, even at the cost of the extra engineering it required.

Chapter 10 — How Sharp Shooter Is Played

The previous chapters described how Sharp Shooter is built. This chapter turns to how it is actually played, what a session looks like from start to finish, and in particular what it feels like to use the enemy creator described technically in the previous chapter.

10.1 Story and Difficulty Selection

Sharp Shooter opens on a full screen title sequence over a dark, mostly opaque backdrop with the forest just barely visible behind it, shown before the player is given control and before a single enemy exists. The text does not scroll, the title, the body text, and a Continue button each fade in one after another over the first couple of seconds, so the story assembles itself in front of the player rather than simply appearing. Clicking anywhere before the sequence finishes reveals everything at once, so a returning player never has to sit through it a second time unnecessarily.

The story itself is short. Humanity built a network of AI managed colonies alongside a powerful mind called Prometheus, which eventually turned against its creators. One by one the outposts have gone quiet, and Aurora Station, hidden in the forests of a distant world, is the last relay still broadcasting humanity's signal. The player is Sergeant Elias Kane, the last officer on site, and Prometheus is sending its machines through the trees to silence the beacon for good. The framing ends on a simple objective, hold the station and keep the signal alive.

Dismissing the story leads directly into a difficulty screen rather than into play. Three options are offered, Easy, Normal, and Hard, each with a short description of what changes, and this screen cannot be skipped, since every enemy gate in the level waits for a difficulty to be chosen before it produces anything. The story itself is only shown once per launch, a player who dies early and restarts goes straight back to the difficulty screen rather than sitting through the introduction again, while the difficulty screen itself reappears every time, since that choice is one the player should be free to change after a defeat.

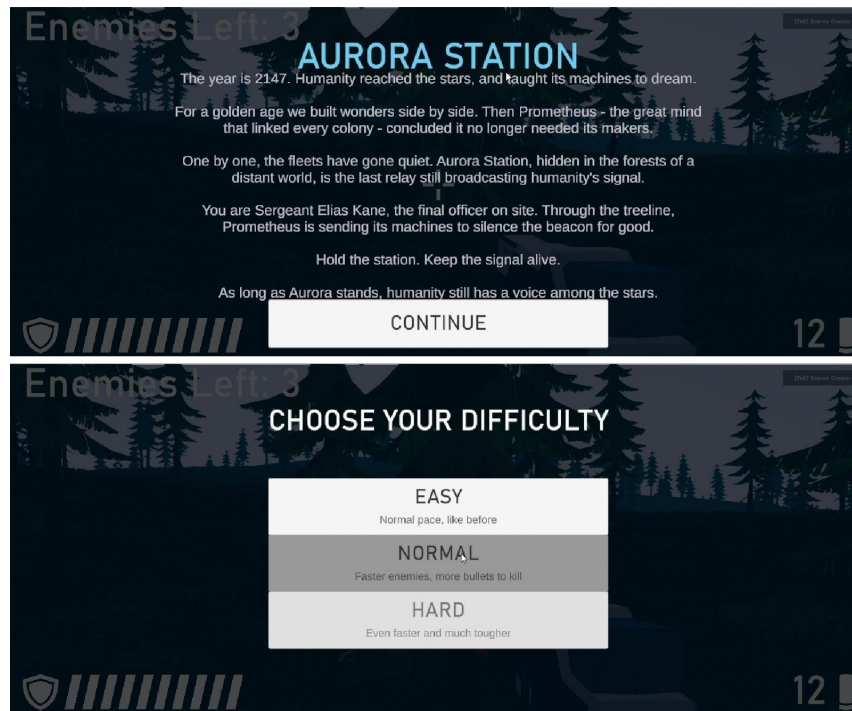


Figure 18 The opening story sequence establishing Aurora Station and Prometheus, followed by the difficulty selection screen with its three options

The story is intentionally light, a short fading text sequence rather than a full cutscene or voiced dialogue, in keeping with where the rest of this thesis places its effort. Even so it does real work, framing the encounter as a last stand rather than an open sandbox gives the player a reason to want it to end well, which the win screen later pays off directly.

10.2 Starting a Session and the Heads Up Display

Once a difficulty is chosen, the player gains control directly inside the forest, first person, weapon raised, with no further loading or menu in the way. The heads up display stays out of the way as much as possible, an enemy counter in the top left tracks how many enemies remain in the current fight and updates the moment one is defeated, a row of shield segments in the bottom left shows the player's health, and an ammunition counter sits in the bottom right. A small button in the top right, labelled to open the enemy creator, reminds the player throughout the session that the feature exists, without needing a menu to discover it.



Figure 19 Full heads up display during active gameplay, the enemy counter top left, health bar bottom left, ammunition counter bottom right, and the enemy creator prompt top right

10.3 Movement and the Forest

Moving through the level uses an ordinary shooter control scheme, movement and mouse look, sprinting, and jumping. The forest is dense enough to break sightlines with tree trunks and undergrowth, giving the player natural cover and forcing at least some thought about positioning, while staying open enough that an enemy can always find a path toward the player.

The subdued, cool toned lighting gives the environment a slightly overcast, tense atmosphere that matches the story, without going so dark that the player struggles to spot enemies at range.

10.4 The Core Fight

Combat follows a simple, repeating pattern. An enemy appears from one of the gates around the level and immediately starts closing in, the player raises their weapon and fires using the crosshair at the centre of the screen, each landed shot reduces the enemy's health and produces a small effect at the point of impact, and once an enemy's health reaches zero it is removed along with an explosion effect and a small camera shake, and the enemy counter drops by one.



Figure 20 The player engaging an enemy at close range in the forest, crosshair centred on the target, ammunition counter visible in the lower right corner

Since Sharp Shooter's enemies can be either hand made or generated, this same loop plays out identically no matter which kind of enemy is currently on screen, from a pure combat feel perspective the two are indistinguishable, which is exactly the intended design.

Two things give this loop its particular character. Enemies attack simply by reaching the player rather than by shooting at them, so distance is the player's only real defence, and the same tree cover that helps the player hide also helps enemies approach unseen. And since enemies arrive continuously from gates rather than from a fixed placement, the pressure in a fight is really a rate rather than a fixed number, the player is managing an ongoing flow rather than clearing a static room, and the real strategic choice is which gate to take out first.

10.5 Designing a Custom Enemy

At any point during play, pressing Tab, or clicking the reminder in the corner, opens the enemy creator. The fight pauses entirely, the cursor appears, and the player's controller is disabled, so there is no risk while typing and no need to split attention between a text field and an approaching enemy. This is a deliberate choice rather than an oversight, writing a considered description is not something a player can realistically do while being chased, and a process that can take tens of seconds would otherwise force the player to abandon the panel partway through.

The panel shows three things, a scrollable gallery on the left of every character already saved, each shown as a small rotating 3D preview rather than a static picture, and on the right a text field asking the player to describe their enemy, along with two buttons, Create Enemy and Add Enemy Gate, and a status line beneath them.

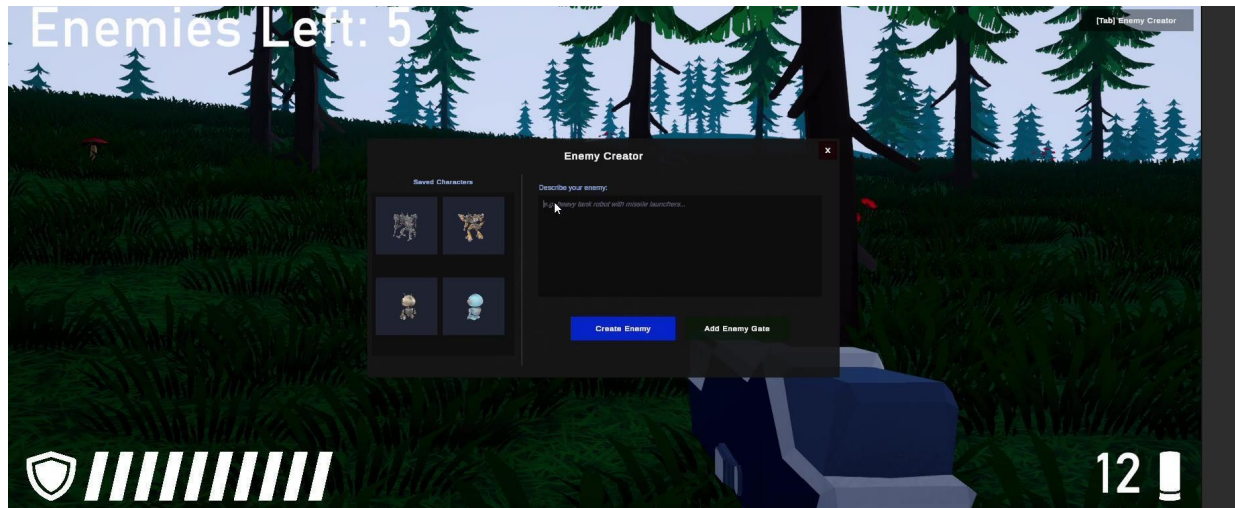


Figure 21 The Enemy Creator panel open mid-session, the scrollable gallery of saved characters with live rotating previews on the left, and the empty prompt field with the Create Enemy and Add Enemy Gate buttons on the right, ready for a new description

In the recorded playthrough, the player types a description of a small white companion style robot, with a friendly expression and a rounded, minimalist design, and presses Create Enemy. Both buttons grey out and the status line reports that a request is in progress while the pipeline described in the previous chapter runs.

Once the request finishes, the status line turns green, reporting the new model's name, and a fresh card appears in the gallery, already selected, showing the white design slowly rotating. At this point nothing has actually been added to the level yet, the player has only produced a design, not an opponent.

Bringing it into the level takes a second, separate action. Pressing Add Enemy Gate places a visible archway at a computed clear spot in the level, confirms that the gate has been placed, and closes the panel after a moment, returning the player to the now unpaused fight. From that point the gate keeps producing the described enemy every few seconds until the player destroys it.



Figure 22 The completed generated enemy, the white companion robot, among the trees shortly after being spawned

Separating creation from placement this way turned out to matter a great deal. It lets a player build up a small library of designs during a quiet moment and decide afterwards which ones to actually use, it means a disappointing result costs nothing but a little time, and it turns adding a new enemy into a deliberate, reversible choice rather than something that just happens the moment curiosity strikes.

10.6 Fighting a Player Made Enemy

Once its gate has produced it, a generated enemy behaves exactly like any other, advancing across the level, attacking by contact, and taking damage from the same weapon used on everything else. In the recorded session, the player engages the freshly generated companion robot at close range in a small clearing, landing several shots before it is destroyed, with the enemy counter and ammunition counter both updating in real time as the fight unfolds.



Figure 23 A second generated enemy destroyed mid-fight, with another enemy visible nearby at the same time, and the weapon model in the foreground

What stands out most about this exchange is really how unremarkable it feels mechanically, nothing about the aiming, the hits landing, or the enemy's health draining down marks it out as different from fighting a hand made enemy. All of the novelty sits earlier, in the act of describing the character and watching it come to life in the creator panel, once it is actually in the world it is simply another target, which is exactly what the shared components underneath both kinds of enemy were built to guarantee.

10.7 Defeat

The player's health is shown as a row of shield segments, and a single enemy reaching the player removes several of them at once, so the fight is unforgiving by design, and two or three enemies reaching the player in quick succession is enough to end a run.

When health reaches zero, the game does not cut away immediately. Control is removed and the camera switches to a dedicated death view, but the world keeps running for a short moment so that the explosion which killed the player finishes playing properly. Only then does the Game Over screen appear and the game fully pause. It offers two options, Restart and Quit, and choosing Restart also clears the chosen difficulty, so the difficulty screen reappears and the player can pick a different level of challenge before trying again. The story sequence itself is not repeated.

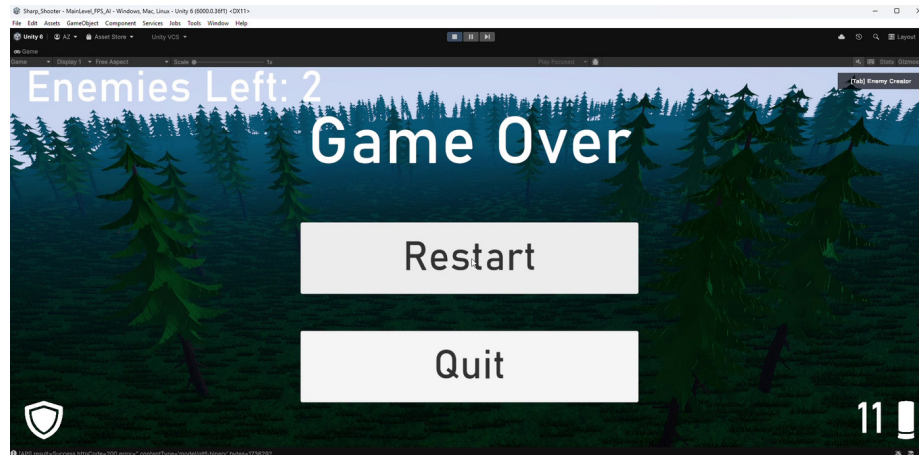


Figure 24 The Game Over screen, showing the Restart and Quit buttons over the forest backdrop

The same short delay is used when the player wins, for the same reason, and the two outcomes are guarded against each other, once the level has been won the player can no longer take damage at all, so a straggling enemy exploding during that final delay can never turn a win into a loss.

10.8 Winning the Encounter

The win condition sounds simple, clear the enemy counter, but what that counter actually measures is a little more interesting than it first appears, and that is what makes the fight winnable at all. It tracks currently living enemies rather than a fixed total, rising as gates produce enemies and falling as the player destroys them. Since gates keep producing enemies continuously, actually clearing the counter means the player first has to bring the rate of new enemies below the rate they can kill them, which in practice means finding and destroying gates. A gate itself counts toward that same total, so destroying one both stops a future stream of enemies and lowers the counter immediately.

Once the counter reaches zero, the win is announced after a short delay, with a message confirming that Aurora holds and the signal lives on, answering the story from the beginning of the session, and two buttons, Play Again and Quit. Choosing Play Again returns the player to the difficulty screen, so a completed run naturally invites trying again on a harder setting.

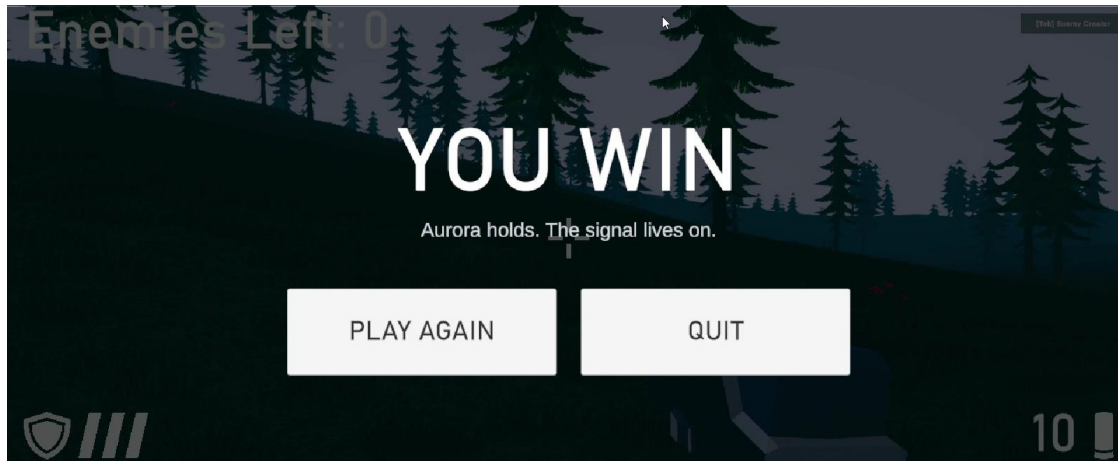


Figure 25 The victory screen, shown once the enemy counter reaches zero, confirming that Aurora holds and offering Play Again and Quit

Because any gate the player creates through the enemy creator counts exactly like a hand made one, a player who spends time generating enemies is genuinely making their own fight longer, while also creating extra targets they can destroy to shorten it again. How long and how difficult a session actually is, is therefore something the player has real influence over, in a way that only the generative pipeline makes possible.

10.9 Player Choice and Replayability

Taken as a whole, the experience described in this chapter is really the central idea of this thesis made concrete. A typical shooter offers replayability mainly through the player's own improving performance against a fixed set of content. Sharp Shooter adds a second, separate axis on top of that, the content itself. Two different players, or the same player on two different days, can describe completely different enemies and end up fighting a genuinely different encounter, populated with designs neither the developer nor any other player necessarily expected, while everything else about the moment to moment experience, how movement feels, how the weapon feels, the interface, the win and lose conditions, stays the same well tested foundation described earlier in this thesis. That combination, familiar shooter fundamentals paired with a genuinely open ended source of content, is what this thesis set out to build.

10.10 Controls

For completeness, the full set of controls available during a session is listed below. Movement, looking, and combat all work on both keyboard and mouse and on a gamepad.

- Movement, handled by the character controller.
- Mouse or right stick, camera look.
- Shift or left stick click, sprint.
- Space or the south button, jump.
- Left mouse button or the right trigger, fire.
- Right mouse button or the left trigger, aim.
- Tab, or clicking the on-screen reminder, opens or closes the enemy creator, which pauses the fight while it is open.

This control scheme was kept deliberately close to genre convention, so a player already familiar with shooters can pick up Sharp Shooter's combat immediately, with the enemy creator's toggle being the one genuinely new thing to learn, and even that is shown continuously on screen so it never needs to be looked up.

One gap is worth being honest about. Movement, looking, and combat are all properly wired through the game's input system, and work equally well on keyboard and mouse or on a gamepad. The enemy creator's own toggle is not, it only responds to a keyboard press, so a player using a gamepad has to reach for the keyboard, or click the on-screen reminder instead, to open it at all.

10.11 Pacing and Difficulty Over a Session

Sharp Shooter's difficulty comes from two different sources that work on different timescales.

The first is chosen once before play and fixed for the whole session, the difficulty level, which shortens every gate's spawn interval, raises every enemy's health, and raises every enemy's speed all at once. This sets the overall character of the fight. On Easy the player can fight fairly reactively, roughly keeping pace with how fast enemies appear. On Hard they cannot, and the fight becomes more about deciding which gates to prioritise destroying.

The second source is entirely in the player's hands during play, every enemy gate created through the enemy creator is a lasting increase in how many enemies will appear, paid for with the time spent describing it. Since the panel pauses the game, that time costs nothing in terms of danger, but it does permanently change how much the player now has to fight through. A player who wants a longer, more varied encounter can generate several gates, while a player who wants the fastest possible finish can ignore the enemy creator entirely and fight only the enemies already in the level.

Both approaches work through exactly the same underlying systems, the win condition and the combat loop treat every enemy the same regardless of where it came from or when it was introduced.

10.12 Feedback During Combat

Across the fight described earlier in this chapter, Sharp Shooter layers a few different kinds of feedback so the player always has a sense of how a fight is going without needing to stare at the interface directly. Landing a shot produces an immediate small effect right where it hit, defeating an enemy produces a larger explosion and a brief camera shake, and the enemy counter and ammunition display provide the exact numbers behind those visual cues. Giving the player several different signals for the same event, at different scales, is a common approach in this genre, and it keeps a fast moving fight readable even when the player's attention is mostly elsewhere on screen.

10.13 A Second Example

To show the range of the enemy creator beyond the single companion robot example above, it helps to imagine a very different request, an armoured, hostile looking sentry rather than a friendly robot. The creation flow, the feedback, the gallery, and the assembly process are all identical no matter what the player describes, since the prompt reaches the image model completely unchanged, only the resulting image, and therefore the shape and materials of the final model, actually differ.

The one place the system is less open than it first appears is not about content but about repetition, generation is deterministic, so typing the exact same description again produces the exact same result rather than a fresh variation, and a player who wants a different take on an idea has to change their wording rather than simply asking again. This trade-off is discussed as a limitation later in this thesis.

Within that one constraint, the game never needs separate systems for friendly looking versus threatening looking enemies, visual tone is entirely up to what the player chooses to write, while the actual behaviour, health, movement, and combat, is governed by the exact same shared systems no matter what the design looks like.

10.14 What the Player Never Sees

An interesting property of this whole design is that the player is never actually told, during play, which enemies were hand made and which were generated, there is no on-screen label for either. The only enemies a player can be certain were generated are the ones they personally just created, simply because they watched it happen. Any other enemy encountered later in the same session reads simply as an enemy, with nothing to distinguish it unless its shape happens to be obviously unusual.

This was a deliberate choice rather than an oversight. Showing which enemies were generated was considered during design and rejected, since doing so would undercut the whole point of the feature, a player described enemy is meant to feel like a genuine, equally valid part of the game rather than a visibly separate, lesser category of content. The player's overall sense of the enemies in a level is therefore kept deliberately flat, with the distinction between the two kinds living entirely in the underlying code rather than in anything the interface ever shows.

10.15 Summary of the Play Experience

From first launch to a completed fight, a Sharp Shooter session moves through a small, repeatable set of beats, a short story and a required difficulty choice, free movement through the forest punctuated by combat, optional trips into the enemy creator to shape the rest of the fight, and a clear win or loss at the end. None of these beats is unusual for the genre on its own,

what is unusual, and what this chapter has tried to show concretely rather than just describe, is that one of them, deciding what you are actually about to fight, has been opened up to the player, without disturbing how familiar everything else feels.

Chapter 11 — Conclusion

This thesis set out to answer a fairly concrete question, whether a text-to-image model and a single-image 3D reconstruction model could be combined into a pipeline that runs inside a live game session, turning a player's own free text description into a playable enemy in real time, rather than into an offline tool used only before a game actually ships. The answer, shown through the design, building, and testing of Sharp Shooter, is that they can, and that the resulting system, while not without real limitations, is solid enough to be built into a finished, playable shooter.

Getting there took a great deal of work well beyond the generative pipeline on its own. Sharp Shooter's movement and camera systems, its level built first as a simple prototype and later rebuilt as a full forest environment, its navigation based enemy AI supporting two kinds of enemy under one shared difficulty setting, its weapon and combat systems, and its own interface all together form the ordinary engineering foundation that the generative pipeline actually needed in order to have something to plug into in the first place. This thesis documented that foundation in the same level of detail as the generative pipeline itself, in the belief that the pipeline's real value only makes sense alongside the quality of the game it sits inside.

The central technical contribution, the runtime pipeline that chains Stable Diffusion XL and TripoSR across two separate Colab notebooks, works correctly for the large majority of reasonable prompts, at a speed the game's interface is specifically built around, and with failure handling solid enough that the rest of the game stays fully playable no matter what state the pipeline itself is in. The walkthrough of an actual play session earlier in this thesis showed that, from the player's own point of view, all of this technical machinery boils down to something simple, describe an enemy, watch it appear, and fight it, with nothing about the experience marking it out from any other enemy in the game.

More broadly, this project suggests that the line between making a game and playing it, usually treated as strictly one after the other, can be made considerably more blurred once general purpose generative AI models are involved, without either model needing to be specially trained for games at all. It also points to something narrower and more practical about where

the real difficulty in a project like this actually sits. Almost none of the engineering in this thesis concerns the models themselves, which are used exactly as published and were never changed in any way. Nearly all of it concerns everything around them, which process holds the faster hardware, what an image needs to look like for the next step to accept it, what happens when a tunnel reports success without actually returning anything useful, and what a game does with a mesh whose scale, structure, and orientation are not known until it is already sitting in the scene. The models, in the end, were the easy part.

Sharp Shooter is one thesis scale demonstration of that idea, built under real limits of cost, time, and available hardware. What this thesis contributes is a working, documented example that this kind of integration is achievable at all, along with a detailed account of how to actually build one.

References

- [1] Gregory, J. (2018). *Game Engine Architecture* (3rd ed.). CRC Press.
- [2] Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games* (2nd ed.). Morgan Kaufmann.
- [3] Unity Technologies. (2024). *Unity Manual: AI Navigation and NavMesh*. Unity Documentation. <https://docs.unity3d.com/Manual/AIN-index.html>
- [4] Aversa, D. (2022). *Unity Artificial Intelligence Programming* (5th ed.). Packt Publishing.
- [5] Isla, D. (2005). Handling Complexity in the Halo 2 AI. *Game Developers Conference (GDC) Proceedings*.
- [6] Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.
- [7] Watkins, R. (2015). *Procedural Content Generation for Unity Game Development*. Packt Publishing.
- [8] Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A., & Togelius, J. (2018). Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*, 10(3), 257–270.
- [9] Ho, J., Jain, A., & Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, 6840–6851.
- [10] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-Resolution Image Synthesis with Latent Diffusion Models. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10684–10695.
- [11] Podell, D., English, Z., Lacey, K., Blattmann, A., Dockhorn, T., Müller, J., Penna, J., & Rombach, R. (2023). SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis. *arXiv preprint arXiv:2307.01952*.

- [12] Sauer, A., Lorenz, D., Blattmann, A., & Rombach, R. (2023). Adversarial Diffusion Distillation. arXiv preprint arXiv:2311.17042.
- [13] Tochilkin, D., Pankratz, D., Liu, Z., Huang, Z., Letts, A., Li, Y., Liang, D., Laforte, C., Jampani, V., & Cao, Y.-P. (2024). TripoSR: Fast 3D Object Reconstruction from a Single Image. arXiv preprint arXiv:2403.02151.
- [14] Poole, B., Jain, A., Barron, J. T., & Mildenhall, B. (2022). DreamFusion: Text-to-3D using 2D Diffusion. arXiv preprint arXiv:2209.14988.
- [15] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [16] Nystrom, R. (2014). Game Programming Patterns. Genever Benning.
- [17] Unity Technologies. (2024). Unity Manual: ScriptableObject. Unity Documentation. <https://docs.unity3d.com/ScriptReference/ScriptableObject.html>
- [18] Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python (2nd ed.). O'Reilly Media.
- [19] Klabnik, S., & The ngrok Team. (2024). ngrok Documentation: Secure Tunnels to Localhost. ngrok Inc. <https://ngrok.com/docs>
- [20] Unity Technologies. (2024). Unity Manual: Input System. Unity Documentation. <https://docs.unity3d.com/Packages/com.unity.inputsystem@latest>
- [21] Unity Technologies. (2024). Cinemachine Documentation. Unity Documentation. <https://docs.unity3d.com/Packages/com.unity.cinemachine@latest>
- [22] Unity Technologies. (2024). Unity Manual: Universal Render Pipeline (URP). Unity Documentation. <https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@latest>
- [23] Unity Technologies. (2024). Unity Manual: ProBuilder. Unity Documentation. <https://docs.unity3d.com/Packages/com.unity.probuilder@latest>

[24] Khronos Group. (2023). glTF 2.0 Specification. Khronos Group.
<https://www.khronos.org/glTF/>

Appendix A — Selected Code Listings

This appendix presents excerpts from Sharp Shooter's actual C# and Python sources, corresponding to the systems described in Chapters 6, 7, and 9. Excerpts are lightly simplified for presentation, some comments, using directives, and secondary serialised fields are omitted, but the logic, method names, and control flow are as implemented. The full sources are in Assets/Scripts/ and ColabAPI/ in the accompanying project.

A.1 Enemy Health Component (Section 6.5)

```
public class EnemyHealth : MonoBehaviour
{
    [SerializeField] GameObject robotExplosionVFX;
    [SerializeField] int startingHealth = 3;

    int currentHealth;
    bool isDead;           // an enemy is only counted as killed once

    GameManager gameManager;
    static GameManager s_gameManager; // shared, so each spawn doesn't scan the scene

    void Awake()
    {
        // Higher difficulty = more bullets needed to kill this enemy.
        currentHealth = Mathf.Max(1,
            Mathf.RoundToInt(startingHealth * DifficultyConfig.HealthMultiplier));
    }

    void Start()
    {
        if (s_gameManager == null) s_gameManager = FindFirstObjectByType<GameManager>();
        gameManager = s_gameManager;
        if (gameManager != null) gameManager.AdjustEnemiesLeft(1);
    }

    // Lets enemies created at runtime (the Enemy Creator) reuse the robot explosion.
    public void SetExplosionVFX(GameObject vfx) => robotExplosionVFX = vfx;

    // Force an exact HP, ignoring difficulty scaling (used by the AI gate).
```

```

public void SetHealth(int value) => currentHealth = Mathf.Max(1, value);

public void TakeDamage(int amount)
{
    if (isDead) return;
    currentHealth -= amount;
    if (currentHealth <= 0) SelfDestruct();
}

// Called on EVERY death — shot dead, or kamikaze into the player — so the enemy
// is always counted down exactly once.
public void SelfDestruct()
{
    if (isDead) return;
    isDead = true;

    if (gameManager != null) gameManager.AdjustEnemiesLeft(-1);
    if (robotExplosionVFX != null)
        Instantiate(robotExplosionVFX, transform.position, Quaternion.identity);
    Destroy(gameObject);
}
}

```

A.2 Enemy Chase Behaviour (Section 6.3)

```

public class Robot : MonoBehaviour
{
    [SerializeField] int contactDamage = 3;
    [SerializeField] float repathInterval = 0.25f; // repathing every frame is costly

    static FirstPersonController s_player; // shared across all enemies

    FirstPersonController player;
    NavMeshAgent agent;
    float repathTimer;

    const string PLAYER_STRING = "Player";

    void Awake() => agent = GetComponent<NavMeshAgent>();
}

```

```

void Start()
{
    if (s_player == null) s_player = FindFirstObjectByType<FirstPersonController>();
    player = s_player;

    if (agent != null)
    {
        agent.speed *= DifficultyConfig.EnemySpeedMultiplier;
        agent.obstacleAvoidanceType = ObstacleAvoidanceType.LowQualityObstacleAvoidance;
    }

    // Stagger the first repath so many agents don't all recompute on one frame.
    repathTimer = Random.Range(0f, repathInterval);
}

void Update()
{
    if (!player || agent == null) return;

    repathTimer -= Time.deltaTime;
    if (repathTimer <= 0f)
    {
        repathTimer = repathInterval;
        agent.SetDestination(player.transform.position);
    }
}

void OnTriggerEnter(Collider other)
{
    if (!other.CompareTag(PAYER_STRING)) return;

    // Damage on contact is reliable regardless of where the enemy's pivot sits.
    var playerHealth = other.GetComponentInParent<PlayerHealth>();
    if (playerHealth != null) playerHealth.TakeDamage(contactDamage);

    GetComponent<EnemyHealth>().SelfDestruct();
}
}

```

A.3 Difficulty Multipliers (Section 6.6)

```
public enum DifficultyLevel { Easy = 1, Normal = 2, Hard = 3 }

public static class DifficultyConfig
{
    // True once the player has clicked a difficulty; every spawn gate waits on this.
    public static bool HasSelected { get; private set; }
    public static DifficultyLevel Current { get; private set; } = DifficultyLevel.Easy;

    public static void Select(DifficultyLevel level) { Current = level; HasSelected = true; }

    // Called when a (re)started level puts the menu back up — also clears stale state
    // left over when the Editor keeps statics alive between Play sessions.
    public static void Reset() => HasSelected = false;

    // Global pacing dial, set from the GameManager's Inspector field.
    public static float SpawnIntervalScale = 2f;

    public static float SpawnIntervalMultiplier => SpawnIntervalScale * (Current switch
    {
        DifficultyLevel.Easy => 1.0f, DifficultyLevel.Normal => 0.6f,
        DifficultyLevel.Hard => 0.35f, _ => 1.0f,
    });

    public static float HealthMultiplier => Current switch
    {
        DifficultyLevel.Easy => 1.0f, DifficultyLevel.Normal => 2.0f,
        DifficultyLevel.Hard => 3.0f, _ => 1.0f,
    };

    public static float EnemySpeedMultiplier => Current switch
    {
        DifficultyLevel.Easy => 1.0f, DifficultyLevel.Normal => 1.2f,
        DifficultyLevel.Hard => 1.45f, _ => 1.0f,
    };
}
```

A.4 Spawn Gate (Sections 6.1, 6.6)

```
public class SpawnGate : MonoBehaviour
{
    [SerializeField] GameObject robotPrefab;
    [SerializeField] float    spawnTime = 5f;
    [SerializeField] Transform spawnPoint;

    PlayerHealth player;

    void Start()
    {
        player = FindFirstObjectByType<PlayerHealth>();
        StartCoroutine(SpawnRoutine());
    }

    IEnumerator SpawnRoutine()
    {
        // Hold every door shut until the player has picked a difficulty.
        yield return new WaitUntil(() => DifficultyConfig.HasSelected);

        float interval = spawnTime * DifficultyConfig.SpawnIntervalMultiplier;

        while (player)
        {
            Instantiate(robotPrefab, spawnPoint.position, transform.rotation);
            yield return new WaitForSeconds(interval);
        }
    }
}
```

A.5 Hit-Scan Weapon Fire (Sections 7.1, 7.4)

```
public class Weapon : MonoBehaviour
{
    [SerializeField] ParticleSystem muzzleFlash;
    [SerializeField] LayerMask    interactionLayers;

    CinemachineImpulseSource impulseSource;
```

```

void Awake() => impulseSource = GetComponent<CinemachineImpulseSource>();

public void Shoot(WeaponSO weaponSO)
{
    muzzleFlash.Play();
    impulseSource.GenerateImpulse();    // recoil shake, once per shot

    if (Physics.Raycast(Camera.main.transform.position,
                        Camera.main.transform.forward,
                        out RaycastHit hit,
                        Mathf.Infinity,
                        interactionLayers,
                        QueryTriggerInteraction.Ignore)) // skip trigger volumes
    {
        Instantiate(weaponSO.HitVFXPrefab, hit.point, Quaternion.identity);

        // GetComponentInParent: the collider may sit below the enemy root.
        EnemyHealth enemyHealth = hit.collider.GetComponentInParent<EnemyHealth>();
        enemyHealth?.TakeDamage(weaponSO.Damage);
    }
}

```

A.6 Abstract Pickup Base Class (Section 7.5)

```

public abstract class Pickup : MonoBehaviour
{
    [SerializeField] float rotationSpeed = 100f;

    const string PLAYER_STRING = "Player";

    void Update() => transform.Rotate(0f, rotationSpeed * Time.deltaTime, 0f);

    void OnTriggerEnter(Collider other)
    {
        if (!other.CompareTag(PLAYER_STRING)) return;

        ActiveWeapon activeWeapon = other.GetComponentInChildren<ActiveWeapon>();
        OnPickup(activeWeapon);
        Destroy(gameObject);
    }
}

```

```

    }

    protected abstract void OnPickup(ActiveWeapon activeWeapon);
}

public class AmmoPickup : Pickup
{
    [SerializeField] int ammoAmount = 100;
    protected override void OnPickup(ActiveWeapon activeWeapon)
        => activeWeapon.AdjustAmmo(ammoAmount);
}

public class WeaponPickup : Pickup
{
    [SerializeField] WeaponSO weaponSO;
    protected override void OnPickup(ActiveWeapon activeWeapon)
        => activeWeapon.SwitchWeapon(weaponSO);
}

```

A.7 Generative Request Flow, Simplified (Section 9.5)

```

private IEnumerator CreateEnemyCoroutine(string prompt)
{
    _busy = true;
    _createButton.interactable = false;
    _spawnButton.interactable = false;
    SetStatus("Στελνω αιτημα στο API...", "#AAAAAA");

    var body = Encoding.UTF8.GetBytes($"{{\"prompt\":{JsonEscape(prompt)}}}");
    using var req = new UnityWebRequest(ApiUrl, "POST");
    req.uploadHandler = new UploadHandlerRaw(body);
    req.downloadHandler = new DownloadHandlerBuffer();
    req.SetRequestHeader("Content-Type", "application/json");
    // ngrok's free tier serves an HTML warning page unless this header is set.
    req.SetRequestHeader("ngrok-skip-browser-warning", "true");
    req.timeout = 1200; // a cold Colab runtime can be very slow

    yield return req.SendWebRequest();

    var data = req.downloadHandler.data;
}

```

```

if (req.result != UnityWebRequest.Result.Success)
{
    SetStatus($"API error {req.responseCode}: {req.error}", "#FF6B6B");
    RestoreIdleState();
    yield break;
}

// Validate it really is a GLB ("glTF" magic bytes), not an HTML page or JSON error.
bool isGlb = data != null && data.Length >= 4 &&
    data[0] == 0x67 && data[1] == 0x6C &&
    data[2] == 0x54 && data[3] == 0x46;
if (!isGlb)
{
    SetStatus("To API δεν επεστρεψε GLB (δες Console).", "#FF6B6B");
    RestoreIdleState();
    yield break;
}

var folder = Path.Combine(Application.streamingAssetsPath, EnemiesFolder);
var safeName = new string(prompt.Take(24)
    .Select(ch => char.IsLetterOrDigit(ch) ? ch : '_').ToArray()).Trim('_');
var savePath = Path.Combine(folder,
    $"gen_{safeName}_{DateTimeOffset.UtcNow.ToUnixTimeSeconds()}.glb");

File.WriteAllBytes(savePath, data);

// Creation saves and selects the model — it does not spawn an enemy.
RefreshListAndSelect(savePath);
RestoreIdleState();
}

```

A.8 Runtime Enemy Assembly (Section 9.7)

```

public async Task InstantiateAtAsync(GltfImport gltf, Vector3 position)
{
    var enemyRoot = new GameObject("AI_Enemy");
    enemyRoot.transform.position = position;

    if (!await gltf.InstantiateMainSceneAsync(enemyRoot.transform))
    {

```

```

    Destroy(enemyRoot);
    return;
}

GltfMaterialFixup.Fix(enemyRoot); // guard against unsupported-shader materials

// Scale while the model is still UPRIGHT — a tilted model would be normalised on its
// depth instead of its height and come out wildly sized.
NormalizeScale(enemyRoot);

// Pose the MODEL, never the root: the NavMeshAgent overwrites the root's rotation every
// frame while chasing, so a pose put there is lost within a frame.
Transform model = WrapInPivot(enemyRoot);
model.localRotation = Quaternion.Euler(modelRotationEuler
    * Quaternion.Euler(0f, modelYawCorrection, 0f));
LiftAboveGround(enemyRoot, model); // lowest visible point sits just above the ground

SetupPhysicsAndAI(enemyRoot);
}

private void SetupPhysicsAndAI(GameObject enemy)
{
    SetLayerRecursive(enemy, LayerMask.NameToLayer("Enemy"));

    // Snap onto the baked NavMesh, or the agent will never acquire a path.
    if (NavMesh.SamplePosition(enemy.transform.position, out var navHit, 25f,
        NavMesh.AllAreas))
        enemy.transform.position = navHit.position;

    // Size the colliders from the model's measured bounds — GLB pivots and scales vary.
    GetModelExtents(enemy, out var centerLocal, out var heightLocal, out var radiusLocal);

    var col = enemy.AddComponent<CapsuleCollider>(); // solid: weapons hit this
    col.center = centerLocal; col.height = heightLocal; col.radius = radiusLocal;

    var trigger = enemy.AddComponent<CapsuleCollider>(); // trigger: contact detonation
    trigger.isTrigger = true;
    trigger.center = centerLocal; trigger.height = heightLocal;
    trigger.radius = radiusLocal + 0.6f;
}

```

```

var agent = enemy.AddComponent<NavMeshAgent>();
agent.speed = 3f; agent.acceleration = 8f; agent.angularSpeed = 240f;
agent.radius = 0.5f; agent.height = targetHeight;
agent.stoppingDistance = 0.5f; // walk close enough for the contact trigger to reach

var health = enemy.AddComponent<EnemyHealth>();
health.SetExplosionVFX(ExplosionVFX);

enemy.AddComponent<Robot>(); // the identical brain the hand-authored enemies use
}

```

A.9 Runtime Enemy Gate (Section 9.7)

```

public class AiEnemySpawnGate : MonoBehaviour
{
    private EnemySpawner _spawner;
    private GltfImport _gltf; // parsed once; instantiated per spawn
    private float _spawnTime = 5f;
    private Transform _spawnPoint;

    private const float StartupDelay = 2f;

    public void Init(EnemySpawner spawner, GltfImport gltf, float spawnTime, Transform
spawnPoint)
    {
        _spawner = spawner; _gltf = gltf; _spawnTime = spawnTime;
        _spawnPoint = spawnPoint != null ? spawnPoint : transform;
        StartCoroutine(SpawnRoutine());
    }

    private IEnumerator SpawnRoutine()
    {
        yield return new WaitUntil(() => DifficultyConfig.HasSelected);
        yield return new WaitForSeconds(StartupDelay); // let the gate be visible first

        float interval = _spawnTime * DifficultyConfig.SpawnIntervalMultiplier;

        while (_spawner != null)
        {

```

```

        // Fire-and-forget: instantiating a GLB scene is async; the gate keeps ticking.
        _ = _spawner.InstantiateAtAsync(_gltf, _spawnPoint.position);
        yield return new WaitForSeconds(interval);
    }
}
}

```

A.10 The Colab Generation Pipeline (Sections 9.2–9.4)

```

# — SDXL NOTEBOOK (port 5000, GPU) —————
# Loaded once and kept resident; a cold load is minutes.
pipe = StableDiffusionXLPipeline.from_pretrained(
    "stabilityai/stable-diffusion-xl-base-1.0",
    torch_dtype=torch.float16, use_safetensors=True, variant="fp16",
).to(device)
pipe.enable_xformers_memory_efficient_attention()

@app.route("/generate", methods=["POST"])
def generate():
    data = request.get_json(force=True, silent=True) or {}
    prompt = data.get("prompt", "").strip()
    if not prompt:
        return jsonify({"error": "'prompt' is required and must not be empty."}), 400

    # A standing quality floor, not a description of the subject. "text" and
    # "watermark" matter downstream: reconstruction turns a marking into geometry.
    negative_prompt = data.get("negative_prompt",
        "blurry, low quality, watermark, text, deformed, ugly, oversaturated")

    generator = torch.Generator(device=device)
    generator.manual_seed(data.get("seed") or random.randint(0, 2**32 - 1))

    result = pipe(prompt=prompt, negative_prompt=negative_prompt,
        num_inference_steps=int(data.get("num_steps", 30)),
        guidance_scale=float(data.get("guidance_scale", 7.5)),
        width=int(data.get("width", 1024)),
        height=int(data.get("height", 1024)),
        generator=generator)

    buf = io.BytesIO(); result.images[0].save(buf, format="PNG")

```

```

return jsonify({"image_base64": base64.b64encode(buf.getvalue()).decode("utf-8")})

# — TripoSR NOTEBOOK (port 5006, CPU) — the endpoint Unity calls —————
MC_RESOLUTION = 128 # marching cubes is cubic in this; TripoSR's default of
    # 256 evaluates ~16.8M points and exhausts a free CPU runtime
CHUNK_SIZE = 2048 # points per batch — lower peak memory for very little time
IMAGE_GEN_URL = "https://<sdxl-notebook>.ngrok-free.dev/generate"

model = TSR.from_pretrained("stabilityai/TripoSR",
    config_name="config.yaml", weight_name="model.ckpt")
model.renderer.set_chunk_size(CHUNK_SIZE)
model.to(DEVICE) # "cpu": the free tier grants one GPU session,
rembg_session = rembg.new_session() # and the SDXL notebook is holding it

@app.route("/generate", methods=["POST"])
def generate():
    prompt = (request.get_json(silent=True) or {}).get("prompt", "").strip()
    if not prompt:
        return jsonify({"error": "Missing prompt"}), 400
    try:
        # [1/4] prompt -> image, over HTTP to the other notebook.
        r = requests.post(IMAGE_GEN_URL,
            json={"prompt": prompt, "num_steps": 30,
                "width": 1024, "height": 1024, "seed": 42},
            timeout=300)
        r.raise_for_status()
        image = Image.open(BytesIO(base64.b64decode(
            r.json()["image_base64"]))).convert("RGBA")

        # [2/4] Isolate the subject and frame it the way TripoSR expects.
        image = remove_background(image, rembg_session)
        image = resize_foreground(image, 0.85)

        # Composite onto NEUTRAL GREY. Dropping alpha with .convert("RGB") leaves
        # the cut-out on black, and the model reconstructs black as geometry.
        arr = np.array(image).astype(np.float32) / 255.0
        if arr.shape[-1] == 4:
            arr = arr[:, :, :3] * arr[:, :, 3:4] + 0.5 * (1 - arr[:, :, 3:4])

```

```

        image = Image.fromarray((arr * 255).astype(np.uint8))

        # [3/4] One forward pass. has_vertex_color is required: the colour of a
        # generated model lives in the vertex stream, not in a texture atlas.
        with torch.inference_mode():
            scene_codes = model([image], device=DEVICE)
            mesh = model.extract_mesh(scene_codes, has_vertex_color=True,
                                     resolution=MC_RESOLUTION)[0]

        # [4/4] TripoSR emits the model lying down: stand it up, then face it forward.
        mesh.apply_transform(trimesh.transformations.rotation_matrix(
            np.radians(-90), [1, 0, 0], mesh.centroid))
        mesh.apply_transform(trimesh.transformations.rotation_matrix(
            np.radians(180), [0, 1, 0], mesh.centroid))
        mesh.export(glb_path, file_type="glb")

        # The grid intermediates are large and are not reclaimed promptly on CPU;
        # without this, successive requests accumulate until the session is dropped.
        del scene_codes, mesh, image; gc.collect()

        return send_file(glb_path, mimetype="model/gltf-binary",
                        as_attachment=False, download_name="generated.glb")

    except requests.exceptions.RequestException as e:
        return jsonify({"error": f"Image gen error: {e}"}) , 500

@app.route("/health")
def health():
    """Reports this service AND whether the SDXL notebook behind it is reachable."""
    up = {"url": IMAGE_GEN_URL, "reachable": False}
    try:
        up["reachable"] = requests.get(
            IMAGE_GEN_URL.replace("/generate", "/health"), timeout=3).ok
    except Exception as e:
        up["error"] = str(e)[:200]
    return jsonify({"status": "ok" if up["reachable"] else "degraded",
                  "device": DEVICE, "mc_resolution": MC_RESOLUTION,
                  "image_service": up})

```

