Constraint Is All You Need: Optimization-Based 3D Level Generation with LLMs

Kaijie Xu
Mcgill University
Montréal, Quebec, Canada
kaijie.xu2@mail.mcgill.ca

Clark Verbrugge
Mcgill University
Montréal, Quebec, Canada
clump@cs.mcgill.ca

Abstract

Procedural Content Generation (PCG) has long enabled efficient and varied game level creation. However, integrating high-level design intentions and game mechanics into complex 3D environments remains challenging. This paper introduces a comprehensive framework that transforms narrative-level descriptions into playable 3D game levels. First, Large Language Models (LLMs) parse natural language descriptions of game environments into a structured Game Level Description Language (GLDL), capturing essential spatial constraints. Next, we model level generation as a Facility Layout Optimization problem, ensuring that facility placements and configurations adhere to specified design criteria. Through comprehensive experiments, including automated constraint evaluations and agent-based simulations, our approach ensures both the feasibility and stability of the constraints extracted from textual descriptions. We confirm that the resulting game levels remain interactive, reasonable, and controllable to their original specifications.

CCS Concepts

• **Applied computing → Computer games**.

Keywords

Procedural Content Generation, Facility Layout Problem, Level Generation, Large Language Models, Game Design

ACM Reference Format:
Kaijie Xu and Clark Verbrugge. 2025. Constraint Is All You Need: Optimization-Based 3D Level Generation with LLMs. In *International Conference on the Foundations of Digital Games (FDG '25), April 15–18, 2025, Graz, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3723498.3723840

1 Introduction

Procedural Content Generation (PCG) remains a continuously researched and evolving topic in both academic and industrial settings, enabling the automated creation of diverse, replayable, and contextually rich game environments, particularly in roguelikes and open-world games. However, complex game worlds require PCG systems that integrate high-level design intentions to ensure narrative coherence and effective use of gameplay mechanics. In narrative-driven level design, the spatial arrangement of objects and characters must align with the overall story and gameplay objectives. By incorporating these elements, PCG tools empower designers to retain creative control while achieving environments that are both functionally robust and visually cohesive.

Recent advancements in integrating Large Language Models (LLMs) and Computer Vision (CV) techniques have confirmed substantial progress in generating visually appealing scenes [1, 5, 6, 10, 17–19, 27–29]. However, when applied to game level generation, these methods do not always sufficiently account for critical aspects such as game logic, mechanics, and interactive elements. Consequently, although they are effective for pure scene generation, they may produce environments that are visually engaging but lack the functional coherence. We therefore seek a more universally applicable framework that supports custom asset integration while prioritizing structural coherence and functional playability.

In this work, we propose a novel framework that leverages the capabilities of LLMs to bridge the gap between narrative descriptions and structured level design. This approach introduces a specialized *Game Level Description Language* (GLDL) that encodes essential spatial constraints, relationships, and gameplay mechanics derived from natural language inputs. By formulating level generation as a Facility Layout Optimization problem—a classic challenge focused on arranging facilities within a given space to optimize spatial efficiency and meet design constraints—the framework translates narrative-driven specifications into precise and feasible 3D layouts. This methodology ensures that the generated environments are not only visually coherent but also adhere to the desired game logic and interactive dynamics.

Key contributions of our work include:

- We describe a PCG pipeline for generating non-trivial game levels from textual input, incorporating basic game mechanisms such as lock/key or other distributed puzzles, combat, and spatial arrangement.
- Our pipeline relies on a novel *Game Level Description Language*, a specialized language inspired by *Scenic* [7], designed to encode spatial constraints, relationships, and gameplay mechanics as derived from natural language descriptions. This facilitates the translation of narrative intents into structured level specifications.
- Translation from the GLDL into the actual level structure is modeled as a basic facility layout optimization problem. We explore various optimization methods, enabling the automated placement of game facilities that adhere to the narrative and gameplay constraints.
- Our work is validated through a Unity3D simulation, using a synthetic agent-based approach to ensure the playability and coherence of the generated levels.

FDG '25, Graz, Austria
© 2025 Copyright held by the owner/author(s).
ACM ISBN /25/04
https://doi.org/10.1145/3723498.3723840

2 Related Work

PCG is a large and long-standing research area in game research. Below, we summarize prior work in relation to level generation, the facility layout problem, and the use of LLMs in PCG.

2.1 Procedural Content Generation (PCG) and Level Generation

Procedural Content Generation (PCG) has been extensively explored in game design, emphasizing the automated creation of game elements such as levels, maps, and scenarios. Traditional PCG approaches include random generation, rule-based systems, and search-based methods, each presenting unique advantages and limitations [20, 24]. In recent years, there has been a significant shift towards integrating machine learning and reinforcement learning techniques to enhance the adaptability, complexity, and creativity of generated content [12, 13, 22].

While substantial progress has been made in 2D, grid-based level generation, the exploration of PCG in 3D environments remains relatively limited. Several studies have employed reinforcement learning techniques to construct 3D environments [11]. Additionally, computer vision methods, which have demonstrated substantial efficacy in their native domains, have been adapted for use in the generation of 3D scenes [5, 6, 17–19, 28]. Our research aims to address this gap by developing a framework that leverages large language models and optimization algorithms to create playable, diverse, and coherent 3D game levels.

2.2 Facility Layout Problem

The Facility Layout Problem (FLP) is a well-established optimization challenge in operations research, focused on arranging facilities within a given space to optimize specific criteria, such as minimizing movement costs or maximizing space utilization [2, 21]. Traditionally applied in manufacturing and service industries, FLP seeks to enhance operational efficiency and spatial organization by determining the most effective placement of various facilities or departments. FLP was introduced into the game field in 2006, primarily focusing on generating building interiors and architectural room layouts [8, 14, 15].

In recent years, FLP has extended its applications to domains beyond traditional industries, including scene generation tasks such as traffic scene generation and 2D game world generation [4, 25, 26]. By adapting FLP methodologies to these new areas, researchers can achieve systematic and optimized placement of elements within complex environments. Our research leverages FLP to address the challenges inherent in 3D game level generation. By formulating level design as an FLP, we can simulate the real constraints as abstract numerical constraints and employ advanced optimization algorithms to determine optimal facility placements that adhere to both gameplay mechanics and visual design principles.

2.3 LLMs in PCG

Large Language Models (LLMs) are pre-trained with billions of parameters to perform a wide array of tasks, including natural language understanding and generation [30]. Recent advancements, such as GPTs [16] and Claude [3], have demonstrated remarkable text generation capabilities, underscoring the potential of artificial general intelligence. In the realm of Procedural Content Generation (PCG), LLMs facilitate the creation of game content by interpreting text-based prompts to generate rules and levels [9]. LLMs offer an intuitive interface through natural language prompts, enabling more straightforward human involvement and feedback compared to traditional methods like player trajectories. Latest studies [1, 10, 29] have further expanded the capabilities of LLMs in generating complex 3D scenes and immersive indoor environments. Despite recent efforts to fine-tune LLMs for level generation within fixed game rules and datasets [23], these approaches often require additional training time and are susceptible to randomness and hallucinations inherent to LLMs. In our research, we utilize LLMs to transform game-level description texts into a specialized Game Level Description Language (GLDL), thereby minimizing the instability associated with direct LLM outputs and enhancing the reliability of generated game levels.

3 Methodology

Our procedural content generation framework is designed to efficiently convert natural language descriptions into playable game levels through an integrated workflow. As illustrated in Figure 1, the process begins with the input of high-level, human-written narrative descriptions, which are then processed by an LLM to extract keywords and constraints. These elements are converted into a structured Game Level Description Language (GLDL), guiding the layout optimization algorithms to generate detailed data on facility placements within the room. This data is then used to simulate the final game level. To ensure the quality and stability of the generated levels, the framework includes an "LLM Stability Test," which evaluates the consistency and coherence of the LLM-generated content, as well as an "Agent Simulation Test," which assesses the playability and user experience by simulating agent behaviors within the generated environment. This systematic approach ensures that the generated levels are not only contextually accurate but also playable.

Below we first introduce the GLDL, and then describe the process of using the LLM as a constraint generator, and the different approaches to final layout generation we explored. The LLM Stability and Agent Simulation tests are part of experimental validation and are thus described in the next section.

3.1 Game Level Description Language (GLDL)

GLDL, or Game Level Description Language, is a descriptive language, partly inspired by the more generic Scenic language [7] but more specifically focused on systematically representing three-dimensional game levels. By providing a structured framework for defining facilities, spatial variables, and constraints, GLDL facilitates the automated creation of coherent, contextually meaningful, and playable environments. Facilities include all objects that may appear in a game scenario, including fundamental architectural elements, interactive features, and non-player characters (NPCs). This section presents the core components of GLDL, clarifies the fundamental sentence types, categorizes constraints, and highlights the practical advantages of adopting GLDL. A more complex language definition is given in Appendix A.

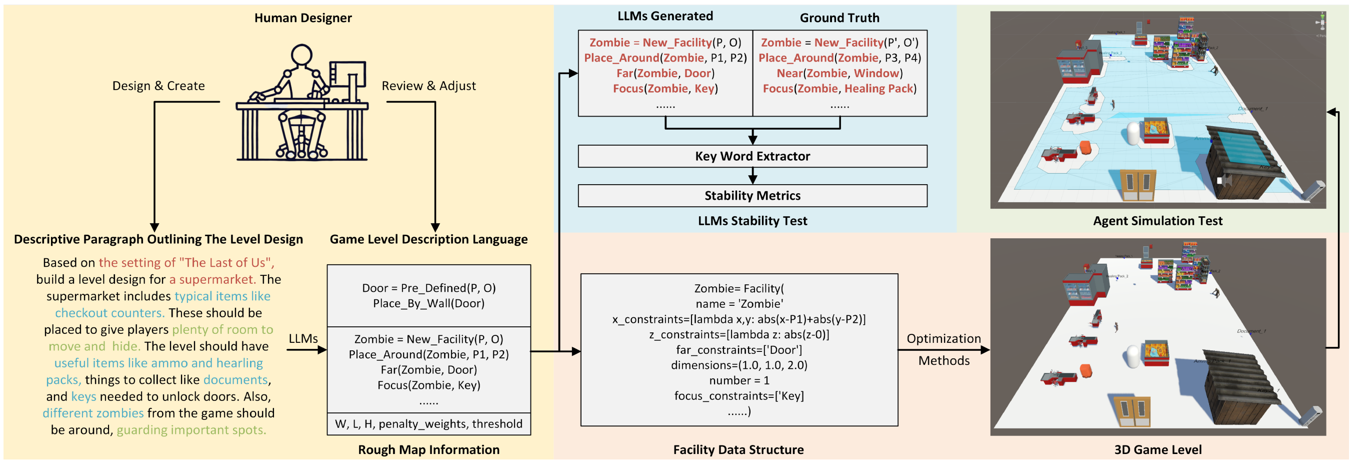
Human Designer
Design & Create
Review & Adjust
Descriptive Paragraph Outlining The Level Design
Based on the setting of "The Last of Us", build a level design for a supermarket. The supermarket includes typical items like checkout counters. These should be placed to give players plenty of room to move and hide. The level should have useful items like ammo and hearling packs, things to collect like documents, and keys needed to unlock doors. Also, different zombies from the game should be around, guarding important spots.
LLMs
Game Level Description Language
Door = Pre_Defined(P, O)
Place_By_Wall(Door)
Zombie = New_Facility(P, O)
Place_Around(Zombie, P1, P2)
Far(Zombie, Door)
Focus(Zombie, Key)
......
W, L, H, penalty_weights, threshold
Rough Map Information
LLMs Generated
Ground Truth
Zombie = New_Facility(P, O)
Place_Around(Zombie, P1, P2)
Far(Zombie, Door)
Focus(Zombie, Key)
......
Zombie = New_Facility(P', O')
Place_Around(Zombie, P3, P4)
Near(Zombie, Window)
Focus(Zombie, Healing Pack)
......
Key Word Extractor
Stability Metrics
LLMs Stability Test
Zombie= Facility(
name = 'Zombie'
x_constraints=[lambda x,y: abs(x-P1)+abs(y-P2)]
z_constraints=[lambda z: abs(z-0)]
far_constraints=['Door']
dimensions=(1.0, 1.0, 2.0)
number = 1
focus_constraints=['Key]
......)
Facility Data Structure
Optimization Methods
Agent Simulation Test
3D Game Level

Figure 1: Overall workflow of our Procedural Content Generation framework: a natural language paragraph describing the target 3D environment is processed by an LLM into GLDL constraints (yellow, left); these constraints are verified via an LLM Stability Test (blue, upper-middle), where red text highlights key components (facility names, constraint types, variable types), and then fed into an optimization algorithm to produce the facility layout (beige, bottom-right); finally, an Agent Simulation Test in Unity (green, upper-right) verifies that the resulting 3D level meets the intended navigability, complexity, and overall design requirements.

3.1.1 Vocabulary and Core Components. GLDL is fundamentally based on the concept of facilities, providing a means to describe the constraints and requirements associated with objects, NPCs, and other entities within a game. The language is comprised of two primary types of sentences: definition sentences and constraint sentences. Definition sentences are employed to declare and parameterize facilities, effectively establishing the fundamental components of the game level. Subsequently, constraint sentences are used to impose specific constraints and requirements on these defined facilities. The typical workflow involves first defining each object with appropriate definition sentences, followed by applying basic constraints relevant to their individual characteristics, such as positional requirements or spatial affiliations within a room. When introducing new facilities that interact with existing ones, relationship constraint sentences are added to define their interdependencies and spatial relationships. This structured approach ensures that each facility is not only accurately defined but also coherently integrated into the overall game environment.

Definition and Constraint Sentences. GLDL specifications consist of two primary sentence types:

- **Definition Sentences:** These sentences declare and parameterize facilities (objects or reference points), following the format:

  $$F = \mathit{Definition}(\mathit{variables})$$

  In this context, F denotes a facility, while *Definition* serves as a placeholder for a specific type of definition sentence. The following subsections describe three variants of definition sentences supported by GLDL:
  - *PredefinedFacility*: A facility with a predetermined position and orientation, serving as a stable reference point. Such facilities are not treated as variables during optimization.
  - *NewFacility*: A facility whose position and orientation depend on specified constraints, enabling adaptive and dynamic layouts. New facilities are treated as variables in optimization.
  - *Points*: Special facilities functioning as fixed reference anchors, guiding the relative placement and orientation of other facilities.
- **Constraint Sentences:** These sentences impose spatial or relational rules on facilities, using the format:

  $$\mathit{Constraint}(F, \mathit{variables})$$

  In this expression, F denotes a facility, and the listed variables define how it relates to positions, other facilities, or environmental parameters. Constraints are categorized into two main types to streamline specification and facilitate reasoning about facility arrangements:
  - **Basic Constraints**: Define fundamental placement rules relative to the environment or specified reference points. Examples include *PlaceAroundPosition*, *PlaceInRange*, and *PlaceByWall*.
  - **Relationship Constraints**: Govern spatial and directional relationships between facilities, addressing aspects such as proximity, visibility, and orientation. Examples include *Near/Far*, *CanSee*, *Focus*, and *Alignment/Orientation*.

  Additional details and rules for different types of constraint sentences are provided in Appendix A.

Variables. Variables represent essential attributes and parameters that define facilities and inform their configuration (e.g., positions, orientations, tags, numbers). They appear in both definition and constraint sentences, ensuring flexible and scalable specifications as the environment or design requirements change over time.

3.1.2 Benefits of GLDL. GLDL offers several benefits for designers and automated processes:

Domain-Specific Simplicity: GLDL's constructs are designed for typical game design scenarios, making it more direct and comprehensible than generic specification languages intended for complex perceptual tasks.

Accessibility to Designers and LLMs: GLDL's straightforward components and syntax can be easily understood by designers without extensive technical expertise. Moreover, LLMs can interpret and produce GLDL code from concise prompts, thereby facilitating rapid initial development, enabling iterative refinement, and supporting collaborative design processes. While the GLDL is designed with an accessible syntax that does not require expert knowledge of its underlying ontology, our current evaluation assumes that users provide free-text inputs without needing to fully understand the detailed definitions and constraint structures. Future work will involve user studies with a diverse group of designers to better quantify the level of familiarity required for effective use.

3.2 LLM-Based Constraints Generation

With Large Language Models (LLMs), our approach translates natural language descriptions of game levels into GLDL. This process involves several key steps:

3.2.1 Prompt Engineering. In designing prompts for LLM-based constraints generation, we structure the prompts to clearly outline the objectives, components, and specific instructions necessary for producing GLDL-compliant syntax. This involves providing the LLM with a concise description of the game environment, defining facility types, their properties, and the relationships between them. By explicitly specifying the purpose of GLDL, the key components, and the syntax for definition and constraint sentences, we ensure that most of the generated output adheres to the required format and accurately reflects the intended level design. For example, when creating a game level, the prompt defines facility types such as `PredefinedFacility` and `NewFacility`, along with their attributes like position, orientation, and dimensions. Additionally, the prompt includes constraint functions like `PlaceByWall` and `CanSee` to guide the placement and relationships of these facilities. Importantly, the initial narrative description—crafted by a human designer (or alternatively generated by an LLM)—is provided as input along with an instruction template, which directs the LLM to translate the prompts into structured GLDL code.

For a concrete illustration of this prompt structure, see Appendix B. The examples detail how we instruct the LLM to create a post-apocalyptic supermarket level, enumerating facility definitions and constraints, highlighting orientation usage without quotes, and referencing environment variables. This ensures the generated GLDL code meets syntactical, spatial, and contextual requirements with minimal manual intervention.

3.2.2 Validation and Refinement. To ensure the accuracy and feasibility of the generated constraints, we implement a validation and refinement process where each generated GLDL script is automatically parsed and evaluated against a reference grammar and predefined domain-specific rules to detect any syntactic or semantic deviations. Most generated sentences are semantically reasonable and require no adjustment; however, this only confirms their validity for level construction, not that they meet all of our design expectations or build a fully playable level. Any discrepancies or errors are iteratively refined through additional prompting or manual adjustments to maintain the integrity of the GLDL definitions, while the match rate metric further assesses how well the scripts satisfy our qualitative criteria. In our experiments, less than 0.01% of the sentences require adjustment due to syntactic error, which is negligible.

3.2.3 Integration phase. The validated GLDL output is then integrated into our optimization pipeline. This smooth transition from natural language to structured data allows for the automated generation of complex game levels that are both diverse and contextually appropriate.

3.3 Optimization Algorithms

The generation of coherent and playable 3D game levels, as described by GLDL, can be formulated as a constrained optimization problem. Each candidate solution corresponds to a specific assignment of positional and orientational parameters for all non-predefined facilities, while predefined facilities maintain their fixed configurations. The optimization algorithms output detailed data specifying the positions and orientations of each facility, which are then imported into the game engine for simulation. The objective is to find a configuration of facilities that minimizes a composite metric reflecting constraints satisfaction, spatial coherence, and environmental quality.

3.3.1 Problem Formulation. At the core of our optimization is an objective function that incorporates multiple criteria:

- **Spatial Constraint Satisfaction:** Penalties are applied for violating GLDL-defined constraints, such as placing facilities outside the environment's bounds, overlapping objects, misalignments, incorrect orientations, and deviations from desired proximity or visibility conditions. These penalty terms are grouped together as spatial penalty P and ensure that the resulting layout adheres closely to the specified design rules.
- **Clustering and Sparsity Measures:** Beyond raw constraint satisfaction, we incorporate metrics that reflect the structural qualities of the environment. For instance, a cluster degree metric C may penalize excessively dense placements, while a sparsity metric S encourages a balanced distribution of facilities. By weighting these metrics alongside the penalty terms, we guide the optimization process to produce not only feasible but also aesthetically and functionally appealing layouts.

Our goal is to find the optimal positions and orientations of the facilities that minimize the total metric. The final objective function aggregates these terms into a weighted sum:

$$\text{TotalMetric} = w_p \cdot P + w_c \cdot C + w_s \cdot S$$

where w_p, w_c, and w_s are weights for the penalty, cluster degree, and sparsity metrics, respectively; P represents the total penalty, C the cluster degree, and S the sparsity metric:

- P **(TotalSpatialPenalty)**: P is the cumulative sum of all penalties resulting from the violation of GLDL constraints. These penalties (detailed in Appendix D) include but are not limited to placing facilities outside the designated bounds, overlapping multiple facilities, misaligning facility orientations, and failing to meet proximity or visibility requirements. Formally, it can be expressed as:

$$P = \sum_{\text{constraints}} \mathit{PenaltyTerm}$$

Each PenaltyTerm quantifies the degree of violation for a specific constraint, weighted appropriately to reflect its importance in the overall layout quality. P ensures adherence to GLDL constraints by penalizing any violations, thereby maintaining the structural and functional integrity of the layout. For additional details on the penalty term, please refer to Appendix D.

- C **(ClusterDegree)**: C quantifies the degree of clustering among facilities within the environment. For every unique pair of placed facilities (f_i, f_j), it calculates the inverse of their Euclidean distance, summing these values to obtain the clustering degree:

$$C = \sum_{i<j} \frac{1}{\text{dist}(f_i, f_j) + \varepsilon}$$

where $\text{dist}(f_i, f_j)$ is the Euclidean distance between facilities f_i and f_j, and ε is a small constant to prevent division by zero. A higher C indicates tighter clustering, which may be undesirable depending on the design goals. C discourages excessive clustering of facilities, promoting a more uniform and balanced distribution across the environment.

- S **(SparsityMetric)**: S measures the extent of empty space within the environment by evaluating the distribution of facilities. The environment is sampled at discrete grid points, and for each grid point gp, the minimum distance to any facility is computed. The sparsity metric is then defined as the maximum of these minimum distances:

$$S = \max_{\text{grid point } gp} \left(\min_{\text{facilities } f} \text{dist}(gp, f) \right)$$

Here, $\text{dist}(gp, f)$ represents the Euclidean distance from grid point gp to facility f. A lower S value suggests a more uniformly distributed set of facilities with fewer large empty regions. We selected our sparsity metric over well-known spatial entropy due to its computational efficiency for large-scale 3D optimization and intuitive interpretability of results as maximum void distances. S prevents the existence of large vacant areas by encouraging facilities to occupy the space evenly, ensuring spatial balance and enhancing navigability.

By adjusting the weights w_p, w_c, and w_s, designers and automated pipelines can prioritize different aspects of the level design. The optimization process seeks to minimize the TotalMetric, thereby achieving a layout that satisfies constraints, avoids overcrowding, and maintains an even distribution of facilities within the environment. Notably, TotalMetric's clustering penalties are tuned for our survival-horror design context, where a dense grouping of items enhances the tension. However, the modular framework allows designers to adjust these weights—for instance, reducing clustering penalties for RPG resource hubs—or even introduce new metrics, such as loot distribution fairness for MMOs or narrative item proximity for story-driven games, to better match genre-specific priorities.

3.3.2 Optimization Methods. We employ three optimization algorithms and select the most effective one during the testing phase to design facility layouts that minimize TotalMetric while adhering to spatial constraints:

- **Simulated Annealing:** Simulated Annealing (SA) optimizes facility layouts by emulating the physical annealing process. Starting with an initial layout, SA iteratively modifies facility positions and orientations to minimize the TotalMetric. At each step, it evaluates the new configuration and probabilistically accepts changes that may increase the metric, controlled by a temperature parameter. This approach allows SA to escape local minima, progressively refining the layout towards a configuration that balances constraint satisfaction (P), clustering avoidance (C), and spatial balance (S).
- **Genetic Algorithms:** Genetic Algorithms (GA) approach layout optimization through an evolutionary framework. Each candidate layout is encoded as a chromosome, with genes representing facility parameters. GA operates over generations, applying selection, crossover, and mutation to evolve layouts that minimize the TotalMetric. By evaluating fitness based on penalties, GA effectively explores diverse solutions and converges towards high-quality layouts that satisfy constraints and maintain spatial coherence.
- **Greedy Methods:** Greedy Methods construct facility layouts incrementally by making the locally optimal choice at each placement step. When positioning a facility, the algorithm selects the location and orientation that immediately reduces the TotalMetric, focusing on minimizing penalties based on the current partial layout. While not guaranteeing a global optimum, Greedy Methods offer computational efficiency, making them suitable for rapid layout generation or as initial solutions and baseline models for further optimization by SA or GA.

4 Experiments & Analysis

This section presents a series of experiments designed to validate LLM-generated GLDL data, assess the effectiveness of our optimization algorithms, and verify the playability of generated levels through Unity-based simulations. The experiments are organized into three main stages:

(1) LLM Stability Validation Test
(2) Optimization Test
(3) Unity Simulation Test

4.1 LLM Stability Validation Test

This experiment evaluates the consistency and reliability of LLM-generated GLDL scripts. We focus on five distinct scene descriptions based on the background settings of "The Last of Us" and "Left 4

Dead 2": one baseline scenario (a supermarket environment manually designed by us) and four additional scenes (Bank Lobby, Hotel Lobby, Abandoned Church, and Music Festival) that were fully generated by the LLM. Each scene description was used to produce 1,000 GLDL scripts via the given prompt, resulting in a total of 5,000 generated scripts using the GPT-4o-mini model. To further validate the robustness and consistency of our method across different LLM capabilities, we conducted two additional experiments: generating 1000 supermarket scenes using the o1-mini model and another 1000 scenes using the GPT-4o model.

4.1.1 LLM Prompts and Scenes. We provided the LLM with a structured prompt outlining GLDL syntax, rules, and example sentences in Appendix B. The LLM was then given five different scene descriptions—one baseline (supermarket) and four generated by the LLM itself (bank lobby, hotel lobby, abandoned church, music festival)—and instructed to produce 1,000 scripts per scene. For example baseline scene descriptions, please refer to Appendix C.

4.1.2 Metrics and Similarity Assessment. To evaluate the generated scripts, we employed a custom Python tool that:

(1) Classifies each sentence into one of three categories: Definitions (facility declarations), Basic Constraints (e.g., PlaceByWall, PlaceInRange, PlaceAroundPosition), and Relationship Constraints (e.g., Near, CanSee, Focus, Alignment).
(2) Parses and normalizes sentences to extract key components (facility names, constraint types).
(3) Measures sentence-level similarity by comparing structure and key elements with a ground truth script for the baseline scenario. The baseline ground truth script (supermarket scenario) was human-curated through iterative designer adjustments to ensure narrative and gameplay alignment, while other scenes' ground truth specifications were programmatically selected from LLM-generated candidates via automated filtering based on structural diversity and constraint completeness metrics, followed by manual review and viability testing through agent simulations to validate playability and logical consistency.
(4) Computes appearing frequencies and match rates for each category. Match Rate quantifies the percentage of generated sentences that structurally and semantically align with the ground truth GLDL, by ensuring that each sentence uses the correct constraint type, valid facility references, and parameter values within 20% of the expected ranges (Match Rate = Matching Sentences / Total Sentences × 100%).

This method ensures a quantitative assessment of how accurately the LLM replicates the structure and semantics of the GLDL specification. It is important to note that higher match rates primarily reflect the LLM's ability to consistently reproduce the expected GLDL structure, which is crucial for automated parsing and downstream optimization. However, because constraints may sometimes conflict, match rates are not taken as a sole indicator of quality but are used in conjunction with qualitative assessments and simulation tests to evaluate the overall level of coherence and playability.

4.1.3 Results and Analysis. Table 1 summarizes the average match rates for each category across all five scenes. Notably, the supermarket (baseline) scenario revealed the highest overall similarity,

Table 1: LLM Stability Validation Results for Each Scene

Scene	Definition (%)	Basic (%)	Rel. (%)
Supermarket (Baseline)	91.77	51.03	54.30
Bank Lobby	73.69	52.63	18.09
Hotel Lobby	93.40	55.02	16.87
Abandoned Church	82.89	64.83	17.71
Music Festival	69.14	63.38	8.09
Baseline (o1-mini)	98.75	56.00	80.71
Basline (4o)	94.58	53.33	71.43

while scenes entirely generated by the LLM showed more variability in basic and relationship constraints. To ensure the robustness and consistency of our experiments, we incorporated additional validation tests using different versions of the LLM. The initial dataset comprised 5,000 scripts generated by GPT-4o-mini, a model known for its limitations in handling complex instructions. The supplementary experiments aimed to demonstrate that even with less capable models, our framework maintains a reasonable level of accuracy, and with more advanced models, performance is expected to improve, thereby ensuring the consistency and robustness of our optimization framework.

We further aggregated sentence frequencies from all 5,000 scripts, as summarized in Table 2. Definitions were the most prevalent, while relationship constraints, though fewer, exhibited lower match rates due to their inherent complexity. In aggregate, definitions ranged from 23,000 to 34,000 instances, basic constraints from 15,000 to 31,000, and relationship constraints from 7,000 to 19,000. These aggregated results indicate that essential foundational sentences are generated with high consistency, ensuring that critical elements are rarely missed. Meanwhile, the more varied match rates for relationship constraints meet our expectations by introducing diversity into the levels while maintaining a consistent core framework.

Table 2: Aggregated Sentence Frequencies Across All 5,000 Scripts (Not Including Extra Baseline Tests)

Sentence Type	Frequency	Avg Match Rate (%)
Definition	131000	82.36
Basic Constraint	131000	57.77
Relationship Constraint	55000	25.38

Overall, our results confirm that LLMs reliably reproduce facility definitions with high consistency, while complex relational constraints exhibit greater creativity and diversity. This consistency validates our approach, and future improvements may involve advanced models or a hybrid method that integrates automated generation with targeted manual refinement.

4.2 Optimization Test

After selecting a baseline GLDL script generated from the LLM outputs, we integrated it into the optimization pipeline to produce 10,000 final layout configurations for each optimization method. These configurations were generated by applying the previously introduced optimization algorithms under the chosen parameters and weights, ensuring that each resulting layout was guided by the defined constraints and environmental conditions. For constraints-based optimization, we tested the Simulated Annealing algorithm, Genetic Algorithm, and Greedy method, subsequently selecting the method with the best performance for further tasks.

4.2.1 Parameters and Weights. We adopted environment dimensions representative of a supermarket setting ($W = 20m, L = 30m, H = 6m$) and set the near/far thresholds (5m and 10m, respectively) to ensure appropriate facility spacing. Penalty weights were assigned to various constraints (e.g., overlaps, boundary violations, and misalignments), balancing their relative importance. The overall metric weights (w_p for penalty, w_c for cluster degree, and w_s for sparsity) were chosen to maintain an equilibrium between strict constraint adherence and spatial quality. Comprehensive details on the calculation of penalties are provided in Appendix D.

4.2.2 Results and Analysis. We began by fine-tuning the penalty and metrics weights to ensure that the optimization process would provide desirable results. After selecting weights (shown in Table 4), which generally produced stable numerical outcomes and visually coherent layouts, we evaluated the performance of three optimization methods—Simulated Annealing, Genetic Algorithm, and Greedy heuristic—by generating 10,000 layouts for each method to determine the most effective approach for our constraints-based optimization task. Table 3 presents the average total weighted metric values and their standard deviations for each method.

Table 3: Total Weighted Metric for Different Optimization Methods. Values normalized to [0,1] scale relative to maximum observed metric. Abbreviations: SA = Simulated Annealing, GA = Genetic Algorithm, Greedy = Greedy Method

Method	Total Metric	Norm. Metric	Std. Dev.
SA	35337.70	0.56	2912.642
Greedy	58947.89	0.94	13079.87
GA	62600.97	1.00	3948.396

SA achieved the lowest total weighted metric (35,337.7), followed by the Greedy method (58,947.9), with GA yielding the highest values (62,600.9). While Greedy outperformed others in computational speed, its higher standard deviation (13,079 vs. SA's 3,948) indicates less stable solutions compared to SA. In contrast, SA maintained a good balance between numerical performance and visual quality, making it the most dependable choice for this scenario. Therefore, we selected SA for further experimentation and analysis.

Focusing on SA, we evaluated 10,000 generated layouts using our chosen penalty and metric weighting scheme. As shown in Table 4, the proximity constraints (*penalty_near* and *penalty_far*) contributed 31.70% and 26.20% of the total weighted metric despite moderate weights (10 and 15), highlighting their key role in shaping spatial relationships while allowing controlled flexibility in facility placement. In contrast, the high weights assigned to *penalty_overlap* (30) and *penalty_bounds* (30) effectively enforced collision-free layouts within environmental boundaries, as evidenced by their negligible weighted contributions (0.05% and 0.003%). Meanwhile, vertical placement constraints (*penalty_z*) and orientation requirements contributed modestly (2.78% and 1.81%), reflecting minor but consistent deviations from ideal configurations.

Table 4: Penalty Weights and Average Metrics from 10,000 Generated Layouts with Weighted Averages.

Metric	Weight	Avg. Value	Weighted Avg.
penalty_xy	18	75.39	1357.02 (3.84%)
penalty_z	20	49.06	981.20 (2.78%)
penalty_near	10	1120.33	11203.30 (31.70%)
penalty_far	15	616.95	9254.25 (26.20%)
penalty_can_see	2	4.79	9.58 (0.03%)
penalty_overlap	30	0.59	17.70 (0.05%)
penalty_bounds	30	0.04	1.20 (0.003%)
penalty_alignment	15	31.31	469.65 (1.33%)
penalty_orientation	20	31.97	639.40 (1.81%)
penalty_focus	10	9.42	94.20 (0.27%)
total_penalty	1	24027.48	24027.48 (67.99%)
cluster_degree	100.0	77.57	7757.00 (21.95%)
sparsity_metric	500.0	7.11	3555.00 (10.07%)
total_weighted_metric			35337.70 (100%)

The cluster degree metric ($w_c = 100.0$) contributed 21.95% by discouraging excessive facility density, while the sparsity metric ($w_s = 500.0$) accounted for 10.07%, thereby ensuring a balanced spatial distribution. Notably, the aggregate of all penalty terms amounted to 67.99% of the total weighted metric, which validates our constraint-driven optimization approach. These results indicate that constraints with lower weights but higher frequencies of violation (such as proximity rules) naturally emerge as dominant cost drivers, whereas high-weight penalties (such as overlaps) serve as strict guardrails with minimal impact on the overall cost due to their near-zero incidence of violation.

The successful alignment between penalty weights and average metrics confirms that our optimization approach is both proper and valid for generating game level layouts. Prioritizing high-weight constraints ensured collision-free, boundary-compliant designs, while balanced cluster and sparsity weights promoted navigable environments. Future work could explore adaptive weighting strategies based on layout complexity or playtesting feedback, alongside enhanced penalties and spatial metrics to improve proximity adherence and spatial coherence.

4.3 Unity Simulation Test

Experiment Settings. To evaluate the playability and coherence of the generated layouts, we imported 10,000 optimized configurations into a Unity environment to simulate real-time gameplay. An AI agent, designed to mimic human exploration and interaction, navigates each level—which contains various interactive elements and enemies—with the goal of collecting all scattered keys, gathering items, and unlocking the exit to complete the level. Our AI

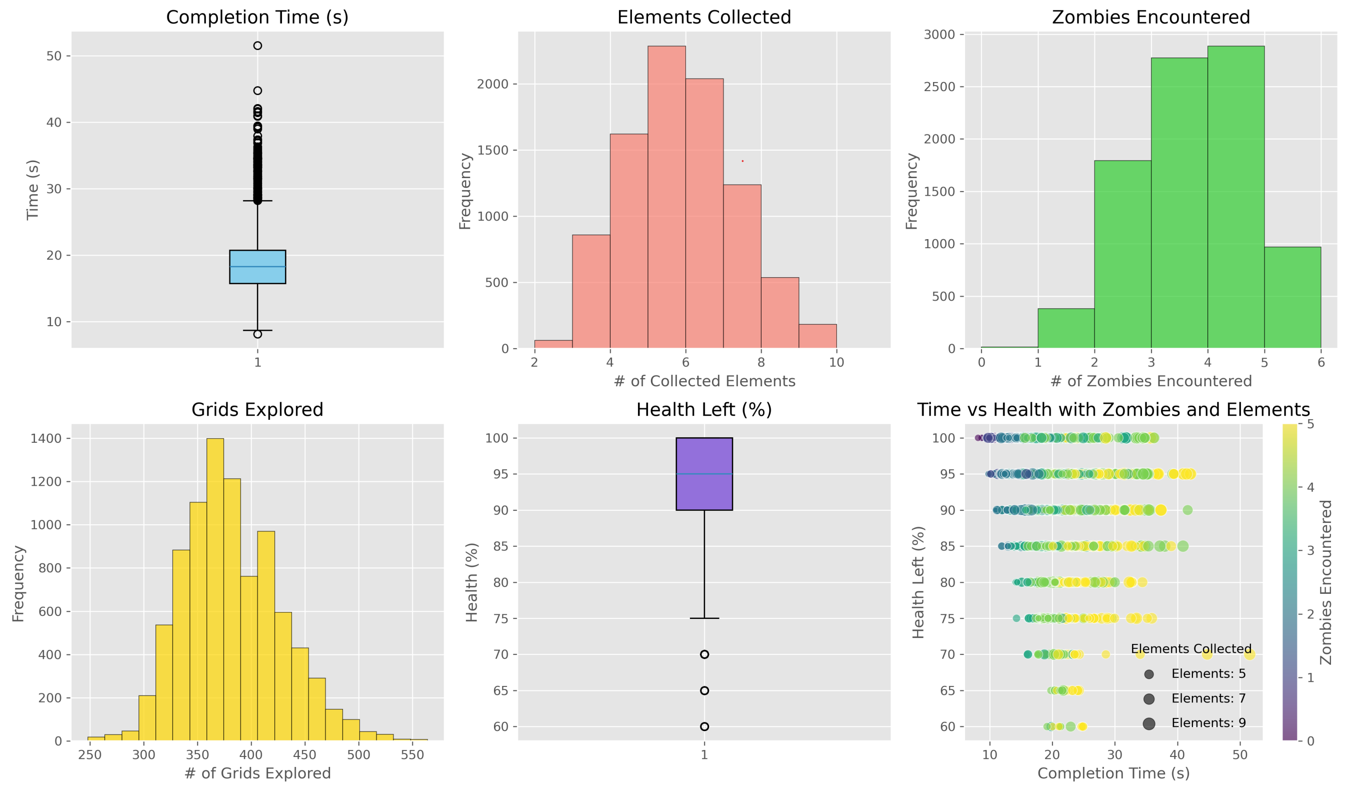

Figure 2: Overview of key simulation metrics. The boxplots of completion time (top-left) and final health (bottom-middle) reflect the overall difficulty, while the histograms of elements collected (top-middle), zombies encountered (top-right), and grids explored (bottom-left) indicate the usage and impact of the level's designed features. The scatterplot (bottom-right) combines completion time, health, and enemy encounters, revealing how difficulty and content utilization intersect within the simulated environment.

logic (see Algorithm 1) works as follows: when the agent finds a key, it records the location and prioritizes the nearest key; if an enemy is encountered, the agent stops to engage; if items are within reach, they are collected; and if no keys are visible, the agent moves toward areas with the most unexplored grid cells:

$$P^* = \arg\max_P \sum_{(i,j)\in \text{Grid}} \mathbb{1}_{\text{Unseen}(i,j)} \cdot e^{-\beta\cdot \text{dist}(P,(i,j))}$$

where $\mathbb{1}_{\text{Unseen}(i,j)}$ equals 1 if a grid cell is unseen and 0 otherwise, β is a decay factor that prioritizes nearby cells, and $\text{dist}(P,(i,j))$ is the Euclidean distance between P and grid cell (i,j).

Enemies were assigned distinct properties and parameters, reacting by pursuing and attacking the agent upon entering their detection range. A level was deemed failed if the agent lost all HP, any of the keys became unreachable, or if certain paths were invalid, impeding level completion. Detailed parameter configurations are provided in Appendix E.

In addition to evaluating our framework on the 10,000 main configuration dataset, we conducted two sub-tests of 1,000 simulations each to demonstrate tunability and controllability. In the first sub-test (*Combat Density Test*), the LLM was prompted to create a scenario with more frequent enemy encounters. In the second sub-test (*Task Duration Test*), the LLM was instructed to generate a level configuration likely to prolong the agent's overall completion time. Both sub-tests followed the framework pipeline described earlier, resulting in the same Unity simulation test with identical agent logic and environment parameters.

Results and Analysis. Figure 2 shows key gameplay metrics from 10,000 main simulations. Completion times generally range from 10 to 25 seconds (17%–42%), with some outliers over 50 seconds. The number of collected elements typically falls between 5 and 7 (average 5.38, 54%), and zombie encounters average 3.25 (54%). Most simulations explore between 300 and 450 grids (55%–82%), and agent health usually remains above 90%. The scatterplot further indicates that longer completion times are associated with lower health and higher enemy encounters.

Table 5 summarizes the average metrics for the three test conditions, with each raw value accompanied by its corresponding normalized percentage. For example, under the default configuration, the AI agent completes levels in an average of 18.58 seconds (31%), collects 5.38 elements (54%), encounters 3.25 zombies (54%), explores 382.07 grids (70%), and retains 92.43% health.

For tunability, two additional GLDL scripts were generated. The Combat Test, which increases enemy density and complexity, leads to a modest increase in completion time and a significant rise in

Table 5: Unity Simulation Test Results

Metric (Max)	Default Generation	Combat Density Test	Task Duration Test
Number of Simulations	10,000	1,000	1,000
Average Time Taken (s) (60)	18.58 (31.0%)	20.32 (33.9%)	25.13 (41.9%)
Min Time (s) (60)	8.14 (13.6%)	10.04 (16.7%)	14.53 (24.2%)
Max Time (s) (60)	51.55 (85.9%)	42.14 (70.2%)	55.42 (92.4%)
Average Elements Collected (10)	5.38 (53.8%)	5.47 (54.7%)	5.74 (57.4%)
Average Zombies Encountered (6)	3.25 (54.2%)	3.89 (64.8%)	3.83 (63.8%)
Average Grids Explored (550)	382.07 (69.5%)	408.37 (74.3%)	398.31 (72.4%)
Average Health Left (%) (100)	92.43 (92.4%)	85.54 (85.5%)	91.05 (91.1%)
Invalid Levels			
Time Exceeded	228 (2.28%)	12 (1.2%)	19 (1.9%)
Unreachable Keys	950 (9.5%)	78 (7.8%)	47 (4.7%)
Environment Details			
Width (m)	20	20	20
Length (m)	30	30	30
Height (m)	6	6	6
Interaction Elements	10	10	10
Enemies	5	5	5
AI Agent Health	100	100	100
Time Threshold	60s	60s	60s

Algorithm 1 AI Agent Navigation and Interaction Logic

```
1: Initialize: Memory $\leftarrow \emptyset$
2: while GameRunning do
3:     $\mathcal{K} \leftarrow$ DetectKeys()
4:     $\mathcal{E} \leftarrow$ DetectEnemies()
5:     $\mathcal{I} \leftarrow$ DetectInteractionElements()
6:     Memory $\leftarrow$ Memory $\cup \mathcal{K}$
7:     if $\mathcal{E} \neq \emptyset$ then
8:         EngageEnemy($\mathcal{E}$)
9:     else if $\exists i \in \mathcal{I}$ : distance($i$) $\leq$ collectRange then
10:        MoveToAndCollect($i$)
11:    else if Memory $\neq \emptyset$ then
12:        $k^* = \arg\min_{k \in \text{Memory}} \text{distance}(k)$
13:        MoveTo($k^*$)
14:        CollectKey($k^*$)
15:        Memory $\leftarrow$ Memory $\setminus \{k^*\}$
16:        if AllKeysCollected $\wedge$ ExitReachable then
17:            MoveToExit()
18:        end if
19:    else
20:        $P^* = \arg\max_P \sum_{(i,j) \in \text{Grid}} \mathbb{1}_{\text{Unseen}(i,j)} e^{-\beta \cdot \text{dist}(P,(i,j))}$
21:        MoveTo($P^*$)
22:    end if
23: end while
```

zombie encounters (from 54% to 65%), with final health dropping to 85.5%. In contrast, the Task Test, which spaces key objectives further apart, extends the average completion time to 25.13 seconds (42%) and slightly increases the average elements collected to 5.74 (57%). These results demonstrate that the framework can be effectively tuned to produce distinct gameplay profiles by modifying LLM prompts or directly adjusting facility constraints.

Furthermore, the 228 levels that exceeded the time threshold and the 950 levels with unreachable keys highlight specific areas for future optimization. Overall, the system achieved an 88.22% validity rate, indicating that the majority of generated levels are both playable and compliant with essential design constraints. Future work will involve human playtesting to further validate these automated assessments.

5 Conclusion

In this study, we presented a framework that integrates Large Language Models, a specialized Game Level Description Language, and facility layout optimization to generate coherent, playable 3D game levels automatically. Starting from narrative-level descriptions, the approach produces layouts that reflect intended spatial constraints and gameplay mechanics. Agent-based simulations confirm that these environments are not only navigable and engaging but also maintain structural coherence and controllability.

Nonetheless, the current framework has limitations. While GLDL effectively encodes basic constraints, it does not capture the full range of complex constraints that game designers may require—some of which necessitate combinations of multiple sentences and additional manual fine-tuning. Similarly, the simplified assumption that both rooms and objects can be represented as cubes oversimplifies many real-world scenarios. Potential improvements include approximating irregular shapes with minimal encompassing cubes, using predefined facilities for filler regions, or employing meshes and advanced geometric representations, even if these methods increase computational overhead. Additionally, the current validation is limited to a single game genre (survival-horror) and a single AI agent playstyle, which may not fully capture the diverse requirements of other game types.

Future work includes extending this framework to larger-scale scenes, incorporating temporal dimensions, and introducing more game mechanics and logical elements to support dynamic scenarios that consider game flow and narrative progression. Testing the framework across multiple game genres (e.g., RPGs, strategy games) with genre-specific facilities and constraints would further validate its generalizability. Employing diverse AI agents with varying behavior profiles (e.g., stealth-based navigation vs. combat-oriented strategies) would ensure robustness against overfitting to a single playstyle and enhance the framework's adaptability to heterogeneous player interactions. By building facility data locally, the framework can be adapted to operate entirely offline, reducing reliance on online services. By addressing these challenges and leveraging local assets, the approach can evolve to design dynamic environments that better accommodate the complex needs of contemporary game development.

Acknowledgments

This research was supported by NSERC Discovery Grant RGPIN-2019-05213. We acknowledge that parts of the manuscript text were modified using generative AI, with all changes reviewed by authors.

References

[1] Rio Aguina-Kang, Maxim Gumin, Do Heon Han, Stewart Morris, Seung Jean Yoo, Aditya Ganeshan, R. Kenny Jones, Qiuhong Anna Wei, Kailiang Fu, and Daniel Ritchie. 2024. Open-Universe Indoor Scene Generation using LLM Program Synthesis and Uncurated Object Databases. arXiv:2403.09675 [cs.CV] https://arxiv.org/abs/2403.09675

[2] Abbas Ahmadi, Mir Saman Pishvaee, and Mohammad Reza Akbari Jokar. 2017. A survey on multi-floor facility layout problems. *Computers & Industrial Engineering* 107 (2017), 158–170. https://doi.org/10.1016/j.cie.2017.03.015

[3] anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family

[4] Aren A. Babikian, Oszkár Semeráth, and Dániel Varró. 2024. Concretization of Abstract Traffic Scene Specifications Using Metaheuristic Search. *IEEE Transactions on Software Engineering* 50, 1 (2024), 48–68. https://doi.org/10.1109/TSE.2023.3331254

[5] Dave Epstein, Ben Poole, Ben Mildenhall, Alexei A. Efros, and Aleksander Holynski. 2025. Disentangled 3D scene generation with layout learning. In *Proceedings of the 41st International Conference on Machine Learning (ICML'24)*. JMLR.org, Vienna, Austria, Article 500, 13 pages.

[6] Chuan Fang, Yuan Dong, Kunming Luo, Xiaotao Hu, Rakesh Shrestha, and Ping Tan. 2024. Ctrl-Room: Controllable Text-to-3D Room Meshes Generation with Layout Constraints. arXiv:2310.03602 [cs.CV] https://arxiv.org/abs/2310.03602

[7] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 63—-78. https://doi.org/10.1145/3314221.3314633

[8] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. 2006. Persistent realtime building interior generation. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames* (Boston, Massachusetts) *(Sandbox '06)*. Association for Computing Machinery, New York, NY, USA, 179—-186.

[9] Chengpeng Hu, Yunlong Zhao, and Jialin Liu. 2024. Game Generation via Large Language Models. In *2024 IEEE Conference on Games (CoG)*. 1–4. https://doi.org/10.1109/CoG60054.2024.10645597

[10] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. 2024. SceneCraft: an LLM agent for synthesizing 3D scenes as blender code. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 776, 31 pages.

[11] Zehua Jiang, Sam Earle, Michael Green, and Julian Togelius. 2022. Learning Controllable 3D Level Generators. In *Proceedings of the 17th International Conference on the Foundations of Digital Games* (Athens, Greece) *(FDG '22)*. Association for Computing Machinery, New York, NY, USA, Article 71, 9 pages.

[12] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. PCGRL: Procedural Content Generation via Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 16, 1 (Oct. 2020), 95–101. https://doi.org/10.1609/aiide.v16i1.7416

[13] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N. Yannakakis, and Julian Togelius. 2021. Deep learning for procedural content generation. *Neural Computing and Applications* 33, 1 (Jan. 2021), 19–37. https://doi.org/10.1007/s00521-020-05383-8

[14] Riccardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Bidarra. 2010. A Constrained Growth Method for Procedural Floor Plan Generation. In *Proceedings of the 11th International Conference on Intelligent Games and Simulation (GAMEON 2010)*. Leicester, United Kingdom, 13–20.

[15] Jess Martin. 2006. Procedural House Generation: A Method for Dynamically Generating Floor Plans. In *Proceedings of the Symposium on Interactive Computer Graphics and Games*.

[16] OpenAI. 2024. Learning to Reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/

[17] Ryan Po and Gordon Wetzstein. 2024. Compositional 3D Scene Generation using Locally Conditioned Diffusion. In *2024 International Conference on 3D Vision (3DV)*. 651–663. https://doi.org/10.1109/3DV62453.2024.00026

[18] Xuanchi Ren, Jiahui Huang, Xiaohui Zeng, Ken Museth, Sanja Fidler, and Francis Williams. 2024. XCube: Large-Scale 3D Generative Modeling using Sparse Voxel Hierarchies. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4209–4219.

[19] Jonas Schult, Sam Tsai, Lukas Höllein, Bichen Wu, Jialiang Wang, Chih-Yao Ma, Kunpeng Li, Xiaofang Wang, Felix Wimbauer, Zijian He, Peizhao Zhang, Bastian Leibe, Peter Vajda, and Ji Hou. 2024. ControlRoom3D: Room Generation using Semantic Proxy Rooms. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6201–6210.

[20] Noor Shaker, Julian Togelius, and Mark Nelson. 2016. *Procedural Content Generation in Games*. Springer. https://doi.org/10.1007/978-3-319-42716-4

[21] S. P. Singh and R. R. K. Sharma. 2006. A Review of Different Approaches to the Facility Layout Problems. *Int J Adv Manuf Technol* 30 (2006), 425–433. https://doi.org/10.1007/s00170-005-0087-9

[22] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* 10, 3 (2018), 257–270. https://doi.org/10.1109/TG.2018.2846639

[23] Graham Todd, Sam Earle, Muhammad Umair Nasir, Michael Cerny Green, and Julian Togelius. 2023. Level Generation Through Large Language Models. In *Proceedings of the 18th International Conference on the Foundations of Digital Games* (Lisbon, Portugal) *(FDG '23)*. Association for Computing Machinery, New York, NY, USA, Article 70, 8 pages. https://doi.org/10.1145/3582437.3587211

[24] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186. https://doi.org/10.1109/TCIAIG.2011.2148116

[25] Yi Wang, Jieliang Luo, Adam Gaier, Evan Atherton, and Hilmar Koch. 2024. PlotMap: Automated Layout Design for Building Game Worlds. In *2024 IEEE Conference on Games (CoG)*. 1–8. https://doi.org/10.1109/CoG60054.2024.10645627

[26] Jim Whitehead. 2020. Spatial Layout of Procedural Dungeons Using Linear Constraints and SMT Solvers. In *Proceedings of the 15th International Conference on the Foundations of Digital Games* (Bugibba, Malta) *(FDG '20)*. Association for Computing Machinery, New York, NY, USA, Article 101, 9 pages.

[27] Zhennan Wu, Yang Li, Han Yan, Taizhang Shang, Weixuan Sun, Senbo Wang, Ruikai Cui, Weizhe Liu, Hiroyuki Sato, Hongdong Li, and Pan Ji. 2024. BlockFusion: Expandable 3D Scene Generation using Latent Tri-plane Extrapolation. *ACM Transactions on Graphics* 43, 4 (2024), 17 pages. https://doi.org/10.1145/3658188

[28] Yongzhi Xu, Yonhon Ng, Yifu Wang, Inkyu Sa, Yunfei Duan, Yang Li, Pan Ji, and Hongdong Li. 2024. Sketch2Scene: Automatic Generation of Interactive 3D Game Scenes from User's Casual Sketches. arXiv:2408.04567 [cs.CV] https://arxiv.org/abs/2408.04567

[29] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, Chris Callison-Burch, Mark Yatskar, Aniruddha Kembhavi, and Christopher Clark. 2024. Holodeck: Language Guided Generation of 3D Embodied AI Environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 16227–16237.

[30] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2024. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL] https://arxiv.org/abs/2303.18223

A GLDL Specification Details

This appendix provides a detailed reference for the syntax and format of the Game Level Description Language (GLDL) used in the main text.

A.1 Definition Sentences

Definition sentences are used to declare and parameterize the facilities (objects or reference points) that make up the game level. The general syntax is:

$$[F = \text{Definition}(\text{variables})]$$

Where F represents the facility being defined, and the `Definition` keyword can take one of three forms:

A.1.1 PredefinedFacility.

- Format: $F = \text{PredefinedFacility}(P, O, N, T)$
- Arguments:
  - P (Position): The fixed position of the facility
  - O (Orientation): The facing direction of the facility
  - N (Number): The number of instances
  - T (Tags): A list of tags associated with the facility

A.1.2 NewFacility.

- Format: $F = \text{NewFacility}(N, T)$
- Arguments:
  - N (Number): The number of instances
  - T (Tags): A list of tags associated with the facility

A.1.3 NewPoint.

- Format: $F = \text{NewPoint}(P)$
- Arguments:
  - P (Position): The fixed position of the point

A.2 Constraint Sentences

Constraint sentences impose spatial or relational rules on the facilities within the game level. The general syntax is:

$$[\text{Constraint}(F, \text{variables})]$$

Where F represents the facility being constrained, and the `Constraint` keyword can take various forms, such as:

A.2.1 Basic Constraints.

- **PlaceAroundPosition**: Places a facility near a specified position P.
  Format: $\text{PlaceAroundPosition}(F, P)$
- **PlaceInRange**: Confines a facility's placement within a bounding box defined by positions $P1$ and $P2$.
  Format: $\text{PlaceInRange}(F, P1, P2)$
- **PlaceByWall**: Requires a facility to be placed adjacent to a wall, with an optional orientation O.
  Format: $\text{PlaceByWall}(F, O)$

A.2.2 Relationship Constraints.

- **Near**: Specifies that two facilities $F1$ and $F2$ should be close to each other.
  Format: $\text{Near}(F1, F2)$
- **Far**: Specifies that two facilities $F1$ and $F2$ should be distant from each other.
  Format: $\text{Far}(F1, F2)$
- **CanSee**: Ensures a clear line of sight between two facilities $F1$ and $F2$.
  Format: $\text{CanSee}(F1, F2)$
- **Focus**: Orients one facility $F1$ towards another facility $F2$ or position P.
  Format: $\text{Focus}(F1, F2)$ or $\text{Focus}(F, P)$
- **Alignment**: Aligns two facilities $F1$ and $F2$ along one or more axes.
  Format: $\text{Alignment}(F1, F2)$ or $\text{Alignment}(F1, F2, \text{Axis})$
- **Orientation**: Matches the orientation of two facilities $F1$ and $F2$.
  Format: $\text{Orientation}(F1, F2)$
- **OnTheSide**: Places one facility $F1$ beside another facility $F2$, relative to their orientations.
  Format: $\text{OnTheSide}(F1, F2)$
- **Front**: Combines focus, orientation, and alignment to place one facility $F1$ in front of another facility $F2$.
  Format: $\text{Front}(F1, F2)$
- **Group**: Defines a parent-child relationship between two facilities $F1$ and $F2$, with a relative position P and orientation O.
  Format: $\text{Group}(F1, F2, P, O)$

B Prompt for LLM-Based Constraints Generation

This appendix presents the structured prompt used for LLM-based constraints generation:

You are a game level designer creating a detailed level for a 3D game set in a supermarket environment. Use the Game Level Description Language (GLDL) to define facilities, specify their properties, and apply constraints systematically. Follow these guidelines to ensure valid GLDL output:

1. Purpose of GLDL:

- Control facility placement, orientation, and relationships to create interactive environments.

2. Key Components:

- **Facilities**: `PredefinedFacility`, `NewFacility`, `Points`.
- **Variables**: Use `W`, `L`, `H` for environment dimensions.
- **Positions**: Tuples `(x, y, z)`; use `None` for unconstrained axes.

3. Definition Sentences:

- **PredefinedFacility**:

```
facility_name = PredefinedFacility(
    P=(x, y, z),
    O=Orientation,
    N=number,
    T=[tags],
    dimensions=(width, length, height)
)
```

- **NewFacility**:

```
facility_name = NewFacility(
        N=number,
        T=[tags],
        dimensions=(width, length, height)
    )
```

- **Points**:

  `point_name = NewPoint(P=(x, y, z))`

4. Constraint Sentences:

- **Placement**:
  - `PlaceByWall(facility, Orientation)`
  - `PlaceInRange(facility, pos1, pos2)`
  - `PlaceAroundPosition(facility, position)`
- **Relationships**:
  - `Near(facility1, facility2)`
  - `CanSee(facility1, facility2)`
  - `Focus(facility, target)`

5. Specific Instructions:

- Do not use quotes for orientation values (e.g., use `X` instead of `'X'`).
- Apply Z-axis constraints with `PlaceAroundPosition` using `None` for unconstrained axes.
- Define each constraint on a separate line.
- Ensure all referenced facilities and points are previously defined.

6. Output Format:

- One sentence per line for definitions and constraints.
- Exclude definitions of `W`, `L`, `H`.
- Adhere strictly to GLDL syntax for automated parsing.

Generate a GLDL file that includes all facility definitions and constraints based on the above guidelines, ensuring logical consistency and adherence to the specified game environment.

C Prompt for Game Level Description (Post-Apocalyptic Supermarket)

The level is set in a post-apocalyptic supermarket inspired by The Last of Us. The Main Entrance at the front is the primary way in for players, and an Emergency Exit at the back can serve as a possible escape. Near the entrance, several Checkout Counters line the wall, acting as both obstacles and reminders of the store's old daily operations. In the middle of the store, multiple Shelf Rows are neatly arranged, still holding some goods, and scattered Shopping Carts stand nearby. A notable Product Display sits near the center, highlighting featured items, while a Security Camera high on a wall keeps watch on both the Main Entrance and the Checkout Counters.

A Fire Alarm is located in the center of the store, and a Trash Bin is set up near the Checkout Counters. Along the walls, a Vending Machine and a Water Dispenser provide small sources of supplies. Lighting Fixtures are spaced across the store, and a small Restroom sits against one wall near the entrance. Ammo Packs, Healing Packs, and Documents appear on the shelves, each placed apart to encourage some searching. Several zombies now roam these aisles, each near important items or places, adding a serious threat to anyone who enters.

Freezer Displays and Refrigerated Displays stand near the center, in line with the Shelf Rows, while Stock Bags lie close by. All of these elements remain within the supermarket's walls, arranged in ways that create a tight, challenging experience for the player.

D Penalty Computation Details

This appendix provides detailed explanations of how penalties are derived. Each penalty type is computed by evaluating facility placements against specific constraints. The following provides detailed explanations of how each penalty is derived, incorporating mathematical formulations where applicable:

D.1 Penalty_x and Penalty_z

These penalties measure deviations from desired positions along the X and Z axes, respectively. For a facility positioned at coordinates (x, z), the penalties are calculated as the squared difference between the current position and the target position:

$$\text{penalty\_x} = w_x \cdot (x_{\text{current}} - x_{\text{target}})^2$$

$$\text{penalty\_z} = w_z \cdot (z_{\text{current}} - z_{\text{target}})^2$$

where w_x and w_z are the weights assigned to these penalties.

D.2 Penalty_near and Penalty_far

These penalties address the maintenance of specified proximity thresholds between pairs of facilities. For each pair of facilities (f_i, f_j), the Euclidean distance d_{ij} is computed:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (z_i - z_j)^2}$$

- **Penalty_near**: Applied when $d_{ij} < d_{\text{min}}$, where d_{min} is the minimum allowed distance.

$$\text{penalty\_near+} = w_{\text{near}} \cdot (d_{\text{min}} - d_{ij})^2 \quad \text{if } d_{ij} < d_{\text{min}}$$

- **Penalty_far**: Applied when $d_{ij} > d_{\text{max}}$, where d_{max} is the maximum allowed distance.

$$\text{penalty\_far+} = w_{\text{far}} \cdot (d_{ij} - d_{\text{max}})^2 \quad \text{if } d_{ij} > d_{\text{max}}$$

where w_{near} and w_{far} are the respective weights.

D.3 Penalty_can_see

This penalty ensures a clear line of sight between two facilities. A ray is cast from the center of one facility to the other. If the ray intersects any other facility, the penalty is incremented:

$$\text{penalty\_can\_see+} = w_{\text{can\_see}} \cdot \mathbb{1}_{\text{obstructed}}$$

where $w_{\text{can\_see}}$ is the weight and $\mathbb{1}_{\text{obstructed}}$ is an indicator function that is 1 if the line of sight is obstructed and 0 otherwise.

D.4 Penalty_overlap

Overlap penalties are incurred when facilities intersect in space. The volume of overlap V_{overlap} between two facilities is calculated, and the penalty is proportional to this volume:

$$\text{penalty\_overlap+} = w_{\text{overlap}} \cdot V_{\text{overlap}}$$

where w_{overlap} is the assigned weight.

D.5 Penalty_bounds

This penalty enforces that all facilities remain within the defined environment boundaries. For a facility extending beyond the bounds, the squared distance d_{bound} from the facility's position to the nearest boundary is calculated:

$$\text{penalty\_bounds+} = w_{\text{bounds}} \cdot d_{\text{bound}}^2 \quad \text{if out\_of\_bounds}$$

where w_{bounds} is the weight and out_of_bounds is a condition indicating boundary violation.

D.6 Penalty_alignment and Penalty_orientation

These penalties assess how well facilities align with designated axes or share the same orientation. For alignment along an axis, the deviation θ from the desired alignment angle is measured:

$$\text{penalty\_alignment+} = w_{\text{alignment}} \cdot \theta^2$$

Similarly, for orientation consistency:

$$\text{penalty\_orientation+} = w_{\text{orientation}} \cdot \theta_{\text{orientation}}^2$$

where $w_{\text{alignment}}$ and $w_{\text{orientation}}$ are the respective weights.

D.7 Penalty_focus

The focus penalty considers the angle between a facility's facing direction and a target direction. If this angle ϕ exceeds a threshold $\phi_{\text{threshold}}$, the penalty increases quadratically with the deviation:

$$\text{penalty\_focus+} = \begin{cases} w_{\text{focus}} \cdot (\phi - \phi_{\text{threshold}})^2 & \text{if } \phi > \phi_{\text{threshold}} \\ 0 & \text{otherwise} \end{cases}$$

where w_{focus} is the weight.

D.8 Aggregating Penalties

The final objective function aggregates all penalty terms into a weighted sum:

$$\text{TotalMetric} = w_p \cdot P + w_c \cdot C + w_s \cdot S$$

where:

- $P = \sum$ PenaltyTerms represents the total spatial penalty.
- C is the cluster degree.
- S is the sparsity metric.
- w_p, w_c, and w_s are the weights for penalty, cluster degree, and sparsity metrics, respectively.

Adjusting these weights allows designers to prioritize different aspects of the level layout, ensuring that the optimization process aligns with desired design goals.

These penalty computations collectively shape the final facility layout. By adjusting penalty weights and constraints, designers can steer the optimization towards layouts that prioritize certain aspects (e.g., minimizing overlaps, ensuring focus, or maintaining proper alignment) over others, resulting in a more tailored and coherent 3D game level environment.

E Simulation Parameters

This appendix provides detailed parameter configurations used in the Unity Simulation Test. The parameters are categorized into Zombie Types, NavMesh Agent Default Settings, and AI Agent Settings to ensure clarity and ease of reference.

Table 6: Zombie Types and Their Parameters. Abbreviations: BZ = Bloater Zombie, CZ = Crawler Zombie, CKZ = Clicker Zombie, GZ = Guard Zombie, RZ = Runner Zombie

Parameter	BZ	CZ	CKZ	GZ	RZ
Detection Range (m)	10	20	10	20	20
Field of View (°)	360	120	360	180	120
Attack Range (m)	4	3	3	3	3
Attack Damage	20	10	15	10	5
Attack Cooldown (s)	2	1	2	1	1
Move Speed (m/s)	2.5	4	3	4	6
Raycast Spread (°)	30	30	30	30	30

Table 7: NavMesh Agent Default Settings for AI Agent and Zombies

Parameter	Value
Speed (m/s)	3.5
Angular Speed (°/s)	120
Acceleration (m/s²)	8
Stopping Distance (m)	0
Obstacle Avoidance Radius (m)	0.5
Obstacle Avoidance Height (m)	2

Table 8: AI Agent Settings

Parameter	Value
Max Health	100
Shooting Interval (s)	1.5
Zombie Detection Range (m)	3
Object Collection Range (m)	4
Projectile Speed (m/s)	5
Projectile Damage	50

- **Zombie Types**: Defines the characteristics of different zombie variants, including their detection capabilities, attack mechanics, movement speed, and sensory parameters.
- **NavMesh Agent Default Settings**: Standard settings applied to all NavMesh agents, both AI agents and zombies, governing their navigation and obstacle avoidance behaviors.
- **AI Agent Settings**: Specific configurations for the AI agent responsible for simulating human-like exploration and interaction within the game levels, including health, combat capabilities, and interaction ranges.

These parameter configurations were meticulously selected to ensure that the AI agent behaves in a manner consistent with human players, effectively navigating the environment, interacting with elements, and responding to enemies. The diversity in zombie types introduces varied challenges, requiring the AI agent to adapt its strategies accordingly. The NavMesh settings facilitate realistic movement and obstacle avoidance, while the AI agent settings balance survivability with effectiveness in achieving objectives.