Generative Datalog for Procedural Content Generation in Video Games

Mario Alviano*, Pasquale Tudda
Department of Mathematics and Informatics, University of Calabria, Via Bucci 30/B, 87036 Rende (CS), Italy.

* Corresponding author. Tel.: +39 0984 496473; email: mario.alviano@unical.it (M.A.)
Manuscript submitted December 12, 2024; revised December 25, 2024; December 31, 2024; published March 12, 2025.
doi: 10.18178/JAAI.2025.3.1.90-108

Abstract: Procedural Content Generation (PCG) is a cornerstone of modern game development, enabling the automatic creation of game levels, characters, and narratives. This paper presents a novel methodology for PCG using Generative Datalog (GDatalog), a rule-based language extended with probabilistic capabilities. By treating the game state and previously generated elements as logical facts and representing generation rules as probabilistic GDatalog programs, we provide a declarative framework for content generation. Our approach iteratively maps the evolving game state to new game elements, ensuring both variability and adherence to gameplay constraints. The methodology is demonstrated through examples, highlighting its ability to produce diverse, context-sensitive content while maintaining logical consistency. This work lays the groundwork for structured, rule-driven PCG pipelines that leverage logical inference and probabilistic reasoning to enrich player experiences.

Keywords: Procedural content generation, probabilistic logic programming, answer set programming, declarative programming

1. Introduction

Procedural Content Generation (PCG) has become an integral part of modern game development, enabling developers to create diverse and engaging content efficiently. From generating randomized dungeon layouts to crafting adaptive narratives, PCG offers a flexible approach to content creation that enhances replayability and reduces manual design effort. However, achieving both creativity and control in PCG remains a challenging problem, particularly when balancing randomness with adherence to gameplay constraints. Recent advances in AI-driven methods, such as the integration of reinforcement learning and generative models like Generative Adversarial Networks (GANs), have pushed the boundaries of content generation in games (e.g., PCGPT for iterative content creation) [1, 2].

To address this challenge, we explore the use of Generative Datalog (GDatalog) [3–6] as a framework for PCG. GDatalog is an extension of the Datalog language, incorporating probabilistic reasoning into its declarative, rule-based structure. This allows developers to encode game mechanics, constraints, and probabilistic generation processes in a clear and modular manner. By treating the game state and previously generated elements as facts, and leveraging GDatalog rules to produce new content, our approach ensures that generated outputs are both diverse and logically consistent. This approach aligns with ongoing research in PCG, which has been conceptualized as influencing key aspects of game mechanics, dynamics, and player aesthetics [7].

In this paper, we propose a methodology for PCG that leverages the power of GDatalog. The central idea is to model the probabilistic functions driving content generation as GDatalog programs. These programs define how elements of the game world should be generated based on the evolving state of the game. The game state and previously generated elements are represented as input facts in GDatalog, which may include positions of entities, current player status, and other relevant conditions within the game world. These facts serve as the foundational input from which new content is derived. The generated elements, such as newly spawned objects, enemies, or environmental features, are produced by applying rules encoded in GDatalog to the existing facts. The output of this process is the answer set, a collection of newly derived facts that represents the updated game world. These facts include both the newly generated elements and any adjustments to existing elements based on the probabilistic generation rules.

This approach introduces a structured and iterative process to PCG, where each step generates content based on the most current game state and prior elements. The rules can adapt to dynamic changes in the game, making it well-suited for real-time, evolving environments where the game state continuously influences the content generation process. As each new answer set is generated, the game world is updated, ensuring that the content produced remains relevant and contextually appropriate for the player's actions and the overall game progression. Moreover, by incorporating probabilistic rules, we introduce variability into the content generation process, which enhances replayability and provides players with unique experiences in each playthrough. This allows for greater control over content diversity while ensuring that generated elements maintain logical consistency and coherence with the rest of the game world. The declarative nature of GDatalog ensures that these rules are easily understandable, reusable, and modifiable, making it a powerful tool for developers aiming to create procedurally generated content that adheres to both creative and gameplay constraints.

By utilizing GDatalog's logical inference capabilities and probabilistic modeling, our methodology draws from established concepts in procedural content generation while introducing a flexible, scalable framework that can handle complex game worlds and dynamic content generation needs. This method also builds upon prior work in logic programming for PCG, where the use of formal languages like Datalog has been explored for defining generation rules and ensuring content consistency across varied game states [8–11].

The rest of this paper is organized as follows: we first provide the required background about GDatalog and PCG. Next, we describe the methodology for content generation using GDatalog, including its representation of game state, definition of probabilistic rules, and computation of answer sets. We then apply the methodology to the Godot engine through a case study, highlighting its effectiveness in generating diverse and constrained content. Finally, we conclude by discussing the implications of this approach for game development and potential directions for future work.

2. Background

2.1. Generative Datalog

Generative Datalog (GDatalog) is a probabilistic extension of the traditional Datalog declarative language, aimed at modeling uncertainty and randomness in relational databases. This framework allows for the specification of probabilistic models through a declarative syntax that maintains the key properties of Datalog while incorporating stochastic behaviors such as random sampling and probabilistic choices [3] by leveraging Δ-terms, primitive constructs for representing parametrized probability distributions [5].

GDatalog programs consist of two main components:

• *Generative Component*: This allows for sampling from discrete probability distributions, where the output is a probability space over the minimal models derived from the input database and the program itself.

• *Constraint Component*: This imposes logical constraints that must be satisfied by the possible outcomes, effectively conditioning the generated distributions based on these constraints.

The generative component enables GDatalog to define a rich family of probabilistic models while retaining a declarative nature, meaning that the execution order of rules does not affect the final outcome. In this work, we focus on the generative component of GDatalog.

The linguistic extension introduced by GDatalog, i.e., Δ−terms, are expressions used to represent parameterized numerical probability distributions. Formally, a Δ−term has the form

$$\delta\langle\overline{p}\rangle[\overline{q}]$$

where *p* is a is a non-empty tuple of terms representing distribution parameters and *q* is an optional tuple representing an event signature. The event signature serves to differentiate between different contexts or scenarios under which the probability distribution $\delta\langle\overline{p}\rangle$ is applied [3].

Given the applied nature of this article, hereinafter we opt for the syntax used in the currently available implementation (https://github.com/alviano/gdatalog), that is,

@delta(distribution_name(params), signature)

where *distribution_name* is the selected δ, *params* is the list of parameters, and *signature* is the optimal event signature.

Example 1. Let us consider a scenario in which a coin is flipped ten times, and one more time in case there is the need for a tie-breaking. We are interested in the most frequent outcome. The following GDatalog program encodes such a scenario:

```
event(1..10).
result(EventID, @delta(flip(1, 2), EventID)) :- event(EventID).
heads(N) :- N = #count{EventID : result(EventID, 0)}.
tails(N) :- N = #count{EventID : result(EventID, 1)}.
tie_break_result(@delta(flip(1, 2))) :- heads(N), tails(N).
#show.
#show heads : heads(H), tails(T), H > T.
#show tails : heads(H), tails(T), T > H.
#show heads : tie_break_result(0).
#show tails : tie_break_result(1).
```

Above, the first rule produces ten facts of the form *event(i)*, for *i = 1..10*. Each of these facts triggers the sampling of one outcome of the *flip* distribution with parameters (1, 2), that is, 0 (for heads) or 1 (for tails) are sampled with probability 0.5. After that, the number of heads and tails are determined by aggregating the observed outcomes. If a tie-breaking is needed, the coin is tossed one last time and the result is stored in the predicate *tie_break_result*. Finally, the interest for the event “most frequent outcome” is represented by the *#show* directives.

The integration of stable negation into GDatalog [4, 5] enhances its expressiveness by allowing the representation of non-monotonic properties, which traditional GDatalog cannot accommodate due to its inherently monotonic nature. Stable negation is interpreted according to stable model semantics, a concept rooted in logic programming that provides a robust framework.

Example 2. Stable negation is implicitly used in the GDatalog program shown in Example 1. Indeed, the semantics of the #count aggregate can be defined in terms of stable negation.

2.2. Godot Engine

The Godot Engine is a feature-packed, open-source and cross-platform game engine to create 2D and 3D games from a unified interface [12], so that the developers have the possibility to shape their engine to match their expectations retaining the full ownership of their games. Godot supports multiple programming paradigms, with native support for its own high-performance scripting language, GDScript, which is syntactically similar to Python. Additionally, the engine provides bindings for multiple languages including C#, C++, and visual scripting, making it accessible to developers with diverse programming backgrounds.

Unlike many proprietary game engines, Godot is built with a node-based architecture that offers an exceptional modularity and extensibility. A *node* is a single component or object that has a specific purpose, as a matter of example, in Godot there are nodes that represents visual elements, e.g., *AnimatedSprite2D*, nodes that handles physics simulations and interaction *e.g. Collider2D* and *InteractionArea2D*, and many others that varies from *Control* nodes for UI elements to nodes that handles audio sources e.g., *AudioStreamPlayer.* The engine makes use of a *scene* system where every element of the game, from user interfaces and environment to characters, are scenes built as a hierarchical node structure. This design allows for intuitive composition and rapid prototyping, enabling developers to create complex and modular scenes.

We illustrate the main elements involved in the setup of Godot scenes by means of an example.

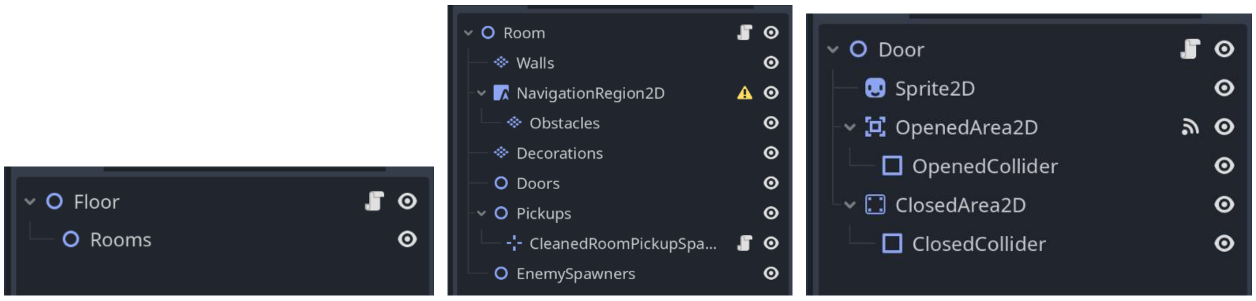

Fig. 1. Node hierarchy of floor, room and door scene.

Example 3. Fig. 1 shows an example of scenes in Godot that can be used for facilitating dynamic content generation. Note that the three roots, namely Floor, Room and Door, occur also as inner nodes of other trees. This approach leverages a decoupled way of bounding together related elements of the game, as for example rooms and doors. A *Door* is unaware of the existence of the *Room* in which it is contained, and similarly a *Room* is unaware of the *Floor* to which it is part of.

Focusing on the *Floor* scene, we observe that the root is a *Node2D* with a script attached. The script contains the code related to room management, and the rooms themselves are contained in the leaf *Node2D* node.

Regarding the *Room* scene, it features various node types:

- *Walls, Obstacles* and *Decorations:* three *TileSetLayer* nodes comprising the tiles regarding the aesthetics elements of rooms.
- *NavigationRegion2D:* a node defining the boundaries of the room on which nodes of type *NavigationAgent2D* (the enemy character) can walk on.
- *Doors:* a simple *Node2D* that contains the *Door* nodes of the room.
- *Pickups:* a *Node2D* containing spawner nodes that introduce new collectible elements in the room.
- *EnemySpawners:* a *Node2D* like the previous node but spawning enemies.

The *Door* scene includes:

- *Sprite2D:* a node used for the visual representation of the door.
- *Collision* and *InteractionArea* nodes, essential for handling the interaction of and the collisions with the player.

3. Methodology for Procedural Content Generation with GDatalog

Procedural Content Generation (PCG) refers to the automated creation of game content through algorithmic processes, significantly enhancing both the efficiency and variety of content in video games. PCG has emerged as a critical area of research and application within game development. This approach leverages various methodologies from fields such as artificial intelligence, computational intelligence, computer graphics, and discrete mathematics to generate a wide array of game elements, including levels, characters, quests, and environments. Using a declarative approach such as Generative Datalog can provide a powerful and flexible framework for PCG because it leverages the logical nature of Generative Datalog to provide a simple way to specify high level constraints and rules.

To utilize GDatalog for PCG in game development, we can define a structured approach that leverages its probabilistic and declarative capabilities. This approach involves the integration of generative components and constraints to create diverse and dynamic game content. In particular, the proposed methodology involves the following steps:

1. representing the game state and generated elements as facts
2. defining probabilistic generation rules
3. generating the answer set and interpret it as new generated elements

3.1. Representing the Game State and Generated Elements as Facts

The procedural generation process starts by encoding the current state of the game and previously generated elements as facts in GDatalog. This representation serves as the knowledge base for content generation. Facts are atomic and declarative, making them easily interpretable and adaptable.

- Game State Facts: These describe the static and dynamic aspects of the game, such as:
    - Spatial configuration: `level_size(10, 10), player_position(5, 5)`.
    - Resource constraints: `available_resources(health_potion, 3)`.
    - Gameplay mechanics: `difficulty(high)` or `weather(rainy)`.
- Generated Element Facts: These represent elements created in previous steps. For example:
    - Placed enemies: `enemy(goblin, (3, 3)).`
    - Items: `item(health_potion, (2, 6)).`

By organizing information as facts, the game environment and its evolving state become explicitly defined, enabling robust, rule-based reasoning in subsequent steps.

3.2. Defining Probabilistic Generation Rules

The core of the PCG methodology lies in defining probabilistic generation rules in GDatalog. These rules map the current state of the game and previously generated elements to new content, incorporating constraints and randomness.

A strength of using GDatalog rules is their declarative nature, which eases the description of relationships and dependencies among elements of the game. Moreover, GDatalog rules can ensure that generated content adheres to logical and gameplay-specific requirements. The use of Δ−terms introduces variability and ensure diverse outcomes.

Example 4. Consider the following GDatalog program:

```
% Spawn enemies with weighted probabilities based on difficulty
enemy(Type, Location) :- valid_spawn_location(Location), difficulty(high),
  Type = @delta((orc(7), troll(3)), Location).
```

Here, the rule generates enemies with a 70% chance of being an orc and 30% chance of being a troll at valid locations. Similarly, the following GDatalog program distribute items according to a probability distribution:

```
% Generate health potions with a probability of 50% for any unoccupied location
item(health_potion, Location) :- valid_item_location(Location),
  @delta((health_potion(1), none(1), Location)) == health_potion.
```

In the above programs, the valid locations can be determined by additional rules. For example, the following two rules:

```
valid_spawn_location(Location) :-
  possible_spawn_location(Location), not occupied(Location).
valid_item_location(Location) :-
  possible_item_location(Location), not occupied(Location).
```

This approach simplifies complex generation problems by breaking them down into logical rules, enabling reusable and composable content generation.

3.3. Generating the Answer Set and Interpret It as New Generated Elements

The final step is computing the answer set using the GDatalog engine. This process involves applying the defined rules to the input facts, generating a new set of facts that represent the created content.

Example 5. Let us consider the following GDatalog program, including facts:

```
player_position((5, 5)). % coordinates in a 2D game
level_size(10, 10).     % 10x10 grid
difficulty(high).       % no fear no more!

possible_spawn_location((3, 3)).
possible_spawn_location((8, 7)).
possible_spawn_location((3, 9)).

possible_item_location((1, 1)).
possible_item_location((10, 10)).

% a goblin was previously generated at location (3, 3)
previous_element(enemy, goblin, (3, 3)).


% Generate new elements
enemy(Type, Location) :- valid_spawn_location(Location), difficulty(high),
  Type = @delta((orc(7), troll(3)), Location).
item(health_potion, Location) :- valid_item_location(Location),
  @delta((health_potion(1), none(1), Location)) == health_potion.

% aux rules
occupied(Location) :- player_position(Location).
occupied(Location) :- previous_element(_, _, Location).

valid_spawn_location(Location) :-
  possible_spawn_location(Location), not occupied(Location).
valid_item_location(Location) :-
  possible_item_location(Location), not occupied(Location).

#show.
#show enemy/2.
#show item/2.
```

A possible outcome is the generation of the facts

```
    enemy(orc, (3, 9))
    enemy(troll, (8, 7))
```

That is, one orc and one troll must be spawned at locations (3, 9) and (8, 7), respectively, and no item must be dropped. Such an outcome has probability close to 13%. These elements are then incorporated into the knowledge base for subsequent iterations of generation.

4. Application to Godot

This section reports on the application of PCG using GDatalog in the development of a top-down, 2D, roguelike game written in Godot (https://github.com/ryuk4real/the-binding-of-UNICAL). The game itself is written using the Godot native GDScript programming language, and GDatalog is used as an external service. The roguelike genre is characterized by levels generated procedurally that leverages replayability and provide players with a unique experience on each session. The game uses PCG techniques to dynamically create various elements such as rooms, enemies and collectibles. Also, the game is set within the University of Calabria so that the player can navigate various locations that aesthetically mirror the university's indoor environments, including hallways, classrooms, laboratories and restrooms. Within each of these rooms, players encounter enemies that actively attempt to engage them in combat (see Fig. 2). The player is equipped with the ability to shoot projectiles to defeat these adversaries. If the player clears a room, meaning that the player defeats all the enemies present in the current room, the player might find a reward that can be an item that restores health or a power-up. The player can also use the stairs located in the hallway, at the starting point of each floor, to access the next floor.

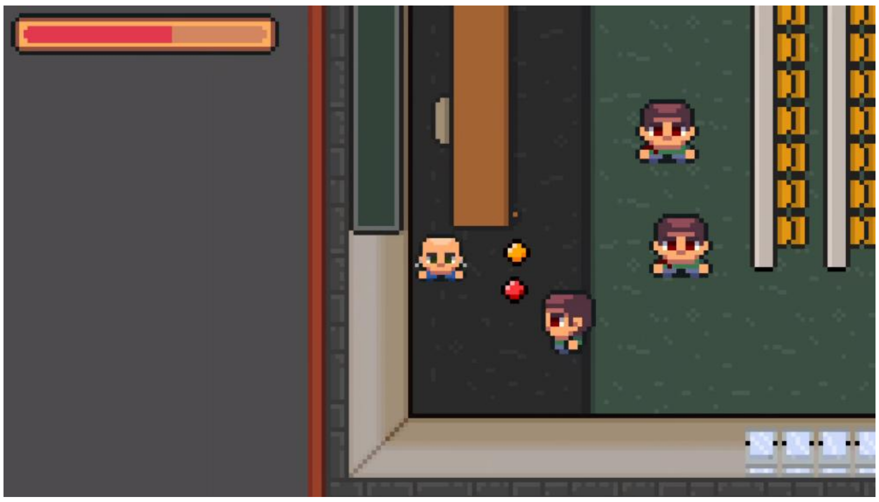
Fig. 2. The player surrounded by enemies inside a classroom.

4.1. Establishing a Connection between Godot and GDatalog

The Godot game engine and the GDatalog implementation use different languages and different paradigms so that a non-trivial issue occurred during the development of the game. This issue was making the two systems communicate in a simple and straightforward way. To solve this problem, we added a server to the Python implementation of GDatalog so to enable HTTP communication using JSON notation. The server is implemented using FastAPI, which is a modern, fast (high-performance), web framework for building APIs with Python [13].

The server contains a POST method that receives a request in JSON format. The request contains:

- *program*: the GDatalog program;
- *max_stable_models*: a field that limits the number of stable models that results from the execution of the program.

The response contains:

- *model*: a field which is a list of the stable models that results from the execution of the GDatalog program;
- *state*: a field that specifies the state of the execution of the GDatalog program;
- *delta_terms*: a field which is the list of the calculation of the Δ−terms that results in the outcome probabilities of the program.

The following block of code presents the code of the POST method of the server (simplified to ease the presentation):

```
@app.post("/run/")
async def _(request: Request):
  json = await request.json()
  try:
    max_stable_models = \
      int(json["max_stable_models"]) if "max_stable_models" in json else 1
    program = Program(json["program"], max_stable_models=max_stable_models)
    sms = program.sms()
    return {
      "state": str(sms.state),
      "models": [
        [atom_to_json(atom) for atom in model] for model in sms.models],
      "delta_terms": [str(delta_term) for delta_term in sms.delta_terms],
    }
    except Exception as e:
        return {"error": str(e)}
```

To run the server from the command line, the *server* keyword must be specified, as shown next:

```
$ poetry run python gdatalog/gdatalog_cli.py -f [filename] server
```

Once the server is running, Godot can establish a connection by using a *Worker* comprising a HTTPRequest node, which is run when the game starts. The following code shows how a program can be exchanged for one of its probabilistic answer sets (or stable models) using the Worker and the GDScript language:

```
func _get_answerset_from_worker(_program: String) -> Array:
  worker.post(_program)
  await SignalBus.response_ready
  var response = worker.response.get("models")
  return response
```

4.2. Floor Generation

The implementation of the floor generation process is carried out by the *FloorGenerator* node, which is embedded in the main scene's hierarchy. Such a node employs the *Worker* introduced in the previous section to communicate with the GDatalog server in order to produce the elements of the generated floor. In particular, the process of generating the *Floor* starts with the instantiation of a *Floor* node in the hierarchy of the main scene. After that, the initial room, which is by default of type *HALLWAY,* is added to the *Floor* instance. Subsequently, the newly instantiated room is added to a queue of rooms that are prepared for processing. For each room within such a queue, as well as for each door associated with these rooms, a request is submitted to the GDatalog server. This request aims to ascertain the potential type of door, and consequently the type of the room that could be appropriately positioned adjacent to the current door. The program of the request is determined by the type of the current processing room, since there are constraints on which type of neighbor a room can have. The *_get_room_neighbor_type* function (shown below) requires as argument the type of the room under processing and the facts that refers to the current floor. These atoms are generated as the rooms are placed. The GDatalog programs needed in these methods are directly read from the file when the game starts. The following is the encoding of the state of the *Floor* that is appended to the GDatalog program.

```
room(ID).  current_room(ROOM_ID).  neighbours(ROOM_ID1, ROOM_ID2).
```

The following code is precisely how the type of the neighbor to place is requested based on the type of the current room in process:

```
func _get_room_neighbour_type(
    room_type: int, _floor_atoms: Array[String] = []) -> int:
  var type_answer_set: Array
  var program: String

  for atom: String in _floor_atoms:
    program += atom

  program += Global.current_floor.current_room_atom

  match(room_type):
    Global.ROOM_TYPE_HALLWAY:
      program += hallway_neighbour_type_guesser_program
      type_answer_set = await _get_answerset_from_worker(program)
    Global.ROOM_TYPE_INNER_HALLWAY:
      program += inner_hallway_neighbour_type_guesser_program
      type_answer_set = await _get_answerset_from_worker(program)
    Global.ROOM_TYPE_CLASSROOM:
      program += inner_hallway_neighbour_type_guesser_program
      type_answer_set = await _get_answerset_from_worker(program)

    Global.ROOM_TYPE_LIBRARY:
      program += library_neighbour_type_guesser_program
      type_answer_set = await _get_answerset_from_worker(program)
  return type_answer_set[0][0].get("arguments")[0].get("number")
```

The following GDatalog program encapsulates the logic and rules of the room generation process passing a distribution as argument to the Δ−term and it gives as result a stable model that contains an atom that encodes the type of the room that will be placed. The GDatalog program is responsible for deriving the type of possible neighbours of an INNER_HALLWAY room, but the same reasoning is used on the program of the other types of room.

```
  #const none = 999.
  #const inner_hallway = 1.
  #const classroom = 3.
  #const office = 4.
  #const storage = 5.
  #const library = 6.

  type(classroom).  type(office).  type(storage).  type(library).

  % Generate neighbours both ways
  neighbours(ROOM_ID1, ROOM_ID2) :- neighbours(ROOM_ID2, ROOM_ID1).

  % Count all neighbours of current room for each room type
  room_type_counter(TYPE, COUNT):- current_room(ROOOM_ID, TYPE2), type(TYPE),
      COUNT = #count { NEIGHBOUR_ROOM_ID, TYPE :
          neighbours(ROOM_ID, NEIGHBOUR_ROOM_ID),
          room(NEIGHBOUR_ROOM_ID, TYPE)
      }.

  % If current room has no neighbours of a certain type, the count is 0
  room_type_counter(TYPE, COUNT):- COUNT = 0, type(TYPE),
      current_room(ROOM_ID1, _), room(ROOM_ID2, TYPE),
      not neighbours(ROOM_ID1, ROOM_ID2).

  % If room types have been counted weights are calculated
  room_weights(W2, W3, W4, W5) :-
      room_type_counter(classroom, W2), room_type_counter(office, W3),
      room_type_counter(storage, W4),   room_type_counter(library, W5).

  % If room types have been counted a weighted distribution is applied where the
probability of having a room of a certain type as a neighbour is increased by
the number of rooms of that type already counted
  inner_hallway_neighbour_type(X) :- room_weights(W2, W3, W4, W5),
      X = @delta((
          (none, 1),
          (inner_hallway, 1),
          (classroom, 3 + W2),
          (office, 3 + W3),
          (storage, 2 + W4),
          (library, 2 + W5))
      ).
  #show inner_hallway_neighbour_type/1.
```

The code is properly commented to facilitate the understanding of the meaning of each rule but the key point is that the distribution that describes the probability mass function of the type of the rooms that can be placed is updated dynamically during the generation of the rooms updating the weights according to the number of neighbours of the same type already placed.

An opposite approach is used for deriving the type of HALLWAYS. In that case the weights are swapped so that if a particular type of room has already been placed the probability of having a room of a different type increases by the number of rooms of that type that has been placed. To avoid redundancy the following piece of code shows just the Δ-term and the associated distribution and weights.

```
    room_weights(W1, W2) :-
        room_type_counter(inner_hallway, W1),
        room_type_counter(bathroom, W2).

    hallway_neighbour_type(X) :- room_weights(W1, W2),
        X = @delta((
                (none,2),
                (inner_hallway,2 + W2),
                (bathroom,1 + W1)
            )
        ).
    #show hallway_neighbour_type/1.
```

Once the type of the room to be placed is determined, the algorithm proceeds to randomly select a room from a pool of preloaded rooms that corresponds to the direction of the current door. The algorithm then attempts to position the selected room adjacent to that door. To assess whether the room can be placed, a support matrix is employed. This matrix is incorporated in the Floor scene through code initialized at the beginning of the generation with a default value of −1. During the generation, if a tile of the room can be placed, meaning that there is sufficient available space, the values at the positions in the matrix are updated from −1 to the *id* of the newly placed room.

When evaluating whether the room can be positioned in a given location, it is ensured that no other rooms overlap at those positions. If these conditions are satisfied, the room is successfully placed, and a map of connections within the room is updated. The map contained in the *Floor* scene contains information about the doors in a way to facilitate the connectivity between rooms. If a *Room* cannot be placed or it has as type none, the current door in process is set as placeholder and locked, so that it is not visible in the game and the player cannot interact with it.

4.3. Enemies' Generation

A similar approach—as the one presented in the previous section—is employed for the generation of enemies within each room. Specifically, for each EnemySpawner node in the room, a request is made to the GDatalog server to determine the type of enemy to spawn. Then an instance of the enemy is selected from a preloaded pool of enemy scenes. To facilitate this process, information regarding the enemies spawned in the current room is encoded into Datalog atoms and appended to the GDatalog program.

```
current_room(ROOM_ID, TYPE)
enemy(ROOM_ID, ENEMY_ID, ENEMY_TYPE)
```

The weights of the distribution are calculated by counting the number of enemies by type, to increase on each call the probability of spawning the less frequent enemy. To avoid redundancy part of the code is intentionally omitted.

```
enemy_weights(W1,W2) :- enemy_counter(zombie,W1), enemy_counter(student,W2).

% If enemy types have been counted, I apply a weighted distribution where the probability of having an enemy of a certain type is increased the less enemies of that type have been counted.
enemy_type(X) :- enemy_weights(W1, W2),
    X = @delta((
        (none, 1),
        (zombie, 1 + W2),
        (student, 1 + W1))
    ).
#show enemy_type/1.
```

4.4. Collectibles Generation

Collectibles are not generated during the floor generation algorithm, but they are generated each time a room is cleared from enemies. Upon the defeat of the final enemy of the room, a request is made to the GDatalog server, which includes the GDatalog program of the collectible's generation, along with the Datalog atoms representing the player's state. The player's state contains parameters such as health points (hp), movement speed, damage output, and the speed and rate of fire of projectiles.

```
player_stat(damage, 100).
player_stat(max_damage, 100).
player_stat(shot_speed, 200).
player_stat(max_shot_speed, 200).
player_stat(shot_rate, 5).
player_stat(min_shot_rate, 1).
player_stat(speed, 100).
player_stat(max_speed, 150).
player_stat(current_hp, 51).
player_stat(max_hp, 100).
```

To derive the weights that regulate the collectibles distribution the percentage of each stat is calculated based on its maximum (or minimum).

```
stat_percentage(hp, PERCENTAGE) :-
    player_stat(current_hp, CURRENT_HP),
    player_stat(max_hp, MAX_HP),
        PERCENTAGE = (CURRENT_HP * 100) / MAX_HP.

    stat_percentage(speed, PERCENTAGE) :-
        player_stat(speed, SPEED),
        player_stat(max_speed, MAX_SPEED),
        PERCENTAGE = (SPEED * 100) / MAX_SPEED.

    stat_percentage(shot_speed, PERCENTAGE) :-
        player_stat(shot_speed, SHOT_SPEED),
        player_stat(max_shot_speed, MAX_SHOT_SPEED),
        PERCENTAGE = (SHOT_SPEED * 100) / MAX_SHOT_SPEED.

    stat_percentage(shot_rate, PERCENTAGE) :-
        player_stat(shot_rate, SHOT_RATE),
        player_stat(min_shot_rate, MIN_SHOT_RATE),
        PERCENTAGE = 100 - ((SHOT_RATE - MIN_SHOT_RATE) * 100) / (10 -
  MIN_SHOT_RATE).

    stat_percentage(damage, PERCENTAGE) :-
        player_stat(damage, DAMAGE),
        player_stat(max_damage, MAX_DAMAGE),
        PERCENTAGE = (DAMAGE * 100) / MAX_DAMAGE.
```

The weights of the distribution of the collectibles are dependent on the player's statistics. Specifically, the closer a stat is to its maximum (or minimum) value, the higher the probability of generating collectibles that increase other stats. This dynamic ensures that the player finds more often collectibles that improves their weaker stats. To better regulate the distribution, it is applied a multiplier to the collectibles weights that increases the health of the player (medikit and bandages).

```
    item_weights(MEDIKIT_BONUS, BANDAGES_BONUS, SPEED_BONUS * multiplier,
  SHOT_SPEED_BONUS * multiplier, SHOT_RATE_BONUS * multiplier, DAMAGE_BONUS *
  multiplier) :-
        stat_percentage(hp, HEALTH_PERCENTAGE),
        stat_percentage(speed, SPEED_PERCENTAGE),
        stat_percentage(shot_speed, SHOT_SPEED_PERCENTAGE),
        stat_percentage(shot_rate, SHOT_RATE_PERCENTAGE),
        stat_percentage(damage, DAMAGE_PERCENTAGE),

        MEDIKIT_BONUS = #count { 1 :
            HEALTH_PERCENTAGE <= 25
        },

        BANDAGES_BONUS = #count { 1 :
            HEALTH_PERCENTAGE <= 50
        },
        SPEED_BONUS = #count { 1 :
            SHOT_SPEED_PERCENTAGE > 50,
            SHOT_RATE_PERCENTAGE > 50,
            DAMAGE_PERCENTAGE > 50,
            HEALTH_PERCENTAGE > 50
        },

        SHOT_SPEED_BONUS = #count { 1 :
            SPEED_PERCENTAGE > 50,
            SHOT_RATE_PERCENTAGE > 50,
            DAMAGE_PERCENTAGE > 50,
            HEALTH_PERCENTAGE > 50
        },

        SHOT_RATE_BONUS = #count { 1 :
            SPEED_PERCENTAGE > 50,
            SHOT_SPEED_PERCENTAGE > 50,
            DAMAGE_PERCENTAGE > 50,
            HEALTH_PERCENTAGE > 50
        },

        DAMAGE_BONUS = #count { 1 :
            SPEED_PERCENTAGE > 50,
            SHOT_SPEED_PERCENTAGE > 50,
            SHOT_RATE_PERCENTAGE > 50,
            HEALTH_PERCENTAGE > 50
        }.

    % Default distribution if collectible types have not been counted yet
    collectible_type(X) :- not item_weights(_, _, _, _, _, _),
        X = @delta((
            (none, 4),
            (medikit, 2),
            (bandages, 3),
            (speed_up, 1),
            (shot_speed_up, 1),
            (shot_rate_up, 1),
            (damage_up, 1)
        )).

    % If collectible types have been counted, I apply a weighted distribution
  where the probability of having a collectible of a certain type increases where
  the corresponding stat is lower
```

```
collectible_type(X) :- item_weights(W1, W2, W3, W4, W5, W6),
    X = @delta((
        (none, 4),
        (medikit, 2 + W1),
        (bandages, 3 + W2),
        (speed_up, 1 + W3),
        (shot_speed_up, 1 + W4),
        (shot_rate_up, 1 + W5),
        (damage_up, 1 + W6)
    )).
#show collectible_type/1.
```

5. Evaluation

We compare our proposed approach to the previously proposed methodology presented in Ref. [8], focusing on computational efficiency and flexibility in the generation process. Without going into much detail, [8] relies on a two-step process: first, it enumerates all possible floor layouts within specified bounds, considering parameters such as floor size and room limits; after enumeration, it randomly selects one layout, discarding all other computed options. This leads to significant computational waste since only one floor is retained while all others are discarded. Furthermore, the method assigns equal probability to all layouts, offering limited control over generation diversity or adherence to specific design constraints. In contrast, our method generates floor elements incrementally. Using a probability distribution, our approach dynamically constructs floors step by step, adapting to constraints and ensuring that all computed elements contribute directly to the final result. This way we avoid waste of computation and provide finer control over the generation process, enabling designers to encode nuanced rules and distributions to guide the content creation.

To evaluate these two different approaches, we conducted experiments generating 24×24 floors with up to 15 rooms. For Ref. [8], we tested with various bounds on the enumeration limit, measuring execution time for each configuration. Results are shown in Table 1, where it can be observed that the first 100 answer sets are produced within roughly the same amount of time. This is due to the eager approach implemented by [8] Seta *et al.* [8], which needs to address the full generation process within a single call to the answer set solver. The answer set solver must then face a nonnegligible grounding task before starting the enumeration of answer sets. Regarding our method, we repeated the generation process multiple times to account for variability and averaged the execution time. Results of 25 runs are shown in Table 2, where we report the number of calls to the server (the GDatalog engine), the cumulated execution time, the number of generated rooms, and the probability associated with the generated floor.

Table 1. Empirical Evaluation of the PCG Approach Presented in [8]

Limit on the number of answer sets	Execution time (s)
1	21.2
10	21.3
100	21.5
1000	66.2
10000	133.5

Table 2. Empirical Evaluation of the Proposed PCG Approach

Run	Number of computed answer sets (calls to the server)	Execution time (seconds)	Number of generated rooms	Probability of the generated floor
1	83	6.9	13	4.82E-45
2	86	7.1	12	1.47E-48
3	70	5.9	10	3.28E-37
4	112	9.3	15	2.32E-61
5	38	3.2	8	2.43E-20
6	93	7.9	14	2.39E-51
7	55	4.6	8	1.32E-29
8	106	8.8	15	1.00E-58
9	103	8.6	14	7.60E-58
10	51	4.2	11	4.76E-28
11	107	8.9	13	1.00E-58
12	62	5.2	9	1.29E-32
13	107	8.9	13	6.00E-58
14	77	6.4	13	5.56E-43
15	74	6.2	10	1.01E-39
16	42	3.5	9	1.26E-22
17	62	5.2	8	5.44E-32
18	45	3.7	8	3.66E-26
19	114	9.5	14	1.63E-61
20	108	9.1	13	5.69E-57
21	90	7.5	12	8.71E-49
22	65	5.4	12	2.05E-38
23	55	4.6	7	8.81E-30
24	107	8.9	15	1.42E-57
25	77	6.4	12	8.89E-41
Average	79.56	6.632	11.52	9.77E-22

The results shown in Tables 1 and 2 demonstrate that our approach is significantly more efficient than Ref. [8], even under stringent enumeration bounds. Indeed, when the enumeration bound is low, Seta *et al.* [8] suffers from a lack of diversity in the generated layouts, while higher bounds introduce impractical execution times due to the exponential growth in enumeration complexity. For instance, to match the diversity achieved by our approach, Seta *et al.* [8] would require enumeration bounds of 10^{22}, which is computationally infeasible. Anyhow, even limiting the enumeration process to a single answer set, the approach presented in Ref. [8] takes more than two times the execution time needed by our approach. Moreover, our approach provides flexibility in controlling generation probabilities and constraints, allowing the creation of diverse and customized layouts within a practical time frame. These advantages make our methodology better suited for large-scale and adaptive procedural content generation scenarios, where efficiency and control are paramount.

6. Related Work

Procedural Content Generation (PCG) has been extensively studied in the literature, with various methodologies proposed to balance creativity, control, and efficiency. We refer Li [14] to a recent overview of existing research that explores the integration of Artificial Intelligence (AI) into video game development processes, and focus here on the declarative approaches that have garnered significant attention due to their ability to clearly encode constraints and logical rules.

Answer Set Programming (ASP) was already used for generating hierarchical structures in roguelike games [7], highlighting the potential of ASP to manage complex game environments effectively, particularly for scenarios requiring hierarchical or multistage structures. This was further expanded in a more recent study [15], which focused on generating dungeons with correct lock-and-key structures. The study proposed a controllable approach for building graph-based models of acyclic dungeon levels, employing declarative constraint solving to ensure logical consistency and adherence to gameplay requirements.

ASP was also combined with the Godot game engine [8], in a similar setting to the one considered in this work. The main difference with respect to the presented work is the way answer sets are produced. In Ref. [8], several answer sets are non-deterministically computed, each one representing the layout of an entire level of the game, and one answer set is randomly selected to be shown to the player. Here, instead, the level is produced according to some probabilistic distributions, and some elements of the game (the collectibles) are lazily generated. The proposed methodology is therefore more efficient and does not waste computation, as exactly one answer set representing the level to be shown to the player is generated. It is thanks to such a different approach that lazy generation is possible without severely impacting on the user experience (for otherwise, the user would observe a significant loading time at each generation step.

Other relevant works in the literature addressing PCG through different logic-based approaches include [16–18]. In Ref. [16], the authors introduce a probabilistic logic framework to unify data-driven and rule-based PCG methodologies. Their work highlights the benefits of Bayesian reasoning to manage randomness and enforce diversity, offering a structured way to generate content that aligns with designer intent while adapting to uncertainty. Regarding Ref. [17], it surveys PCG techniques driven by evolutionary algorithms and metaheuristic search, demonstrating the adaptability of search-based approaches to generate levels, maps, and rulesets that meet specific quality criteria. As for Ref. [18], the authors explore the use of ASP as a declarative tool for defining generative spaces explicitly. ASP enables designers to iteratively refine content generation by encoding constraints and rules directly, fostering greater control over the design space without extensive procedural rework.

7. Conclusions

This paper introduces a novel methodology for Procedural Content Generation (PCG) using Generative Datalog (GDatalog), a rule-based logic programming framework enhanced with probabilistic reasoning. By treating the game state and previously generated elements as logical facts and encoding probabilistic rules as GDatalog programs, this approach provides a structured and iterative process for content generation. The resulting framework combines variability, consistency, and adaptability, making it well-suited for dynamic and evolving game environments. Through the examples and case studies presented, we demonstrate how GDatalog enables developers to produce diverse, context-sensitive content while adhering to gameplay constraints. The declarative nature of GDatalog simplifies the definition and modification of generation rules, offering a modular and reusable structure that can adapt to a wide variety of game genres and design requirements. Future works include the refactoring of GDatalog programs to produce several elements of the games within a single call with the aim of further improving efficiency of the PCG process.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Mario Alviano and Pasquale Tudda conducted the research; Pasquale Tudda implemented the use case application; Pasquale Tudda analyzed the data; Mario Alviano and Pasquale Tudda wrote the paper; all authors had approved the final version.

Acknowledgment

This work was partially supported by Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23003420006 under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); and by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

[1] Mohaghegh, S., Dehnavi, M. A. R., Abdollahinejad, G., & Hashemi, M. (2023). PCGPT: Procedural content generation via transformers. arXiv preprint, arXiv:2310.02405.

[2] Hald, A., Hansen, J. S., Kristensen, J., & Burelli, P. (2020). Procedural content generation of puzzle games using conditional generative adversarial networks. *Proceedings of the 15th International Conference on the Foundations of Digital Games* (pp. 1–9).

[3] Barany, V., Cate, B. T., Kimelfeld, B., Olteanu, D., & Vagena, Z. (2014). Declarative statistical modeling with Datalog. arXiv preprint, arXiv:1412.2221.

[4] Alviano, M., Lanzinger, M., Morak, M., & Pieris, A. (2023). Generative datalog with stable negation. *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (pp. 21–32).

[5] Alviano, M., Zamayla, A., *et al.*, (2021). A speech about generative datalog and non-measurable sets. *Proceedings of ICLP Workshops*.

[6] Grohe, M., Kaminski, B. L., Katoen, J.-P., & Lindner, P. (2022). Generative datalog with continuous distributions. *Journal of the ACM*, *69(6)*, 1–52.

[7] Smith, G. (2014). Understanding procedural content generation: a design-centric analysis of the role of PCG in games. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 917–926).

[8] Seta, D., Alviano, M., *et al.* (2022). An application of ASP for Procedural content generation in video games. *Proceedings of CILC* (pp. 134–140).

[9] Angilica, D., Giorgio, G. M. D., Ianni, G., *et al.* (2023). On the impact of sensors update in declarative AI for videogames. *Proceedings of ICLP Workshops*.

[10] Angilica, D., Ianni, G., Pacenza, F., & Zangari, J. (2023). Integrating asp-based incremental reasoning in the videogame development workflow (application paper). *Proceedings of International Symposium on Practical Aspects of Declarative Languages* (pp. 96–106).

[11] Angilica, D., Ianni, G., & Pacenza, F. (November 19–22, 2019). Tight integration of rule-based tools in game development. *Proceedings of AI* IA 2019*—Advances *in Artificial Intelligence: XVIIIth International*

Conference of the Italian Association for Artificial Intelligence, Rende, Italy (pp. 3–17).

[12] Linietsky, J., Manzur, A., & Community, T. G. (2024). Introduction—Godot engine 4.3 documentation in English. Retrieved from https://docs.godotengine.org/en/stable/about/introduction.html

[13] Ramírez, S., *et al.* (2024). FastAPI—Main page. Retrieved from https://fastapi.tiangolo.com/

[14] Li, D. (2024). Artificial Intelligence in the game development process. *Journal of Advances in Artificial Intelligence*.

[15] Smith, T., Padget, J., & Vidler. (2018). A. Graph-based generation of action-adventure dungeon levels using answer set programming. *Proceedings of the 13th International Conference on the Foundations of Digital Games*.

[16] Madkour, A., Martens, C., Holtzen, S., Harteveld, C., & Marsella, S. (2023). Probabilistic Logic programming semantics for procedural content generation. *Proceedings of AIIDE 2023* (pp. 295–305).

[17] Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intell. AI Games*, *3(3)*, 172–186.

[18] Smith, M., & Mateas, M. (2011). Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intell. AI Games, 3(3)*, 187–200.