School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

Style Aware Procedural Level Generation: Influencing Generators Based on Identified Style Features
Research Proposal

Sean Mahlanza, 2438634

May 18, 2025

Supervised by: Pravesh Ranchod and Branden Ingram

Abstract

Procedural Content Generation (PCG) enables automated creation of game levels but often lacks mechanisms for designer-defined stylistic control. This research presents a style-aware search-based PCG framework tailored to 2D platformer game levels. Our approach integrates user-defined structural and visual style profiles into a Feasible–Infeasible Two–Population (FI-2POP) genetic algorithm, guiding level evolution to satisfy both playability constraints and stylistic targets. We extract graph-based structural metrics and tile-level aesthetic signatures from exemplar levels, normalize them into named or custom style features, and incorporate these profiles as adherence terms within a composite fitness function. The implementation leverages industry-standard libraries for data handling, graph processing, evolutionary search, and visualization, along with a custom parser for the Video Game Level Corpus. Evaluation will compare our style-aware generator against random, constructive, and style-agnostic baselines using quantitative metrics for playability, aesthetics, structure, and style adherence. We hypothesize that embedding user-defined style profiles in the evolutionary loop will produce levels that not only maintain high playability but also closely match designer intent, advancing personalized Procedural Content Generation.

Contents

1 Introduction 5

2 Background 6

2.1 Procedural Content Generation (PCG) 6

2.1.1 How Procedural Content Generation Works 6

2.2 Content Generation Techniques 6

2.2.1 Generate-and-Test 7

2.2.2 Search-Based Procedural Content Generation 7

2.2.3 Wave Function Collapse (WFC) 8

2.2.4 Machine Learning-Based Procedural Content Generation (PCGML) 10

3 Related Work 11

3.1 From Rule-Based Systems to Learning-Driven Models 11

3.2 Hybrid Approaches: Bridging Hand-Crafted and Learned Models 11

3.3 Learning to Generate: The Rise of PCGML 11

3.4 Large Language Models in PCG 12

3.5 Deep Learning for Procedural Generation 12

4 Research Methodology 13

4.1 Problem Statement 13

4.2 Research Question 13

4.3 Aim 13

4.4 Hypothesis 13

4.5 Methodology 13

4.5.1 Data Collection 13

4.5.2 Implementing a Search-Based Level Generator 14

4.5.3 Extracting and Defining Structural Style Features 16

4.5.4 Extracting and Defining Visual Aesthetic Themes . . . 17
4.5.5 Incorporating Style Profiles into the Generator . . . 17
4.5.6 Validation and Post-processing . . . 18
4.5.7 Tools and Frameworks (Environment) . . . 18
4.5.8 Evaluation and Analysis . . . 18
4.5.9 Potential Challenges and Difficulties . . . 19

5 Research Plan 20

6 Conclusion 21

1 Introduction

Procedural Content Generation (PCG) techniques automate the creation of game levels and environments by encoding rules and constraints that guarantee functional correctness, such as playability, solvability, and structural coherence, often at the expense of stylistic qualities like visual aesthetics and thematic consistency. [Togelius et al. 2011b]

A promising approach involves explicitly defining stylistic attributes, such as visual themes, structural patterns, or specific gameplay dynamics, and integrating these into PCG frameworks as guiding constraints. Recent studies have explored semantic-driven methods [Silveira et al. 2015], hybrid grammar-cellular automata approaches [Gellel and Sweetser 2020], and machine learning techniques that model style from existing game data [A. Summerville et al. 2018; Todd et al. 2023].

However, embedding such subjective style constraints into a procedural pipeline poses significant challenges. Style elements are often high-dimensional, nuanced, and context-dependent, making their objective quantification difficult. Approaches utilising machine learning and Large Language Models (LLMs) highlight the difficulties of enforcing stylistic constraints due to data dependence and generalisation issues [Kumaran et al. 2023; Todd et al. 2023].

To address these challenges, this research proposes a style-aware, search-based procedural level generation framework that integrates user-defined structural and visual styles directly into a genetic algorithm. , allowing simultaneous optimisation of playability and style adherence.

The proposal is organised as follows: Background covering the fundamentals of PCG; Related Work surveying existing methods for stylistic procedural generation; Proposed Framework describing our style-aware genetic architecture; Methodology detailing representation, fitness functions, and optimisation procedures; Implementation presenting system details; Evaluation reporting quantitative and qualitative results; and Conclusions summarising the proposal and expected contributions.

2 Background

2.1 Procedural Content Generation (PCG)

Procedural Content Generation (PCG) refers to the algorithmic creation of content with limited or indirect user input. This method is used primarily in video games and digital media to automatically produce large amounts of content such as levels, environments and characters, thereby reducing development costs and enhancing replayability.[Togelius et al. 2011b]

PCG can be deterministic or stochastic, and content generated can be static (fixed once generated) or dynamic (changes in response to player behavior). The roots of PCG trace back to early text-based games like Rogue, which used procedural techniques to create new dungeon layouts with every playthrough. The motivation was both practical (limited memory and storage) and creative (replayability and unpredictability). Over time, as computational power increased, PCG evolved into a central feature of many modern games. Titles like No Man's Sky exemplify the scale and complexity of modern PCG, generating entire universes with billions of planets algorithmically. [Togelius et al. 2011a]

2.1.1 How Procedural Content Generation Works

Procedural Content Generation (PCG) operates through algorithms that autonomously create game content, balancing between control (via constraints and rules) and creativity (through randomness or learning). Although techniques may vary depending on the type of content (e.g., levels, textures, quests), most PCG systems follow a general pipeline:

The generation process begins with **input parameters and constraints**, where structural, stylistic, or gameplay-related settings guide content creation to ensure relevance and playability within the game context. Content is then produced using one of several techniques: *constructive algorithms* employ rule-based systems or grammars (e.g. L-systems, cellular automata) for direct generation—efficient and controllable but sometimes limited in variation [Hendrikx et al. 2013]; *generate-and-test* methods randomly create content and iteratively evaluate it against criteria until a valid design emerges [Hendrikx et al. 2013]; *search-based PCG* uses optimisation strategies like genetic algorithms or simulated annealing to evolve content under a fitness function [Togelius et al. 2011a]; *Wave Function Collapse (WFC)* synthesizes levels by matching local patterns from example inputs under constraint propagation [Kim et al. 2019]; and *machine learning-based PCG (PCGML)* learns from existing content via deep learning, Markov models, or reinforcement learning to produce stylistically consistent designs [A. Summerville et al. 2018]. Once generated, content undergoes **evaluation and fitness testing**, which may include rule-based checks (e.g. playability), heuristic assessments (e.g. balance), or adaptive feedback based on player behaviour. Finally, validated content is **integrated into the game** either statically (pre-generated) or dynamically during gameplay.

2.2 Content Generation Techniques

Constructive Algorithms Constructive algorithms generate game content by executing a sequence of deterministic, rule-based steps, which makes them both fast and computationally lightweight—ideal for real-time applications. For instance, cellular automata iteratively apply local birth-and-death rules to evolve dungeon-like or cave systems from an initial random grid, producing organic structures with minimal computation. Tiling systems and shape grammars assemble levels by selecting and arranging predefined tiles or shape primitives according to adjacency and thematic constraints, ensuring coherent layouts without exhaustive search. Binary Space Partitioning (BSP) recursively subdivides the game area along alternating axes, carving rooms and connecting corridors in a top-down fashion. While these methods inherently guarantee valid, playable layouts and excel in efficiency, they can exhibit limited variation and repetitive patterns if not augmented with stochastic choices or combined with subsequent search-based refinement.[Hendrikx et al. 2013].

2.2.1 Generate-and-Test

Generate-and-test approaches introduce randomness into the content generation process by first sampling potential level designs under loose constraints and then rigorously evaluating each candidate against fitness functions or validation criteria, such as playability, balance, diversity, or solvability, discarding any that fail to meet the required thresholds. This generate-evaluate-iterate loop continues until a satisfactory level is found, producing a high degree of variety and surprise compared to purely constructive methods. However, because each iteration involves potentially expensive fitness evaluations, generate-and-test can incur significant computational overhead; it is therefore most effective in domains where evaluation can be efficiently formalized, such as puzzle layout generation or NPC behaviour tree synthesis [Hendrikx et al. 2013].

2.2.2 Search-Based Procedural Content Generation

Search-based procedural content generation (SBPCG) treats level creation as an optimisation problem in a high-dimensional design space, where candidate level layouts are iteratively refined to satisfy multiple evaluation criteria. Rather than applying fixed rules or relying solely on random generation, SBPCG maintains a diverse *population* of candidate solutions and uses feedback from one or more *fitness functions* to guide evolutionary or metaheuristic search operators [Togelius et al. 2011a]. As shown in Figure 1, this approach contrasts with constructive methods, which follow a single deterministic pipeline, and simple generate-and-test loops, by continuously balancing exploration and exploitation across generations.

In SBPCG, the process begins by *initialising* a population of candidate levels, which may be created randomly, seeded from a constructive generator, or drawn from existing corpora. Each candidate is then *evaluated* using one or more fitness metrics, such as playability, difficulty, novelty, or user-defined style adherence—and assigned a fitness score. Based on these evaluations, a *selection* mechanism preferentially chooses higher-scoring individuals to serve as parents, while *variation* operators (e.g. crossover, mutation, simulated annealing moves) introduce controlled perturbations to generate new offspring. This generate-evaluate-select cycle repeats until a termination condition is met, such as a maximum number of generations, a time budget, or convergence of fitness values.

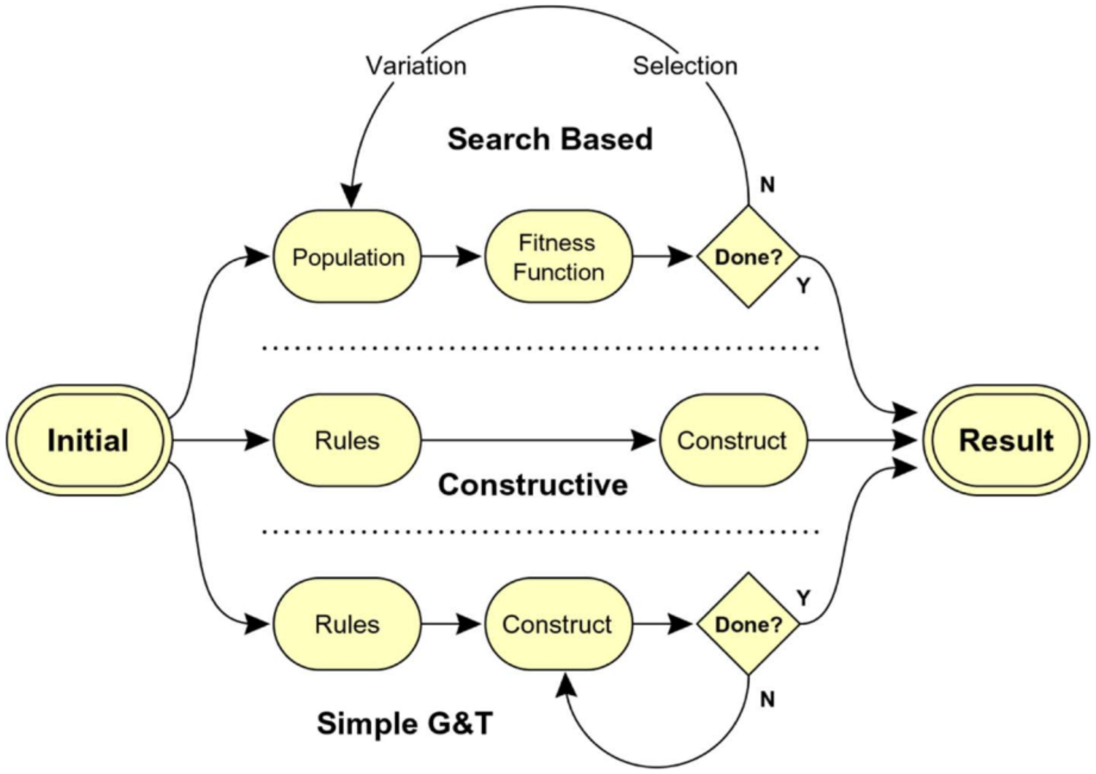
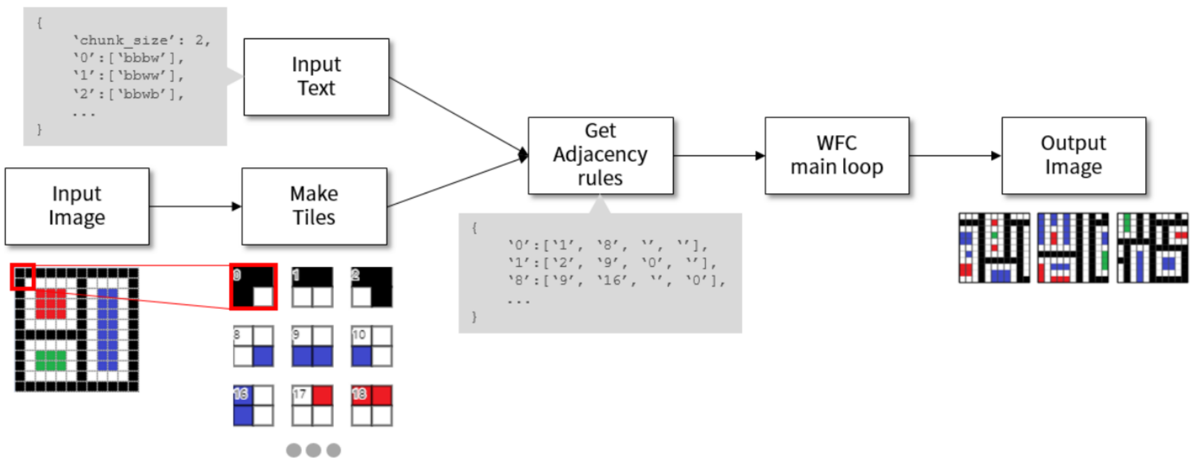


Figure 1: Three main procedural content generation approaches: Constructive, Simple Generate-and-Test, and Search-Based. The diagram illustrates that Search-Based PCG maintains a population of candidate solutions and iteratively applies variation and selection through a fitness function. This contrasts with Constructive methods, which follow fixed rules without iteration, and Generate-and-Test, which uses random construction with evaluation loops. [Togelius et al. 2011a]

2.2.3 Wave Function Collapse (WFC)

Wave Function Collapse (WFC) is a constraint-based procedural content generation technique inspired by quantum mechanics and popularised by Maxim Gumin, widely used for tile-based level and texture synthesis [Karth and Smith 2017]. As illustrated in Figure 2, WFC begins by analysing a small input sample to extract a set of unique tile patterns and learn valid adjacency rules between them. In the *pattern extraction* phase, the algorithm identifies all distinct tiles and records which tiles may legally neighbour each other. During the *constraint propagation and collapse* phase, each position in the larger output grid is initially assigned the superposition of all possible tiles; the algorithm then incrementally "collapses" one cell at a time to a single tile choice while propagating adjacency constraints to neighbouring cells, ensuring consistency across the grid [Kim et al. 2019].

The core WFC loop, shown in Figure 3, operates by selecting the cell with the lowest entropy, i.e. the highest uncertainty among possible tile assignments and collapsing it to a definitive tile. After each collapse, constraint propagation updates the possible tile sets of adjacent cells to reflect the learned adjacency rules. If a cell ever has no valid options, the algorithm backtracks to a previous state and retries a different collapse choice. A graph-based variant of WFC enhances efficiency by maintaining dependency graphs and re-evaluating only those nodes directly affected by a conflict, reducing redundant checks and improving scalability for larger maps [Kim et al. 2019].



```
{
    ‘chunk_size’: 2,
    ‘0’:[‘bbbw’],
    ‘1’:[‘bbww’],
    ‘2’:[‘bbwb’],
    ...
}
```

```
{
    ‘0’:[‘1’, ‘8’, ‘’, ‘’],
    ‘1’:[‘2’, ‘9’, ‘0’, ‘’],
    ‘8’:[‘9’, ‘16’, ‘’, ‘0’],
    ...
}
```

Figure 2: Overview of the WFC algorithm. Tiles are extracted from an input image, adjacency rules are derived, and these rules guide the main loop to generate a new output image with similar structure and coherence. This showcases the WFC's ability to replicate style and structure without repeating exact patterns [Kim et al. 2019].

The main loop of the WFC process, as seen in Figure 3, consists of three steps:

- **Observe:** Select the output tile location with the least entropy (i.e., highest uncertainty) and collapse its possible states into a definite tile.
- **Propagate:** Update neighbouring tile options based on the adjacency rules.
- **Conflict Detection and Backtracking:** If no valid option exists, the algorithm backtracks to a previous state and retries a different configuration.

The graph-based variant introduces a more efficient propagation strategy by selectively checking only the necessary nodes after conflicts, making it faster and more scalable for larger maps.

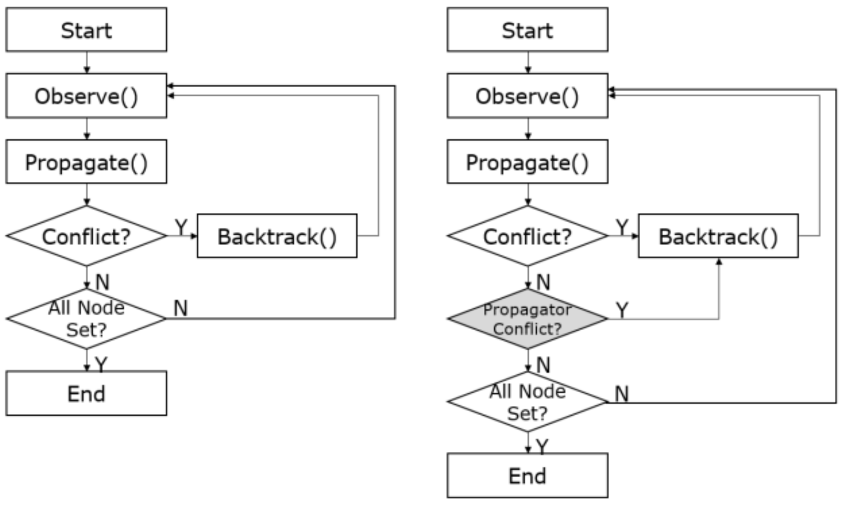


Figure 3: Main loop of the original WFC (left) and the graph-based WFC (right). The graph-based version improves efficiency by resolving conflicts through a focused re-evaluation of nodes that directly caused propagation failures [Kim et al. 2019].

2.2.4 Machine Learning-Based Procedural Content Generation (PCGML)

Machine Learning-Based PCG (PCGML) leverages data-driven models to learn patterns from existing game content and generate new elements that are both diverse and stylistically consistent, moving beyond the limitations of hand-crafted rule systems by generalising from annotated examples [A. Summerville et al. 2018].

PCGML techniques can be organised along two primary dimensions. First, by *input representation*: some methods operate on grid-based tile maps, others on sequence-based event streams (such as player actions or scripted encounters), and still others on graph-based structures like mission trees or room connectivity graphs. Second, by *learning paradigm*: supervised learning models predict content directly from input–output pairs; unsupervised approaches discover latent structure or clusters in level data; reinforcement learning agents learn to construct or modify levels via reward signals; and hybrid neuroevolution methods evolve both network architectures and weights using evolutionary algorithms. This taxonomy highlights the flexibility of PCGML in handling diverse data types and objectives [A. Summerville et al. 2018].

A common instantiation of PCGML employs neural networks trained via backpropagation to minimise a loss function $L(y,\hat{y})$, where y is the ground-truth content and $\hat{y}$ the model's prediction. Network weights w are updated by gradient descent according to

$$w_{t+1} = w_t - \eta \nabla_w L(y,\hat{y}),$$

with learning rate η. This approach has been used to generate coherent level segments, such as platform layouts or enemy placements, by learning spatial patterns directly from example levels.

PCGML has been successfully applied across multiple game domains, including platformer level generation (for example, synthesising new Super Mario Bros. segments), dungeon layout synthesis, procedural quest generation in role-playing games, and adaptive game balancing or difficulty tuning. Its strength lies in the ability to extend and remix existing styles from corpora, making it an ideal complement to rule-based and search-based PCG methods when rich, annotated datasets are available.

3 Related Work

Procedural Content Generation (PCG) has seen remarkable evolution, especially in games, design tools, and adaptive learning systems. The underlying challenge across domains has often been the same: How can systems create content that feels meaningful, personalised, and coherent, all without direct human authorship?

3.1 From Rule-Based Systems to Learning-Driven Models

Early approaches to PCG relied heavily on handcrafted rules and grammars. For instance, grammar-based generation of facades and interiors was explored in urban modelling, where researchers used semantic tags and templates to personalise architectural appearances [Silveira et al. 2015]. These methods excelled at structured, repetitive domains but lacked the flexibility to adapt dynamically to user input or player behaviour.

However, such rule-based systems suffered from limited expressiveness. They were brittle in adapting to new scenarios or content domains without significant re-authoring. This limitation sparked interest in hybrid systems.

3.2 Hybrid Approaches: Bridging Hand-Crafted and Learned Models

To address these gaps, researchers began blending deterministic design with stochastic or learned components. A notable example is the hybrid PCG framework by Hastings et al. Gellel and Sweetser 2020, which combines human-authored content with algorithmic control to allow more variability while maintaining quality.

More recent work explores blending reinforcement learning and machine learning with handcrafted templates — enabling content that adapts based on feedback. For example, in educational games, procedural levels generated from natural language instructions allow player-centered customisation while maintaining semantic learning goals Kumaran et al. 2023.

3.3 Learning to Generate: The Rise of PCGML

The emergence of PCG via Machine Learning (PCGML) opened the door for data-driven creativity. Summerville et al. A. Summerville et al. 2018 introduced a detailed taxonomy of PCGML, ranging from frequency-based models to deep neural networks and neuroevolution.

These models learn patterns from example data — be it 2D platformer levels, quest lines, or city layouts — and can generalise to produce novel content. Yet, even PCGML has its challenges: data scarcity, overfitting to style, and maintaining playability remain open issues.

> “Generating content is easy; generating *playable*, *interesting*, and *intentional* content is the hard part.” — PCGML insights from Summerville et al.

3.4 Large Language Models in PCG

A more recent research involves integrating large language models (LLMs) into PCG. Work such as that by Todd et al. 2023 demonstrates how LLMs can generate entire levels from simple textual prompts, specifically in generating levels for the puzzle game Sokoban. The study finds that LLMs can generate novel, playable levels, but performance scales significantly with dataset size.

3.5 Deep Learning for Procedural Generation

Building on the foundations of PCGML, deep learning has become a major force in procedural generation due to its capacity to learn complex, high-dimensional patterns from data. In their extensive survey, Liu et al. 2021 explore how Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), and Variational Autoencoders (VAEs) have been adapted for level design.

Each deep model brings unique strengths:

- **CNNs** capture spatial structures and have been used to generate tile-based maps or pixel-based layouts.
- **RNNs**, particularly Long Short-Term Memory networks (LSTMs), excel in sequence modelling and are used for generating platformer levels or mission chains.
- **GANs** enable unsupervised generation of realistic and novel level layouts by learning from raw examples, often outperforming rule-based methods in expressivity.

However, despite their representational power, PCGML approaches often suffer from issues such as mode collapse in generative adversarial networks, heavy dependence on large annotated datasets for supervised learning, and limited mechanisms for designer-driven control over structural and visual style [Shaker, Togelius, and Nelson 2016].

4 Research Methodology

4.1 Problem Statement

Procedural Content Generation (PCG) techniques have become integral in automating the creation of game environments, significantly reducing development costs and enhancing gameplay variability [Togelius et al. 2011b]. Traditionally, these techniques prioritise functional aspects, such as playability, often overlooking stylistic considerations like aesthetic consistency and thematic coherence, which are crucial for player engagement and immersion [Shaker, Togelius, and Nelson 2016]. Recent approaches that attempt to integrate style attributes into procedural level generation frequently encounter challenges due to the inherently subjective nature of style, difficulty in precise style feature extraction, and the complexities involved in consistently applying these features within procedural frameworks. Thus, there is a significant gap in effectively embedding well-defined style features into procedurally generated content to ensure both stylistic coherence and gameplay functionality.

4.2 Research Question

1. To what extent can a style-aware search-based procedural level generator generate tile-matrix levels, for 2D game levels, that adhere to user-defined structural and visual style features, compared to a style-agnostic SBPCG (with style weights set to zero) and baseline generators (random, constructive)?
2. How does the inclusion of user-defined visual aesthetic themes influence both the visual adherence and playability of generated 2D game levels levels?

4.3 Aim

The aim of this project is to design, implement and evaluate a style-aware search-based procedural level generator for tile-matrix platformer levels (from the Video Game Level Corpus), incorporating user-defined structural and visual style features within an FI-2POP (Feasible-Infeasible Two-Population) evolutionary framework.

4.4 Hypothesis

1. The style-aware SBPCG will achieve significantly higher structural and visual adherence than both the style-agnostic SBPCG and baseline generators.
2. The style-aware SBPCG will maintain playability on par with the style-agnostic SBPCG and the GVGAI(General video game ai) constructive baseline.

4.5 Methodology

4.5.1 Data Collection

Data for the Style-Aware Procedural Level Generator will be drawn primarily from the publicly available Video Game Level Corpus (VGLC) [A. J. Summerville et al. 2016], which aggregates 428 levels from twelve classic games in tile-based, graph-based, and vector formats. The VGLC provides raw level layouts and also comprehensive JSON legends, mapping symbolic annotations to semantic tags. This repository is maintained on GitHub [A. J. Summerville et al. 2017]. By leveraging this diverse corpus, we will obtain the structural, aesthetic, and semantic data required to identify style dimensions and train our generator.

4.5.2 Implementing a Search-Based Level Generator

This section provides a structured implementation methodology for developing a Search-Based Procedural Level Generator (SBPLG), specifically targeting the creation of tile matrix-based levels for platformer games. The outlined approach initially excludes the integration of stylistic features, focusing instead on demonstrating the foundational process. The implementation leverages extensive data from established sources, primarily the Video Game Level Corpus (VGLC) [A. J. Summerville et al. 2016].

The procedural generation process begins with data acquisition and preparation. Data will be sourced from the publicly available Video Game Level Corpus (VGLC), which contains annotated tile-based representations of various classic platformer games. Each game level from the corpus is represented as a two-dimensional tile matrix, with individual tiles annotated to reflect game-specific elements such as ground, enemies, hazards, and other relevant features [A. J. Summerville et al. 2016].

Initial Population Generation The evolutionary approach to procedural content generation necessitates an initial population of candidate solutions. This initial population can be generated either randomly or using a constructive level generator to establish a feasible baseline of playable levels. Following Zafar, Mujtaba, and Beg 2020 search-based approach, we will generate the initial population by seeding the genetic algorithm with levels produced by the sample constructive generator included in the General Video Game AI framework [Perez-Liebana et al. 2019]. This generator, implemented in `ConstructiveLevelGenerator.java` in the GVGAI repository[GAIGResearch 2025], places exactly one avatar, at least one goal sprite, and basic wall enclosures on the grid according to Video Game Description Language mapping rules [Ebner et al. 2013], ensuring minimal playability and constraint satisfaction from the outset. Levels in this phase will be represented directly as two-dimensional arrays of annotated tiles, based closely on the standardised representations found in The Video Game Level Corpus.

Genetic Representation and Operators In the Search-based procedural level generator, each candidate level is encoded as a direct genotype mapping to the full level grid: a 2D array of tiles, where each tile holds an array of sprite identifiers, so the chromosome length equals the product of level width and height [Zafar, Mujtaba, and Beg 2020]. Variation is achieved via a single-point crossover—choosing a random tile index in the flattened grid and swapping the corresponding segments between two parents and three mutation operators that (i) insert a random sprite into a tile (create), (ii) remove an existing sprite from a tile (destroy), or (iii) exchange sprites between two distinct tiles (swap). This design balances locality, small genomic changes yield small phenotypic differences, with sufficient diversity to explore the level space effectively.

Fitness Function Design The essential component of the search-based level generation is the design of a robust fitness function. This function objectively evaluates the candidate solutions, considering several key metrics. These metrics include reachability, ensuring that generated levels are solvable; complexity, quantifying the presence of multiple viable paths and obstacles; density and diversity of game elements, encouraging balanced and engaging gameplay; and visual and structural symmetry, contributing to a coherent design. The aggregated fitness function will be calculated as a weighted sum of these metrics, mathematically represented as:

$$F_{fitness} = w_1 \cdot F_{reachability} + w_2 \cdot F_{complexity} + w_3 \cdot F_{density} + w_4 \cdot F_{balance}, \quad (1)$$

where the weights w_n are determined empirically through initial experimentation.

Genetic Algorithm Execution A feasible-infeasible two-population (FI2POP) genetic algorithm will guide the iterative improvement of candidate levels. This method maintains two distinct populations: feasible solutions, which adhere to all design constraints, and infeasible solutions, which violate one or more constraints. Candidate solutions that fail to meet essential requirements such as solvability or basic gameplay criteria will populate the infeasible set, motivating continuous refinement until all candidates satisfy these constraints. This dual-population strategy has been shown to converge more reliably and efficiently than single-population, penalty-driven approaches in constrained optimisation tasks [Kimbrough et al. 2008], and has been successfully applied in General Video Game level generation to produce playable and diverse levels across multiple games [Zafar, Mujtaba, and Beg 2020]. The algorithm will employ a selection mechanism based on tournament selection to choose high-quality candidates for reproduction. Offspring will replace the lowest-performing candidates in the feasible population, ensuring the continuous evolution toward optimal solutions.

The iterative evolutionary process will continue until a predetermined termination condition is met, which would be reaching the specific fitness threshold, exhausting the maximum number of generations, or encountering no significant fitness improvements over multiple iterations.

Algorithmic Workflow The following pseudocode outlines the core evolutionary loop of the SBPLG (without style features):

```
P <- ConstructiveInit(POP_SIZE)                       # seed via GVGAI constructive generator
EvaluatePopulation(P)
while (timeElapsed < MAX_TIME) {
    (P_F, P_I) <- PartitionFeasibility(P)             # split into feasible/infeasible
    Parents <- TournamentSelect(P_F, P_I, TOUR_SIZE) # tournament selection
    Offspring <- OnePointCrossover(Parents, CROSS_PROB)
    Offspring <- Mutate(Offspring, MUT_PROB)               # create/destroy/swap
    P <- ReplaceWorst(P, Offspring)
    EvaluatePopulation(P)
}
return BestFeasible(P_F)
```

[Zafar, Mujtaba, and Beg 2020; Kimbrough et al. 2008; Perez-Liebana et al. 2019]

4.5.3 Extracting and Defining Structural Style Features

In order to quantify the structural patterns players intuitively recognise, how many distinct "rooms" a level has, how twisty or linear it feels, where dead-ends occur, and how much looping or branching there is, we will capture the inherent geometry of existing levels . To achieve this, we first convert each tile-matrix into a connectivity graph and compute a suite of topological metrics:

1. **Metric Computation.** On each level graph, we calculate:
   (a) **Tile–to–Graph Conversion.** Flood-fill the tile grid to identify connected "floor" regions as nodes, and insert an edge whenever two regions share a one-tile-wide corridor or doorway. The VGLC's graph-export tools provide a reference implementation of this step [A. J. Summerville et al. 2016].

   (b) **Metric Computation.** On each level graph, calculate:
      - *Room Count* (# components): the total number of discrete navigable areas, indicating how segmented the level is—higher values suggest more fragmented layouts, while lower values imply larger open spaces.
      - *Branching Factor* (mean node degree): the average number of connections per region, capturing the level's non-linearity—higher branching factors yield more choice and exploration.
      - *Linearity* (longest simple path / # nodes): the ratio of the length of the longest path through the graph to the total number of regions, measuring how straight or meandering the progression is—values near 1 indicate a mostly linear path, while lower values imply complex route options.
      - *Dead-End Rate* (% nodes with degree = 1): the proportion of regions that terminate in a single exit, quantifying the prevalence of one-way paths or "cul-de-sacs" and their impact on pacing.
      - *Loop Complexity* ($E - N + 1$): the cyclomatic complexity of the graph, equal to the number of independent loops—higher values denote richer looping structures that facilitate backtracking and exploration.
      - *Segment-Size Variance* (variance of component tile counts): the statistical variance in area size across connected regions, highlighting diversity in room or corridor scale and contributing to perceived pacing and challenge.

      Definitions of these structural metrics are drawn from Summerville et al.'s evaluation of platformer level metrics [A. Summerville et al. 2017], and Togelius, Preuss and Yannakakis's graph-based map generation work [Togelius, Preuss, and Yannakakis 2010]. These metrics have been shown to correlate strongly with perceived level structure in platformers and support targeted structural style analysis [A. Summerville et al. 2017]. By converting tile-based layouts into connectivity graphs and computing topological metrics, we can learn real distributions of structural features from human-designed levels, rather than relying on arbitrary hand-tuning and define precise targets (e.g. "Branching" corresponds to high looping complexity and dead-end rate) that the generator can aim for.

   (c) **Normalization.** Rescale each metric M_i into $[0, 1]$, where 1 indicates "ideal" for a given style.

   We introduce *style profiles* as target vectors $T = (t_1, \ldots, t_m)$ over the normalized metrics. For each archetype (e.g. "Open Linear," "Branching"), compute t_i by averaging M_i across a small example set drawn from the VGLC [A. J. Summerville et al. 2016]. We can expose each t_i via a simple configuration file, allowing designers to dial in bespoke values. Storing T alongside generator parameters lets the system load and apply the chosen profile at generation time.

4.5.4 Extracting and Defining Visual Aesthetic Themes

To capture and control the visual stylistic identity of generated levels, we extract a concise set of quantitative aesthetic metrics from existing examples. These metrics provide interpretable signatures of themes such as tile palette diversity, symmetry, and pattern regularity, enabling us to define high-level visual profiles that guide the PCG process [Dahlskog and Togelius 2013]. We then transform each VGDL (Video Game Description Language) level into a semantically annotated grid, from which we compute the following metrics:

Each VGDL sprite is first mapped to one of a small set of aesthetic categories (e.g. “ground,” “background,” “enemy,” “decoration”) to enable semantically meaningful analysis. On the resulting annotated tile grid we compute five core metrics: the *tile-type distribution*, given by the normalized frequency vector of categories (with entropy measuring palette diversity); *symmetry*, calculated as the proportion of tiles mirrored across horizontal or vertical axes; *repetition*, measured by the average run-length of consecutive identical tiles in rows and columns; *motif frequency*, obtained by counting small sub-patterns (e.g. 2×2 or 3×3 blocks) mined from exemplar levels; and *contrast ratio*, defined as the variance in category frequencies, which captures the prominence of focal elements. Each metric A_j is then rescaled into $[0,1]$. For named themes (e.g. “Icy,” “Dark”), we compute a target vector $V = (v_1,\dots,v_k)$ by averaging these normalized metrics over a small example set; for custom themes, designers will directly specify each v_j via a configuration file.

4.5.5 Incorporating Style Profiles into the Generator

After a designer selects one or both style profiles, structural $T = (t_1,\dots,t_m)$ and visual $V = (v_1,\dots,v_k)$, we fold them into the SBPCG loop as follows:

1. **Load Profiles.** At startup, parse the chosen structural profile to obtain T and the visual theme to obtain V.
2. **Compute Metrics.** For each candidate level ℓ, compute its structural metrics $M_i(\ell)$ (room count, branching, etc.) and aesthetic metrics $A_j(\ell)$ (symmetry, tile-type distribution, etc.) as defined in Sections 4.5.3 and 4.5.4.
3. **Evaluate Structural Adherence.** Measure how well ℓ matches the target structure:
$$F_{\text{struct}}(\ell) = 1 - \frac{1}{m}\sum_{i=1}^{m}\bigl|M_i(\ell) - t_i\bigr|.$$
Higher values indicate closer alignment with the designer’s chosen geometry [Zafar, Mujtaba, and Beg 2020].
4. **Evaluate Visual Adherence.** Measure how well ℓ matches the target visual theme:
$$F_{\text{visual}}(\ell) = 1 - \frac{1}{k}\sum_{j=1}^{k}\bigl|A_j(\ell) - v_j\bigr|.$$
This captures palette diversity, symmetry, motif frequency, and other visual signatures.
5. **Augment Composite Fitness.** Combine playability/aesthetic baseline F_{base} with the two style-adherence terms:
$$F_{\text{total}} = \alpha F_{\text{base}} + \gamma F_{\text{struct}} + \delta F_{\text{visual}},$$
where γ and δ are user-tunable weights controlling the relative importance of structural and visual style.
6. **Hard Constraints.** If exact style properties are required (e.g. “exactly 3 rooms” or “at least 30% symmetry”), mark any ℓ with $|M_i(\ell) - t_i| > \varepsilon$ or $|A_j(\ell) - v_j| > \varepsilon$ as infeasible (ε is the tolerance threshold). FI-2POP will then

drive these levels toward feasibility before optimizing F_{total}, ensuring strict adherence without manual penalty-weight tuning [Kimbrough et al. 2008].

This pipeline will ensure that, once a designer picks or defines a structural and visual profile, the evolutionary loop actively seeks levels whose geometry matches that profile while still satisfying playability and aesthetic goals.

4.5.6 Validation and Post-processing

Upon generation, candidate levels will undergo a validation phase to confirm their adherence to solvability constraints and gameplay requirements. Additionally, minor corrections and post-processing adjustments will be applied to ensure quality and functionality.Validated levels will be exported in a format compatible with standard game engines, such as JSON or simple text representations. Furthermore, visual renderings will be generated using visualization tools such as Matplotlib to facilitate qualitative assessments and analyses.

4.5.7 Tools and Frameworks (Environment)

The implementation and experimentation will be conducted in a Python-based environment, leveraging the following core tools and libraries:

We will use NumPy and Pandas for efficient array operations and data manipulation; NetworkX to construct connectivity graphs and compute structural metrics; DEAP to implement the FI-2POP genetic algorithm with custom crossover, mutation, and selection operators; scikit-learn to normalize style feature spaces, cluster exemplar levels, and support profile interpolation; Matplotlib to plot metric distributions, fitness convergence curves, and render level previews; JSON and PyYAML for defining and loading structural and visual style profiles; JupyterLab for interactive development, exploratory analysis, and rapid prototyping; a custom Python parser to ingest and preprocess the Video Game Level Corpus (VGLC) tile matrices and JSON legends [A. J. Summerville et al. 2016]; and an A star-based reachability solver written in Python, with optional integration into the GVG-AI Python wrapper for agent-based fitness evaluation. All code and dependencies will be version-controlled with Git and managed via a `conda` environment.

4.5.8 Evaluation and Analysis

To assess the benefits of our style-aware generator, we compare it against three standard baselines. The first baseline, a random generator, assigns tiles uniformly at random subject only to minimal playability constraints. The second, a constructive baseline, returns levels directly from the GVGAI sample constructive generator [Perez-Liebana et al. 2019]. The third, a style-agnostic search-based procedural level generator (SBPLG), implements our evolutionary loop with both structural and visual style weights set to zero, thereby optimizing only playability and basic aesthetic metrics as in Zafar et al.'s original setup [Zafar, Mujtaba, and Beg 2020]. Our style-aware SBPLG differs only in that it incorporates user-defined structural and visual adherence terms during fitness evaluation.

Each generated level is evaluated on four categories of metrics. Playability is measured via an A star-based reachability solver, yielding a reachability ratio that reflects the proportion of essential game objects accessible from the avatar's start position. Aesthetic quality is quantified using metrics such as tile-type distribution entropy, symmetry, and motif frequency, following Summerville et al.'s evaluation of platformer design metrics [A. Summerville et al. 2017]. Structural coherence is assessed by computing room count, branching factor, dead-end rate and loop complexity on the connectivity graph, as defined by Togelius, Preuss and Yannakakis [Togelius, Preuss, and Yannakakis 2010]. Finally, style adherence is captured by the structural F_{struct} and visual F_{visual} terms introduced in Sections 4.5.3 and 4.5.4.

Generated levels will be statistically analyzed to measure convergence of fitness values, diversity among generated solutions, and overall structural and functional quality. These metrics will then be compared with human-authored baseline levels from the corpus to assess the procedural generator's effectiveness.

4.5.9 Potential Challenges and Difficulties

Despite the anticipated benefits of style-aware SBPCG, several challenges must be addressed. Selecting and calibrating structural and visual metrics and their corresponding weights can be difficult, as inappropriate choices may lead to unintended level characteristics or convergence stagnation [Shaker, Togelius, and Nelson 2016]. The multi-objective nature of optimizing playability alongside structural and visual adherence may introduce conflicts, requiring careful tuning of FI-2POP parameters to ensure efficient search without excessive computational cost. The generalization of extracted style targets across diverse VGLC games may be limited by corpus biases or inconsistent level representations [A. J. Summerville et al. 2016]. Finally, validating stylistic success through both quantitative metrics and subjective user interpretation can be challenging, as human perceptions of "style" are inherently varied and may not align perfectly with computed adherence scores.

5 Research Plan

The research will be carried out in five sequential phases, each targeting a key component of the style-aware PCG pipeline. Table 1 summarizes the overall schedule.

Phase 1: Environment Setup and Preliminary Testing. In the first two weeks , we will configure the Python-based environment, install and test core libraries (NumPy, Pandas, NetworkX, DEAP, etc.), implement the Video Game Level Corpus ingestion pipeline, and verify baseline SBPLG and playability solvers.

Phase 2: Baseline Data Collection. Over the next three weeks, we will generate and record levels from the standard baselines (random and GVGAI constructive) across all named profiles, ensuring repeatability via fixed seeds.

Phase 3: Style Feature Extraction and Profiling. During weeks 6–8, we will extract structural (room count, branching, linearity, etc.) and visual (symmetry, motif frequency, tile-type distribution, etc.) metrics from exemplar levels, normalize them, and compute named profile target vectors.

Phase 4: Style-Aware Generator Implementation. In weeks 9–12, we will integrate user-defined structural and visual adherence terms into the SBPCG loop, implement profile loading in the FI-2POP framework, and conduct initial test runs to validate functionality.

Phase 5: Evaluation, Analysis, and Reporting. The final three weeks will be devoted to large-scale generation, quantitative statistical analysis, qualitative validation, drafting the research report and finalising documentation.

Table 1: Project Schedule

Phase	Start	End
Environment Setup and Testing	21 July 2025	3 August 2025
Baseline Data Collection	4 August 2025	24 August 2025
Style Feature Extraction & Profiling	25 August 2025	14 September 2025
Style-Aware Generator Implementation	15 September 2025	12 October 2025
Evaluation, Analysis & Reporting	13 October 2025	2 November 2025

6 Conclusion

This proposal has outlined a novel, style-aware search-based procedural content generation framework for tile-matrix platformer levels, integrating user-defined structural and visual style profiles into an FI-2POP genetic algorithm implemented in a Python environment. By extracting graph-based structural metrics and tile-level aesthetic signatures from the Video Game Level Corpus [A. J. Summerville et al. 2016] and embedding them as target vectors within the evolutionary loop, our approach enables designers to steer level generation toward precise stylistic goals while preserving playability. A comprehensive evaluation comparing against random, constructive, and style-agnostic baselines will validate the efficacy of style adherence and optimization efficiency. Successful completion of this research will provide the game design community with a flexible, extensible toolkit for personalised Procedural Content Generation.

References

Dahlskog, S. and J. Togelius [2013]. “Patterns as objectives for level generation”. In: *Design Patterns in Games (DPG), Chania, Crete, Greece (2013).*

Ebner, M. et al. [2013]. “Towards a video game description language”. In.

GAIGResearch [2025]. *GVGAI: Constructive Level-Generation Track Source Code*. https://github.com/GAIGResearch/GVGAI/tree/master/src/tracks/levelGeneration/constructive. Accessed: 2025-05-17.

Gellel, A. and P. Sweetser [2020]. “A hybrid approach to procedural generation of roguelike video game levels”. In: *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pp. 1–10.

Hendrikx, M. et al. [2013]. “Procedural content generation for games: A survey”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9.1, pp. 1–22.

Karth, I. and A. M. Smith [2017]. “WaveFunctionCollapse is constraint solving in the wild”. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pp. 1–10.

Kim, H. et al. [2019]. “Automatic generation of game content using a graph-based wave function collapse algorithm”. In: *2019 IEEE conference on games (CoG)*. IEEE, pp. 1–4.

Kimbrough, S. O. et al. [2008]. “On a feasible–infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch”. In: *European journal of operational research* 190.2, pp. 310–327.

Kumaran, V. et al. [2023]. “End-to-end procedural level generation in educational games with natural language instruction”. In: *2023 IEEE Conference on Games (CoG)*. IEEE, pp. 1–8.

Liu, J. et al. [2021]. “Deep learning for procedural content generation”. In: *Neural Computing and Applications* 33.1, pp. 19–37.

Perez-Liebana, D. et al. [2019]. “General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms”. In: *IEEE Transactions on Games* 11.3, pp. 195–214.

Shaker, N., J. Togelius, and M. J. Nelson [2016]. “Procedural content generation in games”. In.

Silveira, I. et al. [2015]. “Real-time procedural generation of personalized facade and interior appearances based on semantics”. In: *2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, pp. 89–98.

Summerville, A. et al. [2017]. “Understanding mario: an evaluation of design metrics for platformers”. In: *Proceedings of the 12th international conference on the foundations of digital games*, pp. 1–10.

Summerville, A. et al. [2018]. “Procedural content generation via machine learning (PCGML)”. In: *IEEE Transactions on Games* 10.3, pp. 257–270.

Summerville, A. J. et al. [2016]. “The vglc: The video game level corpus”. In: *arXiv preprint arXiv:1606.07487*.

Summerville, A. J. et al. [2017]. *TheVGLC: The Video Game Level Corpus*. https://github.com/TheVGLC/TheVGLC. Accessed: 2025-05-17.

Todd, G. et al. [2023]. “Level generation through large language models”. In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*, pp. 1–8.

Togelius, J., M. Preuss, and G. N. Yannakakis [2010]. “Towards multiobjective procedural map generation”. In: *Proceedings of the 2010 workshop on procedural content generation in games*, pp. 1–8.

Togelius, J. et al. [2011a]. “Search-based procedural content generation: A taxonomy and survey”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3, pp. 172–186.

Togelius, J. et al. [2011b]. “What is procedural content generation? Mario on the borderline”. In: *Proceedings of the 2nd international workshop on procedural content generation in games*, pp. 1–6.

Zafar, A., H. Mujtaba, and M. O. Beg [2020]. “Search-based procedural content generation for GVG-LG”. In: *Applied Soft Computing* 86, p. 105909.

Wits University Faculty of Science post-graduate student AI declaration

I understand that the use of generative AI tools (such as ChatGPT or similar) without explicitly declaring such use constitutes a form of plagiarism and is classified by Wits University as academic misconduct.

I declare that in the course of conducting the research towards my degree or in the preparation of this thesis/dissertation/research report (select one by marking with an X):

I **did not** make use of generative AI tools ☐

I **did** make use of generative AI tools for the following (tick all that apply):

1.	Idea Generation (research problem/design, hypothesis)	☐
2.	Sourcing Related Work (summarising, identifying sources)	☐
3.	Methods and Experiment Design (experiment setup, model tuning)	☐
4.	Data Analysis (presentation, coding, interpretation)	☐
5.	Theoretical Development (theorem proving, conceptual analysis)	☐
6.	Code Development (generating algorithms, writing scripts)	☐
7.	Presentation (rendering graphics, formatting)	☐
8.	Editing (grammar, readability)	☒
9.	Writing (text generation, document structuring)	☐
10.	Citation Formatting (structuring, organising)	☒

If other uses were involved, please specify below:

Generative AI tool used (list all)	**Used for?**

If generative AI tools were used as an integral part of the experimental design or in the direct execution of my research, I confirm that details of this use are clearly outlined in the relevant experimental/methodology chapters of my thesis/dissertation/research report.

Student number:
2438634

Candidate signature:

[illegible]

Date:
20/05/2025