Procedural Content Generation (PCG) Implementation Framework

This experimental design to evaluate a Delaunay-triangulation-inspired Procedural Content Generation (PCG) system implemented in Bathala's map generation. The combines computational geometry, pathfinding algorithms, and distribution techniques to assess the effectiveness of graph-based corridor generation in maintaining player engagement and achieving flow state through exploration.

System Development Approach

The PCG system was developed using TypeScript and follows a modular pipeline architecture to ensure consistent world generation throughout gameplay sessions. The implementation draws inspiration from established roguelike titles including Hades, Dead Cells, and Spelunky, which employ procedural generation based on spatial graphs and deterministic seeding.

PCG System Architecture

Chunk System:

The world is divided into square regions called "chunks" (default: 50×50 tiles each). Chunks are the basic unit of generation, loaded dynamically as players explore. Coordinates are expressed as (chunkX, chunkY) representing the chunk's position in the infinite world grid.

Corridor Generator: Executes a seven-stage pipeline to create interconnected pathways within each chunk

Cache Manager: Stores up to 100 previously generated chunks using LRU (Least Recently Used) eviction

Node Placement System: Distributes gameplay points-of-interest (combat encounters, shops, events) on valid corridor tiles

Connectivity Manager: Creates seamless connections between adjacent chunk

Deterministic RNG: Ensures identical chunk layouts from identical seeds, enabling reproducible worlds

Generation Pipeline

The system operates through seven sequential stages. It begins with Region Point Sampling, which places anchor points that will serve as corridor endpoints. Next is Triangulation, where these anchor points are connected to form a connectivity graph. In the Edge Sorting stage, connections are ordered by distance to prioritize local corridors. Following this, Multi-Waypoint Pathfinding is applied using the A* algorithm to carve out the actual corridors. The 2×2 Block Pruning stage then removes overly wide sections to maintain narrow and coherent pathways. Afterward, Dead-End Extension extends terminal corridors to minimize unnecessary backtracking. Finally, Node Placement positions combat, shop, and event nodes on valid tiles within the generated layout.

Region Point Sampling System

The first stage uses a sampling system that generates seed points that serve as anchor locations for the maze corridor network. These points are distributed across the grid with enforced minimum distance to prevent clustering.

Each candidate point (xi,yi)(x_i, y_i)(xi,yi) is only accepted if it satisfies the minimum distance requirement from all previously placed points, ensuring even spatial distribution and preventing overlap between regions.

Distance Check Formula:

Condition: $distanceSq \ >= \ minRegionDistance^2$

$distanceSq \ = [(x_1 - x_2)^2 \ + \ (y_1 -)^2]$

Table 22:
Distant Check Variables:

Notation	Description
distanceSq	The squared Euclidean distance between two points.
x_1	The x-coordinate of the newly sampled candidate point being evaluated
x_2	The y-coordinate of the newly sampled candidate point being evaluated
y_1	The x-coordinate of an already-accepted seed point in the set
y_2	The y-coordinate of an already-accepted seed point in the set
minRegionDistance	The minimum allowed spacing between any two seed points; default value is 4 tiles

The Distance Check Formula is used in Bathala's procedural overworld generation to ensure that key nodes, such as events or enemy encounters, are evenly spaced across the map. By calculating the squared Euclidean distance between a newly sampled candidate point and all previously accepted seed

points, the system enforces a minimum region distance—typically set to 4 tiles—between nodes. This prevents clustering and maintains balanced distribution, supporting both gameplay pacing and exploration. The formula, $distanceSq = (x_1 - y_1)^2 + (x_2 - y_2)^2$, checks that the candidate's distance from existing points meets or exceeds the required threshold, ensuring a fair and engaging overworld layout for each run.

Sampling Process:

The sampling process runs in a loop that continues until the target regionCount of points has been accepted or the MAX_REGION_POINT_ATTEMPTS limit (10,000) is reached. During each attempt, the system generates a random candidate coordinate (x,y) within the bounds of the chunkWidth and chunkHeight. This candidate is then checked against all previously accepted points. To optimize this check, the system compares squared distances (avoiding the square root operation). If the squared Euclidean distance between the candidate and any existing point is less than the minRegionDistance squared, the candidate is rejected. If the candidate successfully maintains the minimum distance from all accepted points, it is added to the list, and the process repeats.

Design Rationale:

Rejection sampling ensures minimum spacing without complex spatial indexing structures. The squared-distance optimization improves performance.

The attempt limit prevents infinite loops when configuration is over-constrained (e.g., trying to place 200 points with 10-tile spacing in a 50×50 chunk).

Performance Metrics:

regionCount (default: 100): Controls corridor density. Higher values create denser networks with more branches.

minRegionDistance (default: 3 tiles): Prevents clustering. Lower values allow tighter packing; higher values enforce uniform distribution.

Simplified Delaunay Triangulation

A simplified nearest-neighbor triangulation that approximates Delaunay triangulation properties by connecting each seed point to its three closest neighbors. This creates a network of edges that will become corridor paths. The system approximates connectivity through k-nearest neighbor graphs:

Algorithm:

To build the graph, the algorithm iterates through each region point. It computes the Euclidean distance to all other points in the set, sorting these distances to identify the k=3 nearest neighbors. Undirected edges are then created to connect to these three closest points. To manage the edge list efficiently and eliminate duplicates (such as both an edge from A to B and B to A), all edges are stored in a canonical format, such as (min_point, max_point).

Euclidean Distance Formula:

$$distance = \sqrt{[(xi - xj)^2 + (yi - yj)^2]]}$$

Table 23:
Euclidean Distance

Notation	Description
distance	The straight-line Euclidean distance between two points.
xi	The x-coordinate of the first point
yi	The y-coordinate of the first point
xj	The x-coordinate of the second point
yj	The y-coordinate of the second point

The Euclidean Distance Formula is used in Bathala's overworld generation to calculate the straight-line distance between two points, such as seed nodes and their potential neighbors. The formula, edgeLength(p, q) = √[(xi − xj)² + (yi − yj)²], determines the actual distance between endpoints, which is essential for sorting and processing edges during map construction. In Table 23, xi and yi represent the coordinates of the first seed point, while xj and yj are the coordinates of the second point being evaluated. This calculation helps manage node placement and connectivity, ensuring balanced spacing and efficient pathfinding throughout the procedurally generated

Edge Length Calculation:

$$edgeLength(p,\ q)\ =\ \sqrt{[(xq\ -\ xp)^2\ +\ (yq\ -\ yp)^2]]}$$

Edges are sorted by length before pathfinding, prioritizing short local connections over long-distance corridors.

Table 24: *Edge Length*

Notation	Description
edgeLength	The Euclidean distance between two endpoints of an edge, used for sorting the edge processing queue
xq	The x-coordinate of the first endpoint (point p) of the edge
xp	The y-coordinate of the first endpoint (point p) of the edge
yq	The x-coordinate of the second endpoint (point q) of the edge
yp	The y-coordinate of the second endpoint (point q) of the edge
p	The first endpoint of the edge, containing properties (x,y)
q	The second endpoint of the edge, containing properties (x,y)

The edge length calculation in Bathala uses the Euclidean distance formula, edgeLength(p, q) = √[(xq − xp)² + (yq − yp)²], to measure the straight-line distance between two endpoints of an edge. This value is essential for sorting edges before pathfinding, ensuring that shorter, local connections are

prioritized over longer corridors during overworld generation. In Table 24, xq and xp represent the coordinates of the first endpoint (point p), while yq and yp are the coordinates of the second endpoint (point q). Each endpoint, p and q, contains its own (x, y) properties. By sorting edges by length, the system supports efficient node connectivity and balanced map layout, which is crucial for procedural generation and gameplay pacing.

Design Rationale:

This design was chosen for its practical simplicity. While a true Delaunay triangulation using an $O(n \log n)$ sweep-line algorithm is more efficient, the $O(n^2)$ distance computation approach is significantly simpler to implement. The choice of k=3 (connecting each point to its three nearest neighbors) ensures a reasonable average vertex degree, providing good graph connectivity without creating excessive edge density. Finally, sorting and processing the shorter edges first is a key heuristic, it encourages the formation of naturally clustered corridor regions before introducing long-distance connections, which helps guide the subsequent map generation steps.

Multi-Waypoint A* Pathfinding

A cost-based pathfinding algorithm that finds optimal paths between region points while favoring straight corridors and penalizing excessive turns. The system carves corridors between connected region points using A* pathfinding with custom cost modeling:

A* Cost Formula:

$$f(n) = g(n) + h(n)$$

Table 25:
Cost Formula Variables:

Notation	Description
n	The grid cell being evaluated.
f(n)	Total estimated cost to reach goal through node n. Used to prioritize which nodes to explore next.
g(n)	Actual accumulated cost from start to node n, increases as path progresses
h(n)	Heuristic estimate of remaining cost from n to goal and guides search toward target.

Multi-Waypoint A* Pathfinding in Bathala uses a cost-based algorithm to determine optimal routes between region points on the overworld grid. This system favors straight corridors and penalizes excessive turns, ensuring efficient and natural-looking paths. The algorithm applies the A* cost formula, f(n) = g(n) + h(n), where f(n) is the total estimated cost to reach the goal through node n, g(n) is the actual accumulated cost from the start to node n, and h(n) is the heuristic estimate of the remaining cost from n to the goal. Table 25 outlines these variables, which are used to prioritize nodes during pathfinding and guide the search toward the target region point. This approach supports procedural

generation by carving balanced and navigable corridors throughout the overworld.

Design Rationale:

This ensures the paths are generated with proportional length dependent on its chunk size, discouraging excessive path length. Each potential path is evaluated by combining the actual cost of reaching a point with an estimated cost of getting from that point to the goal. The actual cost ensures that the search avoids unnecessarily long routes, while the estimated cost directs the search efficiently toward the destination. When the estimation is accurate and never exceeds the true remaining distance, the algorithm finds the optimal path. Paths with lower combined costs are considered more promising and are explored first, balancing efficiency and accuracy in the search process.

Heuristic (Manhattan Distance):

$$h(n) = |x_n - x\_goal| + |y_n - y\_goal|$$

Where:

Table 26:
Heuristic (Manhattan Distance):

Notation	Description

h(n)	Estimated cost from current position to goal position
x□	The x-coordinate of the current node being evaluated
y□	The y-coordinate of the current node being evaluated
X_goal	The x-coordinate of the target destination
y_goal	The y-coordinate of the target destination
| |	Absolute value that makes negative distances positive

The heuristic calculation in Bathala's pathfinding system uses Manhattan distance, defined as h(n) = |x□ - x_goal| + |y□ - y_goal|. This formula estimates the cost from the current node to the goal by summing the absolute horizontal and vertical distances. Table 26 outlines the variables: h(n) is the estimated cost, x□ and y□ are the coordinates of the current node, while x_goal and y_goal are the coordinates of the target destination. The use of absolute values ensures all distances are positive. Manhattan distance is ideal for grid-based movement restricted to four directions, supporting efficient and accurate pathfinding in Bathala's overworld.

Tile Traversal Cost:

$$tileCost = BASE\_TILE\_COST + \Delta dir$$

$$\Delta dir = 0.0,\ if\ direction\ unchanged$$

$= 0.1,\ if\ direction\ changed$

Table 27:
Tile Traversal Cost

Notation	Description
tileCost	Total cost to move from current tile to neighboring tile
BASE_TILE_COST	Cost for moving one tile in any direction. (default = 1.2)
Δdir	Additional penalty applied when path changes direction

The tile traversal cost formula in Bathala determines the total cost of moving from the current tile to a neighboring tile during pathfinding. The formula is tileCost = BASE_TILE_COST + Δdir, where BASE_TILE_COST is the standard cost for moving one tile (default value: 1.2), and Δdir is an additional penalty applied when the path changes direction—set to 0.0 if the direction remains the same, or 0.1 if it changes. Table 27 outlines these variables, ensuring that straight paths are favored and excessive turns are penalized, resulting in more natural and efficient corridor generation across the overworld grid.

Design Rationale:

This design rationale is focused on generating high-quality, natural-looking paths. A Direction Penalty is included to discourage frequent turns and zigzag patterns, which results in corridors that are straighter and feel more organic. To improve the overall map structure, Applying to promote an interconnected

network, preventing the formation of isolated, parallel corridors. Finally, the Manhattan Heuristic was chosen for its admissibility; because it never overestimates the actual cost on 4-connected grids, it guarantees that the A* algorithm will find the optimal path.

Waypoint Generation System

A heuristic system that inserts waypoints between path endpoints to create varied, aesthetically interesting corridor shapes. Prevents all corridors and paths to be only consisting of straight line paths. It creates organic, non-linear corridors, between start and goal points based on total distance:

Distance Calculation:

$$D = |dx| + |dy|$$

$$Where: dx = x\_end - x\_start$$

$$dy = y\_end - y\_start$$

Table 28:
Distance Calculation

Notation	Description
D	Total Manhattan distance between start and end points, determines if waypoints are needed
dx	Horizontal offset from start to end
dy	Vertical offset from start to end

x_start	X-coordinate of the path's starting point
y_start	Y-coordinate of the path's starting point
x_end	X-coordinate of the path's ending point
y_end	Y-coordinate of the path's ending point

This system is designed to prevent the procedural generation from creating boring, straight-line corridors. It works by first calculating the Manhattan distance between a path's start and end points. This distance which is simply the total horizontal distance plus the total vertical distance is used as a heuristic. Based on this distance value, the system strategically inserts one or more intermediate waypoints between the start and end. This forces the pathfinding algorithm to navigate to these waypoints, breaking up what would have been a direct path and creating more "organic" and aesthetically interesting, non-linear corridor shapes.

Waypoint Styles:

L-Shape (D < 8 tiles):

- Single waypoint at (x_goal, y_start) or (x_start, y_goal)
- 50% chance determines horizontal-first vs. vertical-first
- Creates simple corner corridors

Step Pattern (8 ≤ D < 15 tiles):

- Two waypoints near ⅓ and ⅔ positions with random jitter (±2 tiles)
- Creates stair-step appearance

Zigzag Pattern (D ≥ 15 tiles):

- Three or more waypoints with perpendicular offsets
- Creates weaving corridors for visual complexity

Path Segmentation: A* runs separately for each segment (start→waypoint$_1$→waypoint$_2$→goal), removing duplicate tiles at segment boundaries.

Fallback: If any segment fails, the system attempts a direct path from start to goal, guaranteeing all edges get carved.

Design Rationale: Pure shortest paths are straight and visually boring. Waypoints introduce controlled randomness and visual variety while maintaining navigability.

Post-Processing Systems

Double-Wide Path Pruning

A* may carve 2×2 blocks of corridor tiles, creating room-like spaces that violate the narrow corridor aesthetic required for poker combat. This post-processing system phase detects and eliminates the 2×2 blocks of path tiles to maintain single-tilewide corridors throughout the maze.

Detection Formula:

$$Block = \{(x, y), (x + 1, y), (x, y + 1), (x + 1, y + 1)\}$$

$$Condition: All\ four\ positions\ must\ be\ PATH\_TILE.$$

Table 29: *Detection Formula Variables*

Notation	Description
Block	Total cost to move from current tile to neighboring tile
x	The top-left x-coordinate of the 2×2 block being checked
y	The top-left y-coordinate of the 2×2 block being checked
PATH_TILE	Tile type value representing walkable corridor space.

It works by iterating through the grid and checking 2x2 groups of tiles. The Block is defined as a set of four adjacent tiles starting from a top-left coordinate (x, y). The condition is met if all four tiles in this Block—(x, y), (x+1, y), (x, y+1), and (x+1, y+1)—are PATH_TILEs. When such a block is detected, it is flagged as an undesirable "room-like" space created by the A* algorithm, which can then be "pruned" to restore the intended single-tile-wide corridor.

Pruning Algorithm:

1. For each tile in detected block, count PATH neighbors outside the block (external degree)
2. Sort tiles by ascending external degree
3. Attempt to remove lowest-degree tile (convert to wall)
4. Skip if removal would create disconnection:

    Tile has ≥3 neighbors (junction point)

    Tile has 2 opposite neighbors (straight corridor segment)

5. Repeat scan until no 2×2 blocks remain or 10 iterations reached

Design Rationale: Maintains single-tile-wide corridors essential for tactical positioning in combat. Iterative approach handles cascading blocks created by initial removals.

Expected Outcome: The grid is adjusted so that in any 2×2 area, no more than three of the tiles are paths tiles. Corridors remain only one tile wide at intersections to maintain structure, while overall path connectivity is preserved. The modification process continues for up to ten iterations or until no 2×2 blocks violate the constraint.

Dead-End Reduction

A post-processing enhancement that extends dead-end corridors forward to create loops and improve connectivity. Pathfinding creates long dead-ends (tiles with exactly one PATH neighbor), increasing frustrating backtracking.

Dead-End Detection Formula:

$$pathNeighborCount = |\{n \in N(x,y) : n \in bounds \wedge grid[n] = PATH\_TILE\}|$$

Table 30:
Dead-End Detection Variables

Notation	Description
pathNeighborCount	Count of valid path neighbors
n	A neighbor position being evaluated
grid[n]	Tile type value at position n
PATH_TILE	Constant (0) for walkable corridors
x	X-coordinate of the tile being evaluated
y	Y-coordinate of the tile being evaluated

It operates by examining a specific tile at coordinates (x, y) and calculating its pathNeighborCount. This count is determined by iterating through all of the tile's adjacent neighbors (n) and checking if they satisfy two conditions: the neighbor must be within the valid grid boundaries (n \in bounds) and its type must be a walkable corridor (grid[n] = PATH_TILE). A tile is therefore confirmed to be a dead-end if its final pathNeighborCount is exactly 1, as this signifies it has only a single connection to the rest of the path.

Algorithm:

1. Identify all dead-end tiles (degree = 1)
2. For each dead-end:
    - Compute extension direction: dir = dead_end - connected_neighbor
    - Extend forward up to 5 tiles in direction dir
    - Stop if out-of-bounds or hits existing PATH (creates loop)
    - Otherwise carve extension tiles as PATH

Dead-End Extension Formula:

$$d = (x\_deadEnd - x\_connection, y\_deadEnd - y\_connection)$$

Table 31:
Dead-End Extension Variables

Notation	Description
d	: Unit direction vector pointing away from the connected corridor
x_deadEnd	X-coordinate of the dead-end tile
y_deadEnd	Y-coordinate of the dead-end tile
x_connection	X-coordinate of the single connected path tile
y_connection	Y-coordinate of the single connected path tile

It works by subtracting the coordinates of the "connection" tile (the path leading into the dead-end) from the coordinates of the "dead-end" tile itself. The result is a simple direction vector, d, that points away from the existing corridor and into the empty space where an extension can be built. For example, if a

dead-end tile is at (10, 5) and its only connected path tile is at (9, 5), the formula (10-9, 5-5) yields (1, 0), a unit vector pointing "right." This d vector can then be used to correctly place a new room or feature, ensuring it attaches to the dead-end and faces the correct, open direction.

Design Rationale: Short extensions (1-5 tiles) increase connectivity without creating long tendrils. Often creates loops, improving navigation options and reducing backtracking.

Deterministic Seeding

Infinite world generation requires reproducible chunk layouts:

Chunk Seed Formula

$$seed = ((chunkX \times f) \oplus (chunkY \times b) \oplus globalSeed)\ \&\ 0x7FFFFFFF$$

Table 32:
Chunk Seed Formula

Notation	Description
seed	The seed value for a specific chunk
chunkX	The x-coordinate of the chunk in chunk-space
chunkY	The y-coordinate of the chunk in chunk-space
globalSeed	The world's master seed value, consistent across all chunks
f	Large prime number for x-axis hash distribution. (default: 73856093)

b	Large prime number for y-axis hash distribution. (default: 19349663)
$\oplus$	Bitwise XOR operation (combines numbers by flipping matching bits)
&	Bitwise AND operation
0x7FFFFFFF	Bit masks result to 31-bit positive integer (prevents negative numbers)

It guarantees that adjacent chunks, such as those at (0,0) and (1,0), use uncorrelated seeds to produce completely different layouts. At the same time, it ensures that the same chunk coordinates will always generate the same seed, making the world layout consistent and reproducible. Finally, using a different global seed will result in an entirely different world, allowing for overall variation while maintaining local determinism.

Linear Congruential Generator

The chunk seed drives a simple random number generator:

$$seed\_new = (seed \times 16807)\ mod\ 2147483647$$

$$randomValue = (seed\_new - 1) / 2147483646$$

Where:

Table 33:
Linear Congruential Generator

Notation	Description
seed_new	The updated seed value after one LCG iteration
seed	The current seed value before iteration
16807	The LCG multiplier chosen for distribution
2147483647	The LCG modulus, a Mersenne prime ensuring full period
randomValue	Normalized random value in range
2147483646	Denominator for normalization (modulus - 1)

This section describes the game's core pseudo-random number generator (PRNG), which uses a classic and computationally efficient Linear Congruential Generator (LCG). This specific implementation is a well-known standard (MINSTD) that uses the multiplier 16807 and the prime modulus $231-1$ (2147483647), constants famous for producing a long, high-quality, and well-distributed sequence of numbers. The system is entirely deterministic: every time a chunk seed is provided as the initial input, the LCG will produce the exact same sequence of numbers. This is critical for procedural generation, as it ensures a specific game chunk can be perfectly and consistently reconstructed. The final step normalizes the large integer result into a standard floating-point

value between 0.0 and 1.0, making it easy for other systems to use for predictable, random decisions.

Design Rationale: Fast computation, minimal memory, sufficient randomness quality for spatial distribution. The same seed always produces the same random sequence, guaranteeing identical chunk layouts.

Node Placement System

Points-of-interest (combat, shops, events) are placed on validated corridor positions:

Valid Position Criteria

A tile is considered a valid position only if it meets three specific criteria. First, the tile must be classified as a PATH, ensuring it is part of the corridor system and not a wall. Second, it must be sufficiently far from boundaries, defined by a distance from any chunk edge of at least 3 tiles. Finally, the position must have adequate surrounding space, which is verified by checking that the number of open neighbors in its 3x3 window is 5 or more, ensuring there is enough room for activities such as combat.

Minimum Node Spacing:

$$minNodeDistance = chunkSize / 4$$

$$For\ 50{\times}50\ chunks:\ minNodeDistance = 12.5\ tiles$$

Node Count:

$$nodeCount = BASE\_NODE\_COUNT + random(0 \; or \; 1)$$

Default: 3-4 nodes per chunk

Placement Algorithm:

1. Find all valid positions (PATH tiles meeting criteria)
2. Place first node at position maximizing distance to chunk center
3. Place subsequent nodes maximizing distance to previously placed nodes
4. If no position meets minNodeDistance, select randomly from valid positions
5. Assign node types: combat, elite, shop, event, campfire, treasure

Design Rationale: Spatial spreading ensures encounters feel distributed rather than clustered. Greedy farthest-point placement creates even coverage. Edge padding prevents awkward node placement at chunk transitions

Data Collection and Metrics

To validate the effectiveness and quality of the generation process, the system implements a robust data collection pipeline. These metrics are essential for objectively evaluating the procedural output against key design goals.

Target Metrics

To validate the effectiveness and efficiency of the Procedural Content Generation (PCG) system, the following table outlines the specific target metrics for performance, playability, and quality. These metrics ensure the system

generates balanced, fully accessible layouts while limiting player frustration and maintaining high performance.

Table 34:
Target Metrics

Metric	Target	Purpose
Path coverage	35-42%	Balances corridor identity with navigation
Connectivity ratio	≥95%	Ensures playable, reachable layouts
Generation time (P95)	<25ms	Maintains average 60 FPS during loading
Cache hit rate	>85%	Reduces redundant generation
Dead-end ratio	<20%	Limits backtracking frustration
Node accessibility	100%	Critical for progression

This validation framework is designed to ensure the PCG system produces high-quality, fully functional, and performant layouts by setting firm targets for playability and technical efficiency. The foremost goal is guaranteeing a complete player experience, enforced by a non-negotiable 100% Node Accessibility target to ensure all critical progression points are reachable. This is complemented by a high Connectivity Ratio (≥95%) to make sure the main layout is navigable, and a Dead-end Ratio of less than 20% to limit player frustration from excessive backtracking. For level "feel," the Path Coverage target of 35-42% ensures a balance between open space and a distinct corridor identity. On the performance front, the stringent P95 Generation Time of under 25ms is set to maintain a smooth 60 FPS experience by preventing stuttering during new chunk loading, while the high Cache Hit Rate ($>$85%) minimizes redundant generation when players revisit areas.

Testing Methodology

To validate the robustness, correctness, and performance of the PCG system, a multi-stage testing methodology was implemented. This process began with validating the core algorithms in isolation through unit testing, followed by integration testing and performance profiling to verify the stability and efficiency of the complete generation pipeline.

Validation Approach

This validation approach involves generating known configurations and verifying a set of expected properties. Tests confirm that seeding is deterministic, ensuring identical seeds produce identical layouts while adjacent chunks have uncorrelated seeds. The graph generation is validated by checking that region points respect minimum distance constraints and that approximately k edges are created per vertex. Finally, the resulting map structure is tested to verify that the A algorithm finds valid paths* between all connected regions and that no 2x2 PATH blocks remain after the pruning step.

Integration Testing

Performance Profiling

To ensure the PCG system operates efficiently and does not compromise the player's experience, rigorous performance profiling was conducted. These benchmarks were designed to measure generation speed, cache effectiveness, and memory stability under sustained use.

Benchmarks:

Table 35:
Benchmarks of Performance Profiling

Test	Description
Cold generation	1,000 sequential chunks, measure latency distribution
Cached Retrieval	Verify rate >85%, retrieval latency <1ms
Memory Stability	Monitor heap during 500-chunk generation, verify no leaks

The benchmark results validate that the Procedural Content Generation (PCG) system is efficient and stable, ensuring it won't negatively impact the player's experience. The Cold Generation test establishes the baseline performance for creating new content, measuring the latency distribution across 1,000 chunks to simulate a player entering completely new areas. This identifies worst-case generation times that could cause stutter. The Cached Retrieval test confirms the system's efficiency for common scenarios, like backtracking; its high hit rate and rapid latency ensure that previously visited areas reload almost instantly. Finally, the Memory Stability test simulates a long play session, verifying that no memory leaks occur during a 500-chunk generation, which prevents the game from crashing due to memory exhaustion over time.

Limitations and Considerations

To provide a transparent and critical assessment of this research, this section details the known limitations and practical considerations of the implemented PCG system. These constraints are categorized into specific algorithmic design trade-offs (System Limitations) and a formal analysis of computational overhead (Performance Constraints).

System Limitations

The architectural and algorithmic design of the PCG system incorporates specific trade-offs to prioritize development efficiency and a narrow corridor aesthetic. These design choices, which balance simplicity against computational workload, result in the following known limitations:

Simplified Triangulation: k-NN approximation, a modified version of Delaunay rather than true Delaunay may produce suboptimal connectivity in rare cases. Impact minimized through post-processing.

Rejection Sampling Constraints: Over-constrained configurations may fail to place all requested regions.

Single-Threaded Generation: JavaScript execution prevents parallel chunk generation, limiting preloading performance.

Performance Constraints

Generation time scales as $O(n^2 + e \times area \times \log area)$ where n=regionCount, e=edges, area=chunk size. Practical limit: 150-200 regions for real-time generation (<25ms).

Expected Outcomes

Based on the described design and methodology, the study anticipates that the PCG system will successfully deliver on four key outcomes, ensuring a high-quality, performant, and reproducible player experience.

Based on methodology, this anticipates:

- **Path Coherence**: System maintains appropriate corridor density across chunks and exploration styles
- **Performance Achievement**: Generation time remains under frame budget (25ms) for standard configurations
- **Connectivity Guarantee**: 95%+ of corridor tiles reachable from any starting position
- **Deterministic Reproducibility**: Identical seeds produce identical layouts, enabling world sharing and competitive play