3.3.2 Procedural Content Generation (PCG) Implementation

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 rejection sampling with a Poisson-disc-like distribution to place region anchor points. 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.

∀ existing points: distance(candidate, existing) ≥ minRegionDistance

Where distance is Euclidean norm:

$$\sqrt{}[(x_1 - x_2)^2 + (y_1 - y_2)^2]$$

Sampling Process:

- Generate random coordinates: x = ⌊random() × chunkWidth⌋, y = ⌊random() × chunkHeight⌋
- Check distance to all previously accepted points using squared distances (avoids expensive square root operations)
- Accept if all distances meet threshold; otherwise reject and try again
- Continue until regionCount points placed or MAX_REGION_POINT_ATTEMPTS (10,000) exceeded

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

Rather than computing true Delaunay triangulation (which requires circumcircle tests), the system approximates connectivity through k-nearest neighbor graphs:

Algorithm:

- For each region point p_i, compute Euclidean distance to all other points
- Sort distances and select k=3 nearest neighbors
- Create undirected edges between p_i and its three closest points
- Store edges canonically as (min_point, max_point) to eliminate duplicates

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.

Design Rationale:

- **Simplified Approach**: $O(n^2)$ distance computation is simpler than $O(n \log n)$ sweep-line algorithms for true Delaunay
- **k=3 Choice**: Ensures average degree of 3 edges per vertex, providing good connectivity without excessive density
- **Edge Sorting**: Processing shorter edges first creates naturally clustered corridor regions before adding long-distance connections

Expected Output: Approximately 300 edges for 100 region points (3 edges × 100 vertices)

Multi-Waypoint A* Pathfinding

The system carves corridors between connected region points using A* pathfinding with custom cost modeling:

A* Cost Function

Total Cost Formula:

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

Where:

- **f(n)**: Total estimated cost to reach goal through node n

- **g(n)**: Actual accumulated cost from start to node n
- **h(n)**: Heuristic estimate of remaining cost from n to goal

Heuristic (Manhattan Distance):

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

Where:

- **h(n)**: Estimated remaining distance
- **x_n, y_n**: Current node's coordinates
- **x_goal, y_goal**: Destination coordinates
- **| |**: Absolute value (makes negative distances positive)

Manhattan distance measures the sum of horizontal and vertical distances, appropriate for grid-based movement restricted to four directions (up, down, left, right).

Tile Traversal Cost:

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

Where:

- **BASE_TILE_COST** = 1.2 (standard movement cost)

- **Δdir** = 0.1 if direction changes from previous move, 0 otherwise
- **Existing PATH bonus** = -0.5 (encourages reusing existing corridors)

Design Rationale:

- **Direction Penalty**: Discourages zigzag patterns, creating straighter, more natural-looking corridors
- **PATH Reuse Bonus**: Promotes interconnected networks instead of isolated parallel corridors
- **Manhattan Heuristic**: Admissible (never overestimates) for 4-connected grids, guaranteeing optimal paths

Waypoint System

To create organic, non-linear corridors, the system inserts intermediate waypoints between start and goal points based on total distance:

Distance Calculation:

$D = |dx| + |dy|$

Where:

- **D**: Total Manhattan distance between start and goal
- **dx**: Horizontal separation (x_goal - x_start)
- **dy**: Vertical separation (y_goal - y_start)
- **| |**: Absolute value

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, system attempts 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

Problem: A* may carve 2×2 blocks of corridor tiles, creating room-like spaces that violate the narrow corridor aesthetic required for poker combat.

Detection: For each position (x,y), check if all four tiles form a PATH block:

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

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.

Dead-End Reduction

Problem: Pathfinding creates dead-ends (tiles with exactly one PATH neighbor), increasing frustrating backtracking.

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

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

$$seed = ((chunkX \times 73856093) \oplus (chunkY \times 19349663) \oplus globalSeed) \,\&\, 0x7FFFFFFF$$

Where:

- **chunkX, chunkY**: Chunk coordinates in world grid
- **73856093, 19349663**: Large prime numbers that ensure good hash distribution (prevent similar inputs from producing similar outputs)
- $\oplus$: Bitwise XOR operation (combines numbers by flipping matching bits)
- **globalSeed**: World-wide seed set at game start

- **& 0x7FFFFFFF**: Bitwise AND operation that masks result to 31-bit positive integer (prevents negative numbers)

Purpose: This formula ensures that:

- Adjacent chunks have uncorrelated seeds (chunks at (0,0) and (1,0) produce completely different layouts)
- Same chunk coordinates always produce same seed (deterministic worlds)
- Different global seeds create different worlds

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:

- **16807**: Multiplier constant (produces well-distributed random sequences)
- **2147483647**: Modulus (largest 31-bit prime number)
- **Result**: Values normalized to range [0, 1)

Design Rationale: Fast computation, minimal memory, sufficient randomness quality for spatial distribution. Same seed always produces 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:

- Tile must be PATH (corridor, not wall)
- Distance from chunk edge ≥ 3 tiles (prevents nodes at chunk boundaries)
- Open neighbors in 3×3 window ≥ 5 tiles (ensures adequate space for combat)

  Minimum Node Spacing:

  $minNodeDistance = chunkSize / 4$

  $For\ 50\times50\ chunks:\ minNodeDistance = 12.5\ tiles$

  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.

Configuration Parameters

Core parameters controlling generation:

Table 7:

Configuration Parameters

Parameter	Default	Range	Purpose
regionCount	100	50-200	Number of corridor anchor points
minRegionDistance	3	2-8	Minimum spacing between anchors
BASE_TILE_COST	1.2	0.8-2.0	A* movement cost
DIRECTION_CHANGE_PENALTY	0.1	0.0-0.5	Cost for turning corners
MIN_WAYPOINT_DISTANCE	8	5-20	Distance threshold for waypoints
BASE_NODE_COUNT	3	1-8	Nodes per chunk
CONNECTION_PROBABILITY	0.7	0.3-1.0	Chance of inter-chunk connection

Alternative Configurations

Three configuration profiles for different gameplay experiences:

Default Configuration: Balanced exploration and combat density

- regionCount: 100, minRegionDistance: 3
- Expected: 35-42% path coverage, 8-12 tile corridors, 12-18ms generation

Dense Configuration: Combat-intensive with frequent encounters

- regionCount: 150 (+50%), minRegionDistance: 2 (-33%), BASE_NODE_COUNT: 5 (+67%)
- Expected: 48-55% path coverage, 5-8 tile corridors, 18-25ms generation

Sparse Configuration: Exploration-focused with rare encounters

- regionCount: 60 (-40%), minRegionDistance: 5 (+67%), BASE_NODE_COUNT: 2 (-33%)
- Expected: 22-28% path coverage, 12-18 tile corridors, 8-12ms generation

Data Collection and Metrics

Session-Level Metrics

The system captures comprehensive generation data:

- **Spatial Quality**: Path coverage percentage, corridor width distribution, dead-end ratio
- **Connectivity**: Graph connectivity ratio (percentage of reachable tiles), loop frequency

- **Performance**: Generation time per chunk, cache hit rate, memory usage
- **Gameplay**: Node accessibility, average distance between nodes, spawn-to-first-node distance

Target Metrics

Table 8: 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

Testing Methodology

Unit Testing

Validation Approach: Generate known configurations and verify expected properties:

- Region points respect minimum distance constraints
- k edges created per vertex (approximately)
- A* finds valid paths between all connected regions
- No 2×2 PATH blocks remain after pruning
- Identical seeds produce identical layouts
- Adjacent chunks have uncorrelated seeds

Integration Testing

Full Pipeline Validation:

- Generate 1,000 complete chunks through all seven stages
- Verify connectivity ratio ≥95% across all chunks
- Measure generation time distribution (mean, P95, max)
- Confirm cache coherence (identical inputs → identical outputs)

Performance Profiling

Benchmarks:

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

Limitations and Considerations

System Limitations

Simplified Triangulation: k-NN approximation 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. System logs warnings and continues with fewer points.

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 methodology, this anticipates:

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