Archigram Tool Document

A spline-driven PCG tool for city-embedded parkour levels

1. What it is

Archigram (`Archigram_3.0`) is a Houdini HDA that turns one or two designer-drawn splines into a complete, playable parkour level — embedded inside a procedurally generated city, populated with enemies, ropes, and progression points, and exported with full instance-path attributes for both Unreal and Unity. All gameplay logic lives in the engine as pre-built modules; Houdini's job is to decide *what goes where* and emit points carrying the right attributes for the engine to instance against.

A designer's interaction with the tool is parametric, not procedural. They draw a spline, tune sliders, paint a difficulty curve, and optionally pin specific module types into specific slots — the system handles every other decision under explicit, inspectable rules.

2. Design philosophy

Three commitments shape every system in the tool.

Designer intent is sovereign. Wherever the system makes a choice, a designer can override it. Module types per slot, difficulty per spline position, enemy density, respawn cadence, deco fade — all expressed as parameters or per-index overrides on the HDA, not buried in code. The procedural layer is a default, never a constraint.

The city is a level, not a backdrop. The parkour path doesn't float through abstract space. It runs along, over, and across a real urban grid with roads that cut between buildings. When the path crosses a road, it has to bridge the gap — and that's where ropes come in. The result is that the city's geometry directly shapes the run's pacing, and the player sees the skyline mid-grapple instead of staring at a corridor.

Engine-agnostic by contract. Every module — parkour, enemy, rope, respawn, goal — is referenced by a single instance-path string set on a Houdini point. The HDA carries parallel Unreal and Unity channels for every module slot (visible in the Engine Settings panel). Adding a new gameplay module never requires rewiring the graph; it requires authoring a blueprint or prefab and pasting its path into one field.

3. Generation pipeline

The level builds in eight stages. Each consumes the previous stage's artifact and adds one new layer of meaning.

Stage	Produces
1. City rectangle	A bounding region around the spline
2. Big blocks + roads	Block grid with road gaps between them
3. Block subdivision	Smaller blocks within each big block
4. Path / deco classification	Ordered PathBlocks along the spline; DecoBlocks elsewhere
5. Box placement	A platform on each PathBlock at spline height
6. Module type + difficulty	Each slot resolved to one of 8 typed, parameterized modules
7. Gameplay overlay	Enemies, ropes, spawn, respawns, goal layered onto the path
8. Engine handoff	Points with instance paths streamed to UE / Unity

Stages 1–3 are pure spatial reasoning. Stage 4 is where the parkour layer takes over. Stages 5–6 produce the typed level. Stage 7 layers gameplay onto the typed level. Stage 8 is the engine contract.

The two-curve design is honored throughout: PathBlocks carry a `splineId` from intersection forward, and every per-spline system (difficulty curve, SOM overrides, rope routing) operates within its own group.

4. Geometry layer

City and blocks

The city is built outward from the spline, not inward toward it. The system computes a bounding rectangle around the spline (scaled by **Land Expansion Scale**), partitions it with a **Road Subdivision Level** that produces the macro-block grid, then subdivides each macro-block again at **Subdivision Level** to produce the small blocks the path will be cut from. **Irregularity** and **Road Irregularity** add controlled noise so the result reads as a city rather than a tile pattern.

The road network falls out of this for free: the gaps between macro-blocks at subdivision time *are* the roads, with their width set by **Road Width**. This is what allows the path to be naturally interrupted — a feature, not a bug.

Path vs. deco

Each small block is intersected against the spline. Blocks that intersect become **PathBlocks**, ordered by the order in which the spline crosses them (this ordering is the foundation of every later system — module connectivity, difficulty curve sampling, respawn cadence, all assume sequential traversal). Everything else becomes a **DecoBlock**.

DecoBlocks are not silent. They're the visible city around the player, and the tool shapes their presence carefully:

- Building heights are randomized between **Min/Max Deco Building Height** with their own seed.
- A vertical fade defined by **Min/Max Deco Building Pos Y** displaces them slightly so the skyline isn't a flat ceiling.
- **Fade Falloff**, **Inner Fade Strength**, and **Outer Fade Strength** dissolve deco buildings as they approach the path. This is the breathing-room mechanism: it keeps the player's immediate environment readable without sacrificing the city silhouette in the middle distance.

Box placement

For each PathBlock, the spline is sampled for its height at that block's location. A platform box is placed on the block at that height. This is what makes vertical level

design possible from a 2D-ish spline: the spline carries elevation, the block carries footprint, and the box reconciles them.

5. Module layer

The eight types

Every PathBlock receives exactly one parkour module on top of its platform. Modules fall into three families.

Jumps — the structural backbone. **Simple Jump** is the baseline obstacle. **Narrow Jump** is the tight-timing variant. **Up Jump** and **Down Jump** chain two jumps together vertically, used to traverse height deltas larger than a single hop.

Slides — the ducking family. **Simple Slide** requires the player to Ctrl-slide under an overhead obstacle. **Slope Slide** angles downward, combining the slide-under read with a downward traversal.

Special — modules with their own pacing. **Bridge** is a narrow single-file walkway, and is the module enemies become most threatening on (forced linear movement plus no room to dodge). **Wallrun** is the high-intensity wall-traversal module, used as a punctuation more than a baseline.

The mix is intentional: jumps form the rhythm, slides break it, special modules spike it.

Type assignment: SOM + connectivity

Each module slot's type is decided by a two-tier system.

The **Sequence of Movement (SOM)** layer is the designer's direct control. In editing mode, every slot displays its index above the generated geometry. The Movements & Difficulty panel exposes one entry per slot per spline (`Movement1_1`, `Movement1_2`, ..., `Movement2_1`, ...), each defaulting to `Default` and overrideable to any of the eight named types. The `Sync Movement Count` button reconciles the override list length with the number of slots the geometry actually produced — a small but important UX detail, because slot count changes whenever the spline or block parameters change.

When a slot is left at `Default`, the **connectivity solver** chooses for it. For each module type, the designer authors a list of allowed successor types (e.g. *wallrun*

cannot follow wallrun; *down jump cannot follow up jump*). The solver also reads the height delta and direction change between adjacent PathBlocks and disqualifies types that don't fit the geometry — a slide can't sit on a block that the previous block jumps up to, for example. From the surviving candidate set, the solver picks pseudorandomly with a designer-controlled seed, with a small stochastic substitution layer that occasionally swaps semantically similar types to prevent over-uniform sequences.

A first-slot exclusion list (`exclude_start`) prevents inappropriate types from appearing at level start, where the player has no momentum context.

The two systems compose cleanly: a designer can pin three or four consequential slots by hand and let the solver fill the connective tissue.

Difficulty: same type, different teeth

Every spline carries its own **difficulty curve** — a 0-to-1 ramp authored against the spline's traversal position, accessible per-spline through the First Curve / Second Curve tabs. The curve is sampled at each PathBlock's normalized position and written as a `difficulty` attribute.

The critical design choice: **difficulty does not pick the module type**. It parameterizes the type that's already been picked. A Simple Jump at curve value 0.2 is a forgiving gap; the same Simple Jump at 0.9 is a punishing one. Each of the eight modules is itself procedural, so width, height, gap, slope, and timing all scale against the curve.

This decoupling is what lets designers shape difficulty independently of layout. Two splines with identical SOM sequences and identical block layouts can play completely differently because their difficulty curves diverge.

6. Gameplay overlay

Once the typed level exists, four systems layer on top of it.

Enemies

Enemies are placed on PathBlocks under a difficulty-weighted sampling rule. The **Enemy Percentage** parameter (0–100) sets the global keep ratio. The sampler

scores each PathBlock as `difficulty + random * enemyRandomness`, sorts descending, and keeps the top fraction.

The behavior this produces matches the design intent: enemies cluster in the harder regions of the curve, with controlled randomness preventing predictable patterns. **Enemy Height** sets a vertical offset so the enemy reads as a threat in the player's path rather than collision-blocking the platform itself.

Enemies behave as fixed-position threats in the Ghostrunner mold — one hit kills the player back to the most recent respawn, and the enemy itself dies in one hit. The most consequential enemy placement is on Bridges, where the corridor gives the player nowhere to break line of sight.

Ropes

The rope system exists to handle two cases the parkour modules can't:

1. **Cross-road gaps.** When two consecutive PathBlocks fall on opposite sides of a city road, no jump module can bridge them — the gap is too wide and the geometry forbids it.
2. **Vertical jumps beyond capability.** When the height delta between consecutive PathBlocks exceeds **Max Player Jump Height**, the player can't reach the next block through the parkour vocabulary.

In both cases, the rope routing system inserts a rope spanning the two PathBlocks: a rope mesh, two anchor balls, and the engine-side rope blueprint that owns the swing physics. **Rope Height** sets the anchor elevation above the platforms; **Rope Thickness** sets the visual mesh diameter.

The design payoff is that ropes are also the moments the player gets a clean view of the city. Because ropes only appear at long horizontal or vertical traversals, the player is mid-air long enough to actually see the skyline that the deco system has been building in the background. The rope system isn't decorative — it converts a topological problem (disconnected path) into a paced visual reward.

Respawn points

Every Nth eligible PathBlock (where N = **Respawn Point Gap**) has its base box swapped for a respawn box. Touching it during a run saves progress; on death, the player resumes there.

“Eligible” is a deliberate filter: respawns cannot land on Bridge, Wallrun, or SlopeSlide modules. These three types don’t provide stable ground — a Bridge demands forward motion, Wallrun is a wall-attached state, and SlopeSlide is a continuous downward traversal. Respawning into any of them would put the player in an unrecoverable state from frame one. The respawn system steps over those slots and uses the next eligible one.

Spawn point and goal

The level start point is recognized by Unreal as a `PlayerStartPos` instance. The goal is a vertical light-pillar mesh placed at the end of the path, doubling as a wayfinding marker visible from a distance.

7. Engine contract

The Engine Settings panel exposes one instance-path string per module type per engine — Player Start, Normal Path Block, Respawn Path Block, Obstacle, Wall Run, Deco Block, Rope, Rope Mesh, Rope Ball, Enemy, Goal — with a complete parallel set for Unreal and Unity.

The contract is deliberately thin. Houdini emits points carrying:
- a transform (position, orientation, scale),
- a `module_name` attribute identifying which type of thing the point represents,
- the engine-specific instance path string,
- type-specific parameters (difficulty, isRespawn, isRope, splineId, etc.).

The engine reads the instance path attribute and instances the corresponding pre-built blueprint or prefab at the transform. All gameplay logic — collision, animation, rope physics, enemy AI, respawn save state — lives engine-side, in the modules themselves.

This is what makes the tool extensible without modifying the graph. A new module type — say, a grappling-hook target or a moving platform — requires only authoring its blueprint, adding a slot for it to the SOM list and the connect rules, and pasting its path into a new Engine Settings field. The PCG graph itself does not change.

8. Designer workflow

The intended use loop is short and parametric.

1. **Draw the spline(s)** in the Houdini scene. One spline is sufficient; two enables parallel routes through the same city.
2. **Tune the geometry** (Geometry tab): set block size and subdivision level, gap ranges, road width, deco fade. The city shape is settled here.
3. **Author the difficulty curve** (Movements & Difficulty tab, per spline): place control points along the 0–1 axis to shape the run's intensity arc.
4. **Pin signature moments via SOM** (same tab): leave most slots on `Default`; override the slots that need to be specific modules (e.g. a wallrun at a known landmark, a bridge over a known road crossing).
5. **Set gameplay density** (Gameplay tab): enemy percentage, respawn gap, rope and jump heights.
6. **Resolve engine paths** (Engine Settings tab): paste the per-engine instance paths once per project. Reusable across levels.
7. **Cook and export.** The HDA outputs to the engine's PCG ingestion path.

Iteration is cheap because every parameter is reactive. A designer can adjust a single difficulty curve point and re-cook to see the run rebalance — the layout doesn't shuffle, but the modules' parameters do. SOM overrides survive geometry regeneration as long as slot count is stable; `Sync Movement Count` resolves the case when it isn't.

9. Extension surface

The system is engineered to grow along three axes.

New module types. Author a blueprint, add a name to `module_names`, define its successor list in `connect_*`, and paste its instance path into Engine Settings (per engine). The connectivity solver will begin selecting it; difficulty parameterization needs to be authored module-side.

New gameplay layers. The enemy / rope / respawn pattern — a sampling or rule-based layer that consumes the typed PathBlock stream and emits its own attributed points — can be replicated for any new system (collectibles, environmental hazards, narrative triggers).

Additional engine targets. The instance-path channel pattern generalizes. A third engine column in Engine Settings would let the same level export to a third runtime without graph changes.

10. Architecture diagrams

Three diagrams accompany this document:

1. **System architecture** — designer inputs flowing through Houdini subsystems into UE / Unity gameplay modules.
2. **Pipeline flow** — the eight stages above as a top-to-bottom sequence with intermediate artifacts.
3. **Module decision logic** — how a single PathBlock slot resolves to a typed, parameterized module via SOM, connectivity, and the difficulty curve.

Appendix A — Parameter reference

Geometry tab

Section	Parameter	Role
Mode	Editing Mode (toggle), Index Display Size	Shows numeric slot indices on geometry for SOM authoring
Basics	Min Block Size	Floor on subdivided block size
Basics	Subdivision Level	How finely each big block is split
Basics	Irregularity	Stochastic perturbation of block layout
Basics	Min Path Block Shared Length	Shared-edge threshold for path adjacency
Basics	Overall Scale	Global multiplier on block dimensions
Building Gaps	Min/Max Gap In X, Min/Max Gap In Z	Randomized shrink ranges between buildings
Land & Road	Land Expansion Scale	How much larger the city rectangle is than the spline bounds
Land & Road	Road Subdivision Level	Macro-block count
Land & Road	Road Irregularity	Noise on the road grid

Section	Parameter	Role
Land & Road	Road Width	Width of the road gaps
Land & Road	Blocks Shrink Scale	Per-block inset from its allotted footprint
Deco Buildings	Min/Max Deco Building Height	Height range for filler buildings
Deco Buildings	Deco Building Seed	Reproducibility seed
Deco Buildings	Min/Max Deco Building Pos Y	Vertical jitter range
Deco Buildings	Fade Falloff	Distance over which deco fade transitions
Deco Buildings	Inner / Outer Fade Strength	Fade intensity near and far from path

Movements & Difficulty tab

Element	Role
First Curve / Second Curve tabs	Per-spline editing context
Difficulty Curve	Ramp editor; control points define difficulty along normalized spline position
Movement Count	Number of SOM slots; reconciled with geometry via Sync Movement Count
Movement{splineId}_{n}	Per-slot type override; `Default` defers to the connectivity solver
Sync Movement Count	Aligns SOM list length with current generated slot count

Gameplay tab

Section	Parameter	Role
Rope	Max Player Jump Height	Height delta threshold above which a rope is required
Rope	Rope Height	Anchor elevation above platforms
Rope	Rope Thickness	Visual mesh diameter
Enemy	Enemy Percentage	Global keep ratio (difficulty-weighted)
Enemy	Enemy Height	Vertical offset of enemy instance above platform

Section	Parameter	Role
Respawn Point	Respawn Point Gap	Spacing in eligible PathBlocks between respawn markers

Engine Settings tab

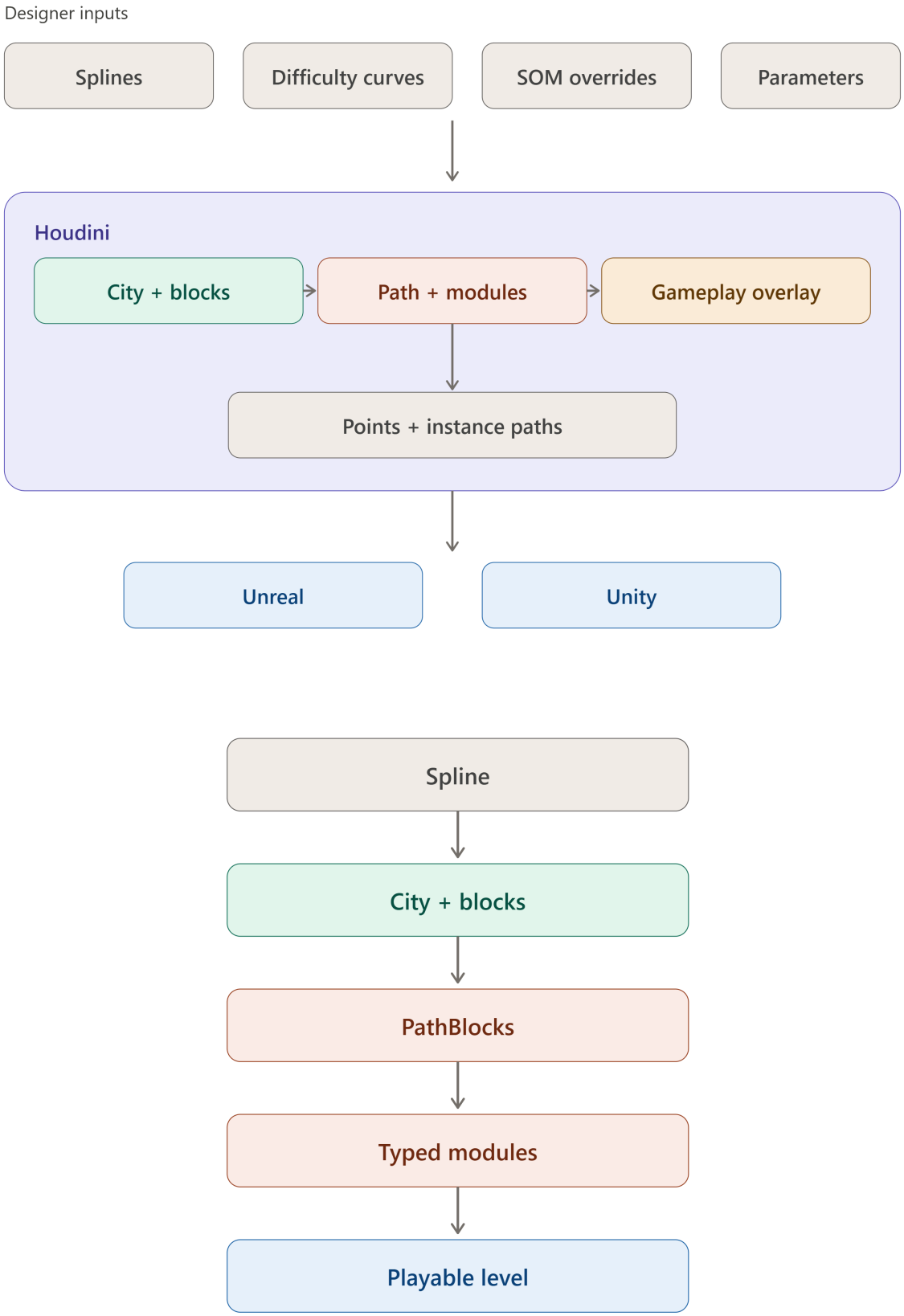
Parallel Unreal and Unity instance-path fields for: Player Start, Normal Path Block, Respawn Path Block, Obstacle, Wall Run, Deco Block, Rope, Rope Mesh, Rope Ball, Enemy, Goal.

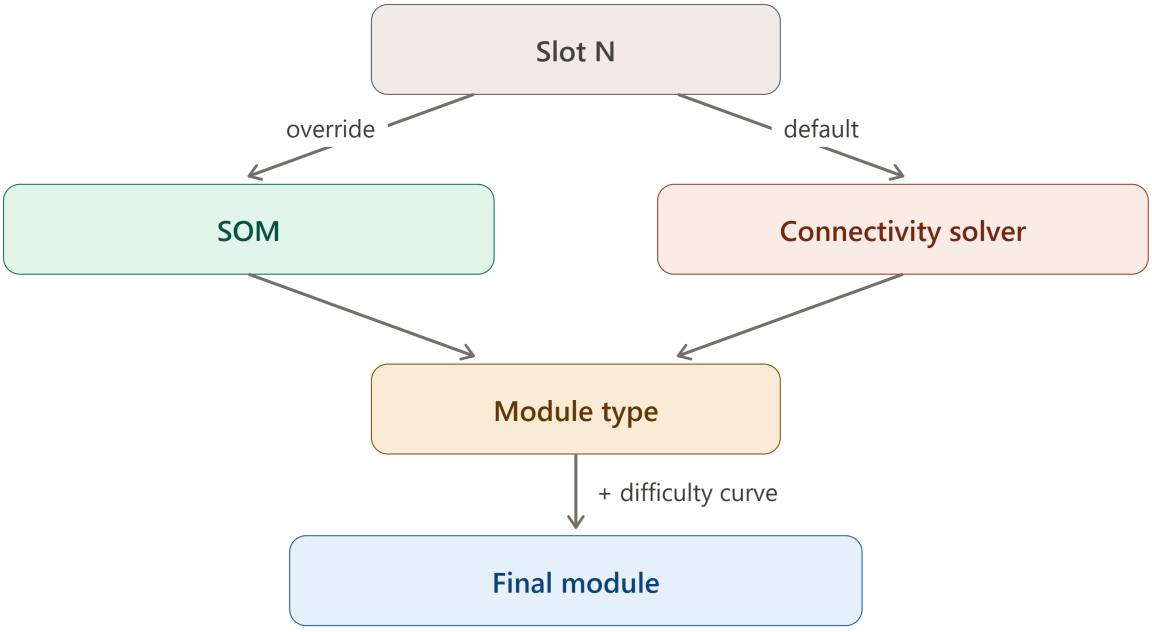
Appendix B — Module type reference

Index	Name	Family	Role
0	Simple Jump	Jump	Baseline obstacle
1	Narrow Jump	Jump	Tight-timing variant
2	Up Jump	Jump	Two consecutive upward hops
3	Down Jump	Jump	Two consecutive downward hops
4	Bridge	Special	Narrow single-file walkway; high enemy threat
5	Wallrun	Special	Wall-traversal punctuation
6	Simple Slide	Slide	Horizontal Ctrl-slide under obstacle
7	Slope Slide	Slide	Angled-downward slide-under

Module indices `{4, 5, 7}` (Bridge, Wallrun, Slope Slide) are excluded from respawn-point assignment.

Appendix C — Tool Diagrams


Designer inputs
Splines
Difficulty curves
SOM overrides
Parameters
Houdini
City + blocks
Path + modules
Gameplay overlay
Points + instance paths
Unreal
Unity
Spline
City + blocks
PathBlocks
Typed modules
Playable level


Slot N
override
default
SOM
Connectivity solver
Module type
+ difficulty curve
Final module