Master Project Report:

A Modular Workflow for Houdini Procedural Assets in Unreal Engine

Xingyue. Jin

Student ID: s5814691

August 2026

Abstract

This project aims to explore how to use Python to control Houdini Digital Assets (HDAs) and design a procedural environment creation workflow in Unreal Engine that is suitable for artists. Although the Houdini Engine supports running HDAs in Unreal Engine, artists still need to manage a large number of assets, expose parameters, and perform recooking operations across various Actors and separate “Details” panels. This process becomes increasingly inefficient when procedural scenes contain multiple asset types, repeated instances, or nested HDAs. To address these issues, this project developed a central Unreal Engine-based tool that allows artists to bind, configure, deploy, and update the required HDAs and StaticMesh resources within a single interface.

The system decouples layout HDAs from their child resources and re-establishes connections through layout data, persistent registry entries, internal snapshots, and controlled update logic. It supports type-based asset binding, shared type-level parameter control, instance generation, and object-level incremental recooking. The system skips unchanged instances, updates reusable characters in place, and submits only affected HDA instances to a controlled, serialized recooking queue. This project demonstrates a complete workflow ranging from layout reading and asset deployment to incremental synchronization and parameter updates. Additionally, the project identifies opportunities for further development in areas such as baking performance, material control, large-scale benchmarking, and automatic feedback on sub-asset sizes.

Content

Contents

1. Introduction

1.1 Project context

In contemporary game scene production, procedural methods are increasingly being used to support the construction of large-scale environments and reduce the costs associated with repetitive manual placement. This shift is closely tied to the practical demands of current game development: projects require larger explorable spaces, higher visual quality, and faster iteration speeds. As a result, scene generation workflows are gradually shifting from a complete reliance on manual construction toward procedural systems that support reuse, variation, and continuous updates.

Against this backdrop, Unreal Engine—thanks to its combination of real-time capabilities and high-quality visual rendering—has become a key platform for bringing procedurally generated scenes into the final presentation and interaction phases. At the same time, Houdini is gradually gaining greater importance in game production workflows due to its strengths in procedural asset creation, rule-driven geometry generation, and the ability to implement complex control over multi-level structures through code. Houdini has long served the film and visual effects industries, and as the gaming industry's demands for environmental complexity and quality continue to rise, these procedural capabilities are becoming increasingly applicable to game scene production. Consequently, the integration between Houdini's procedural asset production and Unreal's real-time scene deployment has become a work area of practical production significance. This project was undertaken precisely against this production backdrop.

1.2 Problem

Although the integration of Houdini and Unreal Engine offers powerful procedural scene-building capabilities, there are still significant usability issues with this workflow in actual environment production. These issues go beyond mere technical implementation challenges; they directly impact the efficiency with which artists view, adjust, and iterate on procedural content within Unreal scenes.

First, within nested HDA scenes, the overhead of recooking and the behavior of updates are highly unpredictable. In a Houdini-centric procedural scene-building workflow—especially when there are multiple levels of HDA nesting—adjustments to outer-layer parameters often trigger a chain reaction of updates to downstream assets. When the dependencies of internal component assets are complex or the number of generated points is high, such parameter modifications can significantly extend “cook” times (Spahr, 2023). In other words, what users perceive as a single localized adjustment may, at the underlying level, trigger extensive rebuilding, thereby slowing down the iteration process and impairing creators’ ability to assess the state of the workflow.

Second, after procedurally generated assets are imported into Unreal, significant management challenges arise in large-scale applications. Although the official Houdini Engine integration exposes HDA parameters within Unreal, these parameters are typically scattered across the “Details” panels of individual Actors and intermingled with a large number of Unreal’s native properties. As the number of HDAs in a scene increases, artists are forced to locate and adjust them one by one, lacking a unified method for batch control and centralized management. More critically, the official integration primarily provides parameter exposure and basic invocation capabilities, but it does not establish a unified management layer for Unreal scenes to address issues such as instance comparison, incremental updates, selective recooking, and stable scheduling. There is a genuine need among practitioners for such a management system (alexsied, 2024).

Although workflow orchestration and batch processing mechanisms such as PDG already exist on the Houdini side, these solutions primarily serve task organization within Houdini itself and cannot directly address the unified scheduling and updating of large numbers of procedural instances in Unreal scenes. Therefore, there remains a practical workflow gap between the completion of procedural asset generation and their efficient iteration and maintenance within Unreal. This project addresses this gap by building a management and update framework for the Unreal side, focusing on solving the challenges of orchestration, synchronization, and stable, batch-based re-baking control for procedural assets.

1.3 Aims and Objectives

1.3.1 Aims

This project aims to build a management and orchestration framework for the Unreal Engine side of the Houdini-to-Unreal Engine procedural scene workflow. The core of the project lies not in the interface itself, but in addressing the underlying workflow challenges—specifically, how procedural assets are identified, configured, synchronized, updated in groups, and reliably re-cooked after being imported into Unreal scenes—thereby providing artists with a Houdini asset control mechanism better suited to real-world production environments.

1.3.2 Objectives

a. Redesign the procedural scene workflow between Houdini and Unreal Engine, changing the original nested HDA workflow and shifting part-level asset configuration and control to the Unreal side. The main body and parts communicate solely through data.

b. Establish a data-driven information processing structure that integrates layout input, configuration conversion from Houdini to Unreal, information flow, and the organization of recook tasks into a single update framework.

c. With the goal of reducing unnecessary recooking, we have developed an incremental synchronization strategy that distinguishes between reusable, transformable, and new scenario instances for separate handling.

d. Establish a consistent generation and assembly workflow for different types of procedural inputs, such as points, lines, and curves. At the same time, support the integration of static meshes such as FBX and UAsset, and create a more efficient workflow.

e. Based on the actual runtime behavior of the Houdini Engine in Unreal, we have introduced a controlled recook sequence, a phased update process, and a post-cook validation mechanism to improve runtime stability.
f. Memory system: The tool will remember the user's changes, store a history, and support rollback.

g. While establishing pipeline specifications, maintain fault tolerance and

controllable customization capabilities in actual production, and build a unified control layer that spans from global management down to component parameters at the type level. Fill the gap in the Artist workflow.

1.4 Contribute

The main contributions of this project can be summarized in the following five points.

First, the project proposes and implements a decoupled collaboration model between the base layout and component assets. Unlike the approach of keeping the base HDA and its child assets within a deeply nested HDA structure, this project reorganizes the relationship between the two into a data-driven workflow tailored for Unreal Engine. In this model, the base layout is responsible for outputting structured layout descriptions, while the binding, parameter configuration, and update control of component assets are handled by an independent management layer on the Unreal side. Consequently, the main body and components no longer rely on direct nested coupling but instead communicate and synchronize via data flows.

Second, the project built a unified orchestration framework on the Unreal side for Houdini-based procedural assets. This framework integrates layout input, asset binding, parameter processing, instance tracking, and update execution into a single management layer, rather than dispersing these processes across the local editing of individual Actors or leaving them entirely within Houdini's production logic.

Third, the project implemented an incremental synchronization mechanism based on snapshots, diffs, and syncs. Unlike approaches that rely on external intermediate files (such as CSV) to save and manage comparison states, this tool maintains an internal "snapshot of the last successfully applied state in the scene" and uses this as the baseline for subsequent incremental comparisons. By comparing new layout inputs against this internal state, the system distinguishes between instances that are reusable, transformable, require replacement, or need to be added, thereby enabling selective execution of scene updates and reducing unnecessary full rebuilds and recooks during the iteration process. Additionally, the update process is continuously documented in the tool's logs, providing greater visibility into synchronization behavior and recook progress throughout the iteration process.

Fourth, the project proposes a set of stabilization recook strategies tailored to the actual runtime characteristics of the Houdini Engine within Unreal. Through serialized cook control, phased update processing, post-cook ensure checks, and settle verification, the system can more effectively address common stability issues encountered when procedurally generated assets are updated within the engine.

Finally, the project builds an artist-facing control layer on top of the aforementioned underlying mechanisms. Its purpose is to expose internal orchestration logic, parameter mechanisms, memory capabilities, and recovery logic in an actionable format. Furthermore, artists can freely search for and link the procedural or static mesh assets they desire using a search bar. In other words, the contribution of this project goes beyond simply creating an interface; it transforms the previously fragile and fragmented procedural integration process into a more structured and actionable Unreal-side scene iteration pipeline.

2 Background

2.1 Existing Workflow and Previous Approaches

The integration of Houdini and Unreal Engine has become an essential procedural workflow in modern game environment production. In this workflow, Houdini typically handles procedural asset creation, rule-driven geometry generation, and the construction of reusable digital assets, while UE serves as the real-time scene environment, handling the deployment, adjustment, and validation of these assets. Through the Houdini Engine API, Houdini Digital Assets (HDAs) can be imported into Unreal, where their enhanced parameters, inputs, and outputs are exposed via plugin interfaces for further editing and recooking

From a practical production perspective, the common procedural workflows in this field can be broadly categorized into three types. The first is an Unreal-centered workflow, in which the procedural scene logic is primarily handled within Unreal Engine. This model focuses more on the organization, distribution, and placement relationships of existing scene assets within the engine environment. The second category is a hybrid workflow, in which Houdini is responsible for generating procedural assets, while Unreal's PCG tools handle placement control within the scene. This approach separates asset generation from scene orchestration, allowing Houdini to focus primarily on procedural construction at the asset level, while Unreal handles composition at the scene level.

The third category is a Houdini-centric workflow. In this model, procedural control applies not only to individual assets but also extends to layout logic, higher-level compositional relationships, and multi-stage connections between scene elements. The completed HDA is then fully imported into UE. At this point, multi-tiered procedural structures can be built by nesting HDAs, organizing base assets, child assets, and spatial logic within a single, larger procedural hierarchy, linked via nodes or code. Compared to the first two approaches, this workflow offers greater logical potential, but its processing speed is relatively slower due to the presence of the nested workflow.

To address the processing speed issue, Houdini introduced the PDG solution, which uses a workflow orchestration mechanism to resolve performance bottlenecks. SideFX describes this capability in Unreal-related documentation via the PDG Asset Link: if a TOP Network is embedded within an HDA, it can be cooked in Unreal, and the corresponding work item output can be imported (SideFX, 2026). However, official staff have also pointed out that such workflows must be explicitly constructed within the PDG network, rather than having external HDAs arbitrarily placed in the

scene automatically incorporated into it (Spahr, 2023).

Therefore, PDG can be understood as a graph-level task orchestration solution within the Houdini ecosystem. In contrast, this project focuses on asset deployment and update orchestration on the Unreal side: the tool generates HDAs, FBX files, and Static Mesh assets—selected at random by artists from the Content library—into the scene in a rule-based and orderly manner according to layout data, and continues to manage the subsequent synchronization, parameter updates, and controlled re-cooking of these instances. The two approaches are not mutually exclusive; rather, they address different levels, priorities, and deployment environments within the pipeline.

2.2 Technical Background

2.2.1 Houdini Public API & HDA

Houdini Digital Assets (HDA) are digital assets in Houdini used to encapsulate procedural logic. An HDA can contain node networks, geometry generation rules, input interfaces, outputs, and exposed parameters. Outputs can be either mesh types or DataTable types. DataTable-type outputs consist of a rowstruct and a datatable, with the former generated before the latter. Through promoted parameters, creators can expose internal controls, allowing the same asset to be reused and adjusted across different scenes. HEU (Houdini Engine for Unreal) provides an API for integrating HDAs into Unreal Engine, enabling HDAs to be instantiated in Unreal, receive inputs, modify parameters, and perform cook or recook operations.

2.2.2 Content & Actor Instance

In Unreal Engine, it is important to distinguish between asset files in the Content Browser and Actor instances in the scene. The HDA and Static Mesh files in the Content folder are the asset sources, while the Actors in the scene are the generated instances. Debugging has confirmed that HDA files can only have their parameters read and possess entities and outputs after they have been correctly instantiated; the output results are saved in the Temp folder within the Content folder.

2.2.3 Wrapper & HAC

In the context of the Houdini API, when an HDA asset is instantiated into an Unreal scene, the wrapper becomes the primary object for interacting with that instance. Tools can use the wrapper to read parameters, trigger events, listen for events, and more. The wrapper also exposes post-processing delegates. Tools can register callback functions (referred to as “handlers” in this project) to these delegates to receive notifications upon completion of a cook. This project utilizes this mechanism to execute post-cook ensure logic, such as rewriting parameters, rebinding curve inputs, and triggering a second recook when necessary.

```
wrapper.on_post_processing_delegate.add_callable(_handler)
```

An Actor represents an HDA instance in a level and is responsible for holding the transform, label, and tag; a “HoudiniAssetComponent” stores the Houdini Engine component state for that instance. The `HAC` is particularly important when reading underlying parameter metadata, which can be accessed via a wrapper. The `HAC` can export T3D.

```
wrapper.get_houdini_asset_component( )
```

Since some Houdini Engine parameter information cannot be reliably obtained through standard property access, this project further integrates HAC, T3D export, and .hda DialogScript parsing. Consequently, scene placement and tag management are applied to Unreal Actors, while parameter writing and recook control are primarily handled by wrappers; however, restoring parameter metadata may require accessing HAC.

2.2.4 .hda DialogScript

Dialog script is a text-based language used to describe parameter interfaces, and it can be used for the parameter interfaces of HDA operators (SideFX, n.d.). It includes the following parameters, which help restore the parameter range and folder/group structure:

```
parm { ... range { ... } }
```

group { ... parm { ... } }

2.2.5 Cook & Recook

Recook() means to reprocess and update the output. At the same time, Houdini officially states that plugins often operate asynchronously. Based on the instability observed during testing, in this project, cook and recook are treated as asynchronous runtime operations. Triggering a recook via the Houdini Engine API does not guarantee that the new results generated after the trigger will be immediately and stably available in Unreal. Therefore, this tool does not rely solely on a single recook call; instead, it combines wrapper status queries, post-cook delegates, and output/geometry detection to determine whether the current HDA has truly completed and entered the `settled` state.

Unavailable or unstable states typically result in the following issues: 1. Failure to complete, resulting in an “invalid ID” error. 2. The Houdini logo appears in the scene instead of an instance. 3. Unable to read parameters exposed by the HDA; matching returns “no match.” 4. The system polls the wrapper’s runtime status.

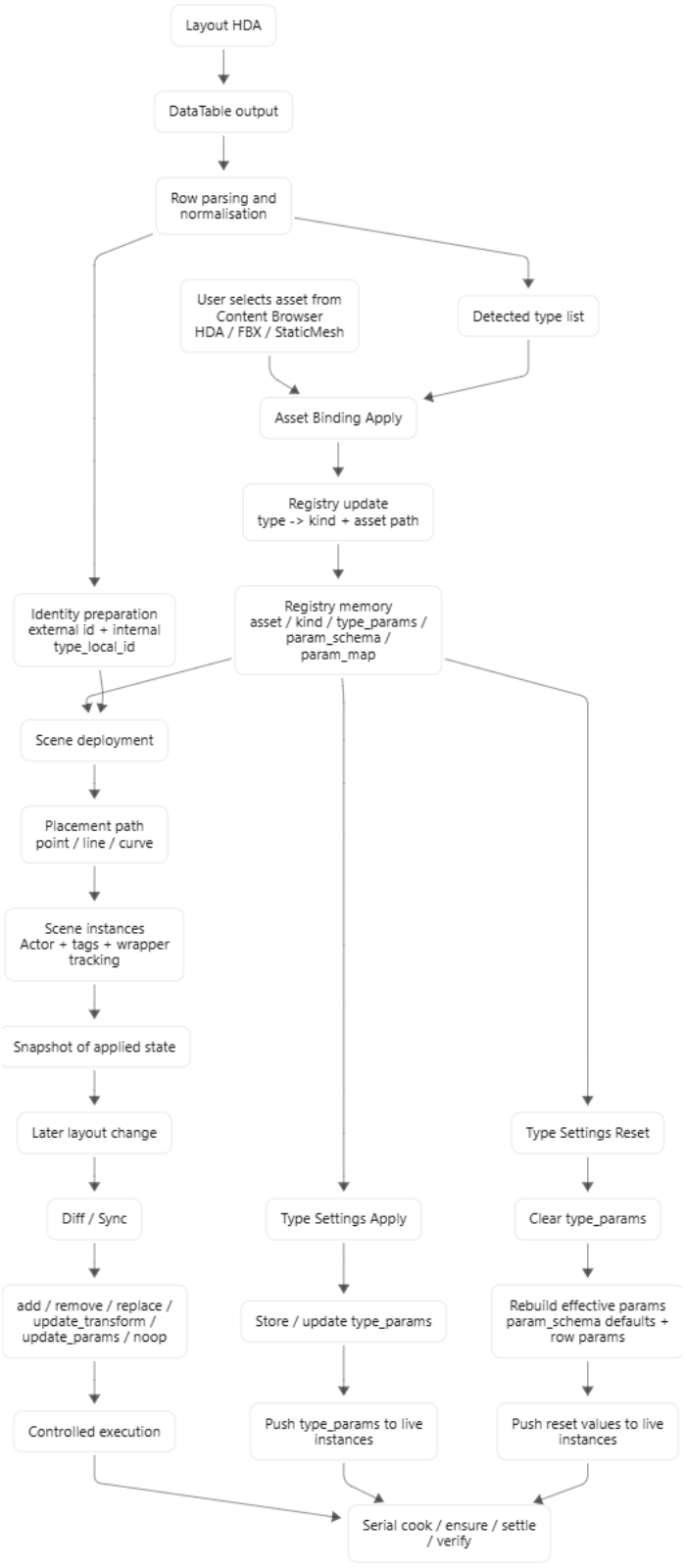
2.2.6 DataTable

An Unreal Engine data table is a structured data asset in which each row follows the same predefined pattern. The Houdini engine for Unreal Engine supports creating data tables based on point cloud output generated from an HDA. In this conversion, each Houdini point becomes a row in the data table, and selected point attributes become columns in the table**. T**o expose a Houdini attribute as a DataTable column, its name must begin with the unreal_data_table_ prefix.

v@unreal_data_table_position = @P;

3 System design and Implementation

3.1 System Overview



Item 1 System

This tool builds a UE-oriented pipeline system designed to handle workflows for generating procedurally generated scenes from Houdini to Unreal Engine. Rather than relying on the creation of nested HDAs in Houdini to directly include and bake all sub-assets into the final product, this system separates the layout generation layer from the asset deployment layer. The Foundation Layout HDA is primarily responsible for generating structured layout data for the scene. The Foundation HDA can be any Layout HDA that meets the specified criteria (see sections 2.2.6 and 3.2 for details). After dragging it into the scene, click on it and bind it in the tool. Once bound, the tool automatically locates the layout data output by the Base HDA. The tool then reads this data within Unreal and, based on that data, deploys, updates, and controls the HDA or Static Mesh (fbx, uasset, etc.) assets—which the user has searched for and selected within this tool—in the Content Browser. Consequently, base assets and child assets are no longer directly linked through nested Houdini files but communicate via data streams.

The main process begins when the tool completes the binding via code—specifically, when it locates the DataTable containing the output data from the Layout HDA. The tool retrieves the DataTable and normalizes each row into a procedural instance description, which includes type, identity, placement, yaw, and params containing various parameters. Before actually generating a scene instance, each identified “type” must first be linked to an Unreal Content asset. Users can select the HDA or Static Mesh they wish to link to that “type” within the tool and click “Apply.” This action writes the asset type and asset path to the registry. Subsequently, whenever a row for the same “type” within the same layout appears, the system can automatically retrieve the corresponding asset from the registry without requiring the user to reselect it.

The Registry serves as the system-wide persistence and memory layer. For each type, once an artist has searched for and selected the assets to be linked, the Registry records the linked asset path, asset type, type-level parameters, parameter schema, parameter mappings, and other configurations. Through this Registry layer, abstract types in layout rows can be resolved into specific Unreal Content assets. Subsequently, the instance generation and synchronization module uses these resolution results to generate the corresponding assets at the correct locations within the Unreal scene, or to reuse, replace, and adjust existing instances during subsequent updates. The generated Actors are tracked via tags, wrapper references, and snapshots so that the system can identify them during subsequent

synchronization processes. Additionally, assigning tags to the generated Actors ensures that they can be accurately located even in the event of unexpected snapshot errors or tool crashes.

After a scenario is deployed, the system maintains an internal snapshot to record the state of the layout that was last successfully applied to the scenario. When the layout HDA is modified and re-cooked, the new DataTable does not immediately trigger a full rebuild; instead, it is compared against the snapshot. The comparison results are categorized into different operations, such as add, remove, replace, update_transform, update_params, and noop. In this way, reusable instances can be retained directly, with parameters moved or updated, while only truly new, deleted, or incompatible instances need to be regenerated. This snapshot-based update logic transforms the process from a one-time generation into incremental synchronization, and this incremental synchronization identifies, at the object level, which objects actually need to be cooked.

The system also includes a separate type-level parameter control window, allowing users to fine-tune the parameters of individual asset types while adjusting global layout variables. When a user modifies the settings for a specific part, these values are written to type_params in the registry and pushed to all live instances of that type in the scene. The final parameters are derived by merging the row parameters with type_params, with type_params taking precedence. However, this propagation follows the principle of incremental synchronization; if an instance of that type does not require changes, it will be skipped. The tool also provides a “reset” option: resetting clears the type-level overrides and reconstructs valid parameters based on the default values stored in param_schema and the original row parameters. Both “Apply” and “Reset” can trigger a controlled recook only for affected instances, rather than forcing a rebuild of the entire scene. This tool delegates HDA and FBX updates to a dedicated system within the tool responsible for task scheduling. The system combines wrapper status checks, post-cook or post-processing delegates, output validation, and ensure logic to determine whether the current HDA is truly complete and stable before releasing the next task. Type-level cook updates, however, include an additional “saw_busy” check. For updates targeting FBX, unnecessary checks are removed to create a faster branching logic.

3.2 Pipeline Rules and Fault Tolerance

3.2.1 Rules

These rules are not intended to compensate for limitations in the tools, but rather to reduce ambiguity between the data generated by Houdini and the execution logic on the Unreal side. In a procedural workflow, layout data, asset bindings, parameter naming, and identity updates must be understood by both artists and the tools. Therefore, this project does not require artists to create assets in a single, rigid manner, but rather defines a set of necessary production conventions and provides fallback logic for common edge cases.

A. Layout Parameter Naming Requirements

For Layout HDA parameters, the tool uses prefix-based filtering rules. Parameters intended to appear in the top layout panel should use the “euw_” or “EUW_” prefix. This rule is in place because the Houdini Engine exposes a large number of internal parameters or parameters unrelated to the current tool, whereas the tool actually needs to present only a small number of controllable items to the artist. Rather than requiring users to manually select useful parameters from a complex list, the prefix rule provides a lightweight whitelist mechanism. In practice, the parameter-reading logic filters Houdini parameters based on the configuration rules and exposes only the relevant parameters to the UI.

B. Component (type_level) Naming Requirements

Regarding linking layout data to a Component, this rule is not mandatory; the tool automatically performs semantic mapping for players (3.2.2). However, if an artist wants a specific child HDA parameter to appear in the type settings panel, the recommended approach is to place that parameter in the EUW folder within the HDA parameter interface. If folder information cannot be reliably recovered, the tool will fall back to parameter name prefix matching—that is, it will identify parameters beginning with euw_ or EUW_. This design allows well-structured assets to retain their original folder groupings while also supporting simple assets that merely follow the naming convention. In practice, the tool reads the child_param_folders setting in the configuration, attempts to recover the parameter folder metadata, and falls back to prefix matching when folder information is incomplete.

C. DataTable

Layout output adheres to a fixed DataTable structure, and the core columns

should be as complete as possible: type, id, and position are the core semantic fields, while placement, yaw, and params are handled by default or for compatibility.

Essential rows' name	Details/example	How to create in houdini VEX
Id	Assign a uniform ID to all lines in the output	i@unreal_data_table_id = @ptnum;
type	Kind. For example, "building", "grass", etc.	s@type = "lamp";
placement	Point / Line (curve)	s@unreal_data_table_placement = "point";
position	The position of the point. When the tool is linked to the hda component, this is the pivot position of the hda component.	v@unreal_data_table_position = @P;
yaw	yaw: Value for rotation in the	float yaw_rad = atan2(forward.z, forward.x);

	horizontal plane	

params	Parameters to be transmitted to the component. only include the parameters that need to be overridden for this type.	dict params = {};
		params["lamp_height"] = chf("lamp_height");
		
		s@specific_params_json = json_dumps(params, 2);

3.2.2 Fault Tolerance

A. Placement Tolerance

The tool supports three types of placement semantics: point, line, and curve.

User Input	Tool Processing Results
curve, or a line that satisfies the curve criteria	curve
line, segment, edge, spline	line
point , pt, vertex	point
No placement but type = “pole”	line
No placement but type = “road”	line
Other situations	Point (default)

For curve data, the tool provides two compatible approaches: if curve points are already specified as an array in “params” entry, the system will directly convert them into a curve instance; if an artist uses multiple rows to represent a single curve, the tool will use curve folding logic to collapse these control points into a unified “curve row”. This way, subsequent generation modules only need to process the unified curve representation.

B. Parameter Name Mapping

Parameter mapping includes both default mappings and user-defined extensions. By default, parameter names in the “params” field of a DataTable are matched against potential HDA parameter names, including both versions prefixed with “euw_” and the original parameter names. For example, “height” in “params” will automatically be matched by default to either the “euw_height” or “height” parameter in a child HDA. If a specific asset was not created according to the default naming convention, users can add a “param_map” in the registry. This does not require rebuilding the HDA; rather, it provides candidate names that the extension tool will attempt when writing parameters. An example is shown below:

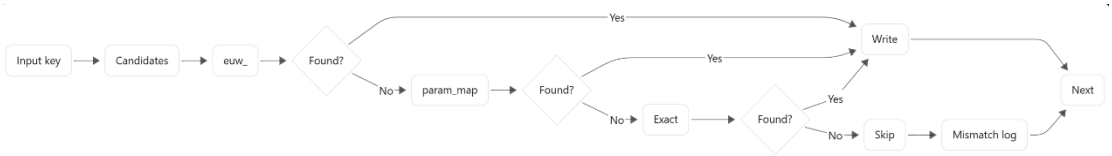
```
"param_map": {
  "height": "euw_custom_height",
  "floor": ["euw_building_floor", "euw_floor", "euw_floor_num"],
  "floors": ["euw_building_floor", "euw_floor", "euw_floor_num"],
  "floor_num": ["euw_building_floor", "euw_floor", "euw_floor_num"]
}
```

Item 2 Param_map example

If a type has a key with the same name explicitly configured in `param_map`, the added entry will be included or used to extend the list of candidate names. The value of `param_map` can be either a string or an array of strings. The code will combine the default name with the values in `param_map`. Ultimately, the semantic mapping within the code will become:

“height” :[“euw_height” ,” euw_custom_height” ,” height”]

The system will sequentially try multiple HDA parameter names, match the first existing parameter, and write the data to it. The order of priority is as follows: first, parameters with the “euw_” prefix; second, parameters defined in “param_map”; and finally, parameters with exactly the same name. If none of the candidate names match, the tool will skip writing to that parameter, log a “mismatch,” and continue processing the rest of the instance.



C. Static Mesh parameters

Static Mesh assets use a separate, lightweight parameter path. Since these assets do not expose Houdini parameters and do not require Houdini recooking, the tool provides a built-in parameter schema for them to display and handle common controls (such as scale and color). However, the value of `height` in the params can also be interpreted as a scale adjustment within the mesh path. This allows non-HDA assets to be integrated into the same registry and layout workflow while avoiding unnecessary Houdini-specific processing steps. These rules, together with fallback behaviors, define the balance between pipeline standardization and artist flexibility. The tool still requires layout data to follow a minimum structural specification so that the system can reliably interpret and execute it; however, it does not force every asset to be created in exactly the same way. When data is missing, parameters cannot be mapped, or certain elements are incompatible, the system prioritizes using default values, skipping individual parameter writes, and issuing warnings and log messages, rather than immediately halting the entire process. This brings the tool closer to real-world production environments, as procedural assets are often created by different artists, and naming, grouping, and output conventions cannot always be completely consistent.

3.3 Layout input layer

The Layout input layer is the first stage in the system to actually enter the execution phase. Its role is to convert the output of the Layout HDA into a stable internal data structure within the tool, ensuring that subsequent modules—such as the registry, instance generation, synchronization, and cook control—can all use the same row format. In this project, the Layout HDA no longer directly contains the final child assets; instead, it outputs a DataTable that describes the scene layout. This DataTable specifies where assets of different types should be generated and which parameters need to be passed to them. In this way, the base layout logic remains within Houdini, while asset deployment and update control are transferred to the Unreal side.

First, the user must bind the HDA they wish to use as the layout. The binding step is implemented as an explicit connection between the editor-selected Houdini actor and the tool’s internal layout source. When the user clicks “Bind,” the panel calls the layout binding function using the actor currently selected in the Unreal level. The code first checks whether the actor is a Houdini Asset Actor, then saves it as the current active layout actor. There are ready-made APIs for this step, and the Blueprint system also provides pre-wrapped nodes. In code, on_bind_selection()

calls “set_layout_from_selection()”, while set_layout_actor() stores the active layout actor and invalidates the cached wrapper.

```python
def set_layout_actor(actor: unreal.Actor) -> unreal.Actor:
    global _g_layout_actor, _g_wrapper
    if not _is_houdini_actor(actor):
        raise RuntimeError(f"Not a Houdini asset actor: {actor.get_name()} ({actor.get_class
    _g_layout_actor = actor
    _g_wrapper = None
    unreal.log(f"[euw_recook] layout actor set: {actor.get_actor_label()} ({actor.get_path_n
    return actor

def set_layout_from_selection() -> unreal.Actor:
    """
    Use the currently selected actor in the editor as Layout HDA.

    """
    subsystem = unreal.get_editor_subsystem(unreal.EditorActorSubsystem)
    selected = list(subsystem.get_selected_level_actors())
    if not selected:
        raise RuntimeError("No actor selected. Select the Layout HoudiniAssetActor in the le
    actor = selected[0]
    return set_layout_actor(actor)

def try_bind_from_selection() -> Optional[unreal.Actor]:
    """
    If a HoudiniAssetActor is selected, bind it;
    otherwise return None (no raise).

    """
    subsystem = unreal.get_editor_subsystem(unreal.EditorActorSubsystem)
    selected = list(subsystem.get_selected_level_actors())
    for actor in selected:
        if _is_houdini_actor(actor):
            return set_layout_actor(actor)
    return None
```

Item 3 3.3

When a user binds a Layout HDA in the tool, the system determines the currently used layout source. The tool then locates the DataTable generated by that Layout HDA and reads its contents using Unreal’s DataTable access functions. The parser reads the row names and expected columns. Each DataTable row is converted into an internal row dictionary. Regardless of whether the target asset is ultimately an HDA or a Static Mesh, subsequent modules will use this unified row format. During the parsing process, the tool also converts spatial data from Houdini’s layout context to Unreal’s scene space. The `position` column is parsed, and the coordinates are transformed based on the position scale and axis conversion specified in the configuration, after which the world transform of the bound Layout Actor is applied.

In addition to the placement processing mentioned earlier, identity information (IDs) is

also prepared for subsequent synchronization. The tool requires users to provide a stable global ID and saves the mapping between the global ID and the corresponding types. However, relying solely on the global ID does not allow for independent re-cooking by type. This is because the addition or deletion of points of other types affects the number of rows, which directly impacts the mapping between IDs and types. If subsequent snapshots were to perform diffing and re-cooking based on this ID, serious errors would occur. Therefore, the tool generates an additional type-local identity during the normalization phase. The original ID is stored in `source_id`, while `type_local_id` is generated based on the row order within the same type. The final internal ID consists of the type and the type-local index (lamp_1). This identity strategy allows snapshots and diffs to operate on a more stable internal ID, reducing cross-type interference caused by fluctuations in row counts. It also provides a clearer basis for locating instances based on type-level settings: when a user modifies the parameters of a specific type, the tool can use the type tag, snapshot records, and internal IDs to locate the live instances under that type and push the parameters only to those instances. For instances where the parameter values themselves have not changed, the parameter write phase skips the unchanged values, thereby avoiding unnecessary parameter writes and subsequent recooking.

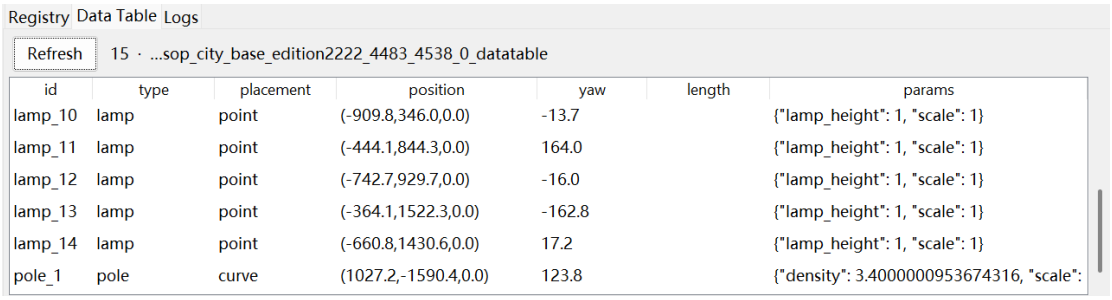
```
item["source_id"] = str(row.get("id") or "")

item["type_local_id"] = str(idx)

item["id"] = f"{row_type}_{idx}"
```

Data Table × Data Table Deta...

Search

	Row Nam	id	position	params	type	yaw	placement
1	Point_0	0	{"X": 4.1605210304260254, "Y": 0, "Z": -14.362462997436523}	{"lamp_height": 1, "scale": 1}	lamp	-123.876869	point
2	Point_1	1	{"X": 2.4292056560516357, "Y": 0, "Z": -16.941181182861328}	{"lamp_height": 1, "scale": 1}	lamp	56.123138	point
3	Point_2	2	{"X": 1.3456172943115234, "Y": 0, "Z": -12.427654266357422}	{"lamp_height": 1, "scale": 1}	lamp	-125.235405	point
4	Point_3	3	{"X": -0.44634935259819031, "Y": 0, "Z": -14.964599609375}	{"lamp_height": 1, "scale": 1}	lamp	54.764603	point
5	Point_4	4	{"X": -1.3520902395248413, "Y": 0, "Z": -10.448305130004883}	{"lamp_height": 1, "scale": 1}	lamp	-127.551521	point
6	Point_5	5	{"X": -3.2451179027557373, "Y": 0, "Z": -12.910758972167969}	{"lamp_height": 1, "scale": 1}	lamp	52.448490	point
7	Point_6	6	{"X": -3.9032418727874756, "Y": 0, "Z": -8.3372459411621094}	{"lamp_height": 1, "scale": 1}	lamp	-132.409988	point
8	Point_7	7	{"X": -5.9980239868164062, "Y": 0, "Z": -10.630523681640625}	{"lamp_height": 1, "scale": 1}	lamp	47.590031	point
9	Point_8	8	{"X": -5.9304227828979492, "Y": 0, "Z": -6.0904321670532227}	{"lamp_height": 1, "scale": 1}	lamp	-146.938995	point
10	Point_9	9	{"X": -8.5335302352905273, "Y": 0, "Z": -7.7848539352416992}	{"lamp_height": 1, "scale": 1}	lamp	33.061016	point
11	Point_10	10	{"X": -6.724726676940918, "Y": 0, "Z": -3.5592823028564453}	{"lamp_height": 1, "scale": 1}	lamp	-174.254807	point

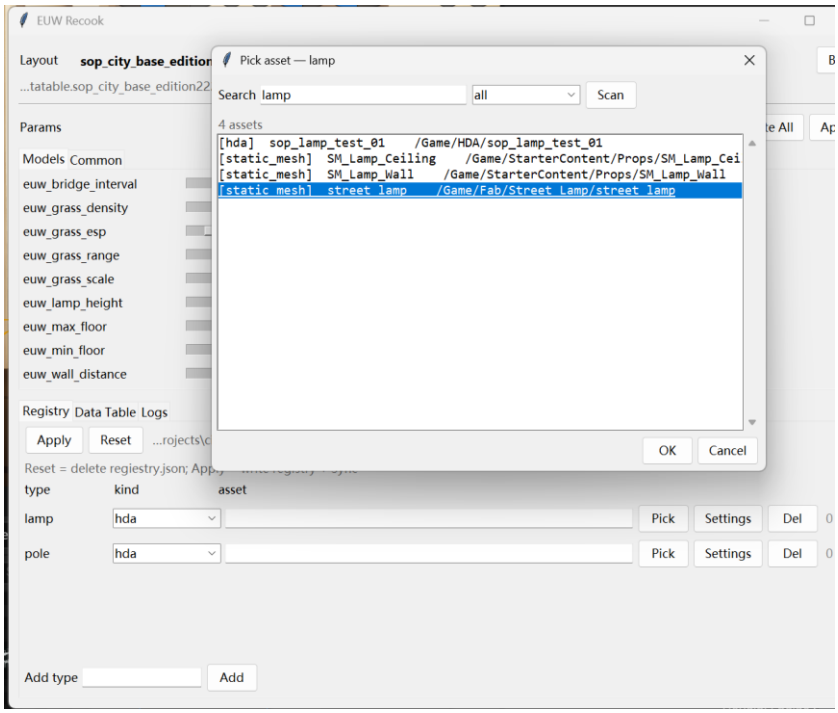
Item 4 datatable



Item 5 This project's UI interface

3.4 Registry and Asset Binding

Once the layout rows have been normalized, the system still needs to resolve the abstract `type` in each row into a specific Unreal asset. The DataTable only describes semantic categories; it does not directly determine which asset in the Content should be used for each category. Therefore, this project introduces a registry layer between the layout input layer and the scene execution layer. The registry is a JSON-based persistent configuration file that serves as both a memory cache and an asset preference profile. When the tool reads the current layout rows, it extracts the type names that appear and creates a placeholder entry for any type not yet present in the registry. This makes it possible to display which asset categories have not yet been configured. Users can then search for assets in the tool's asset rotator, select an HDA or Static Mesh asset for each type, and click "Apply." This action writes the selected asset paths and asset types back to the registry, enabling subsequent instances of the same layout type to be automatically resolved to the corresponding assets.



Item 6 Pick component

Item 7 Line hda and point fbx 's registry after apply

Each registry entry stores more than just an asset path. It also records the asset kind, the selected Content asset, type-level parameter overrides, parameter schemas, parameter mappings, and asset-related configurations such as the curve input index and forward-axis. Therefore, the registry serves both as an asset binding table and a configuration storage layer. When a project is reopened or the tool is refreshed, previously selected assets and type settings can be reloaded without the user having to manually recreate these settings. The registry also plays a role in determining subsequent updates. When a snapshot is created, the system records an `asset_key` for each row based on the registry binding. When a subsequent sync

compares new layout rows with the snapshot, existing instances can only be reused if the internal ID, type, and asset binding remain compatible. For example, if the user's selected Content asset has changed and the old Actor is no longer used, the diff engine will classify that row as a replace.

```
new_asset = snap_mod.registry_asset_key(str(row.get("type") or ""), cfg)

if old_asset != new_asset:

ops.append(DiffOp(op="replace", row_id=row_id, row=row, old=old))
```

Therefore, the Layout HDA does not need to know which child assets are ultimately used, nor do the child assets need to be nested within the Layout HDA. The Registry allows Unreal-side tools to resolve abstract row types into specific assets, remember user selections, and maintain sufficient identity information to support safe incremental updates.

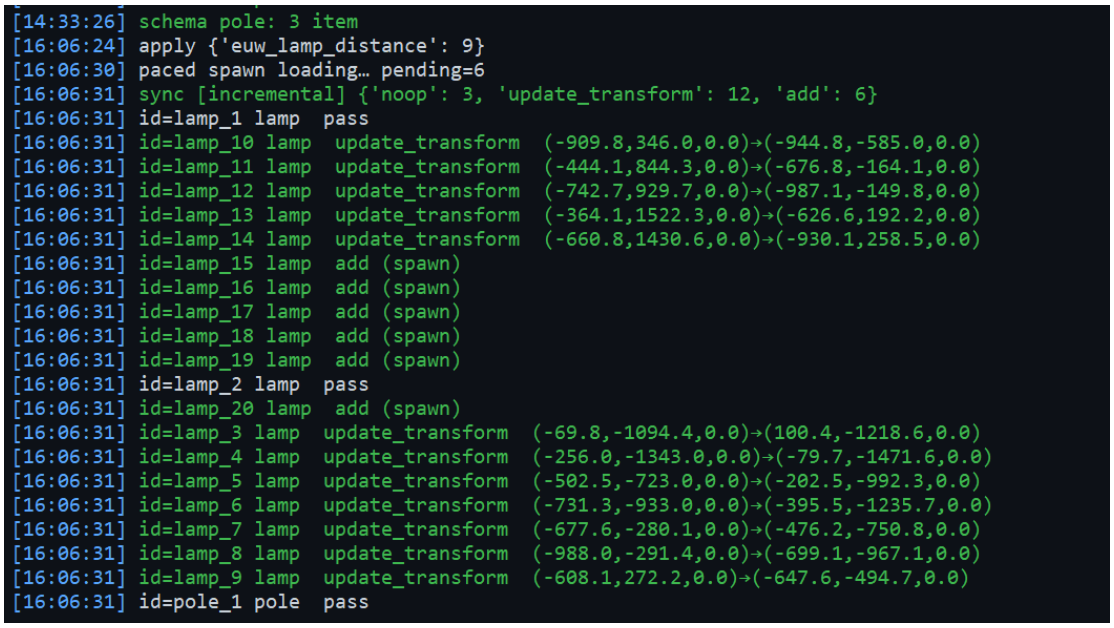
3.5 Diff Layer - Incremental Sync

One of the purposes of this tool is to address the issue of processing overhead caused by nested HDAs. Therefore, a core objective of this tool is to avoid rebuilding the entire scene every time the layout changes and to minimize the number of rebuilds triggered as much as possible. Whether modifying parameters in the base layout HDA or making changes to child HDAs in the Component settings, this does not necessarily mean that all instances in the scene become invalid. If the tool were to delete and regenerate all instances every time, the cost of iteration would remain high, and the advantage of decoupling the layout layer from the asset layer would be diminished. Consequently, this project treats layout updates as a state comparison problem rather than a one-time generation problem. Of course, the first time this tool is used, a full rebuild is performed. The system maintains an internal snapshot to record the layout state last successfully applied to the scene; this is similar to a `dic[]`. After a full rebuild or a successful incremental sync, each normalized row is converted into a snapshot record.

```
_applied_rows = {str(r["id"]): row_to_snapshot(r, cfg) for r in rows}
```

A snapshot records the row ID, type, placement mode, transform-related values, parameters, and the asset key obtained from the registry. The saved snapshot serves as the comparison baseline for the next update. Unlike an external CSV file (the format in which the DataTable is directly saved is CSV), it is a runtime state record maintained internally by the tool, used to compare the differences between the currently generated layout and the state of the previous known scenario.
Take a layout change as an example. When the Layout HDA is modified and the DataTable is re-read, the new rows are compared against the existing snapshot. The diff engine converts the comparison results into executable operations and performs the corresponding actions (add, remove, replace, update_transform, update_params, or noop) based on the changes detected.

If a row appears in the new layout but does not exist in the snapshot, it is considered an “add”;
If a row exists in the snapshot but no longer appears in the current layout, it is considered a “remove”;
If the type, placement mode, or asset binding changes, the instance is considered a “replace”;
If only the position changes, the existing Actor can be reused; this does not trigger a cook operation, and the changes are applied via update_transform.
If only the parameter data changes, the new parameters are written via update_params. The instance can be reused; changes are applied by writing the new parameters only to the HDA wrapper or Static Mesh actor. If there are no relevant changes, the row is classified as a noop and is passed through directly.



Item 8 Tool UI logs

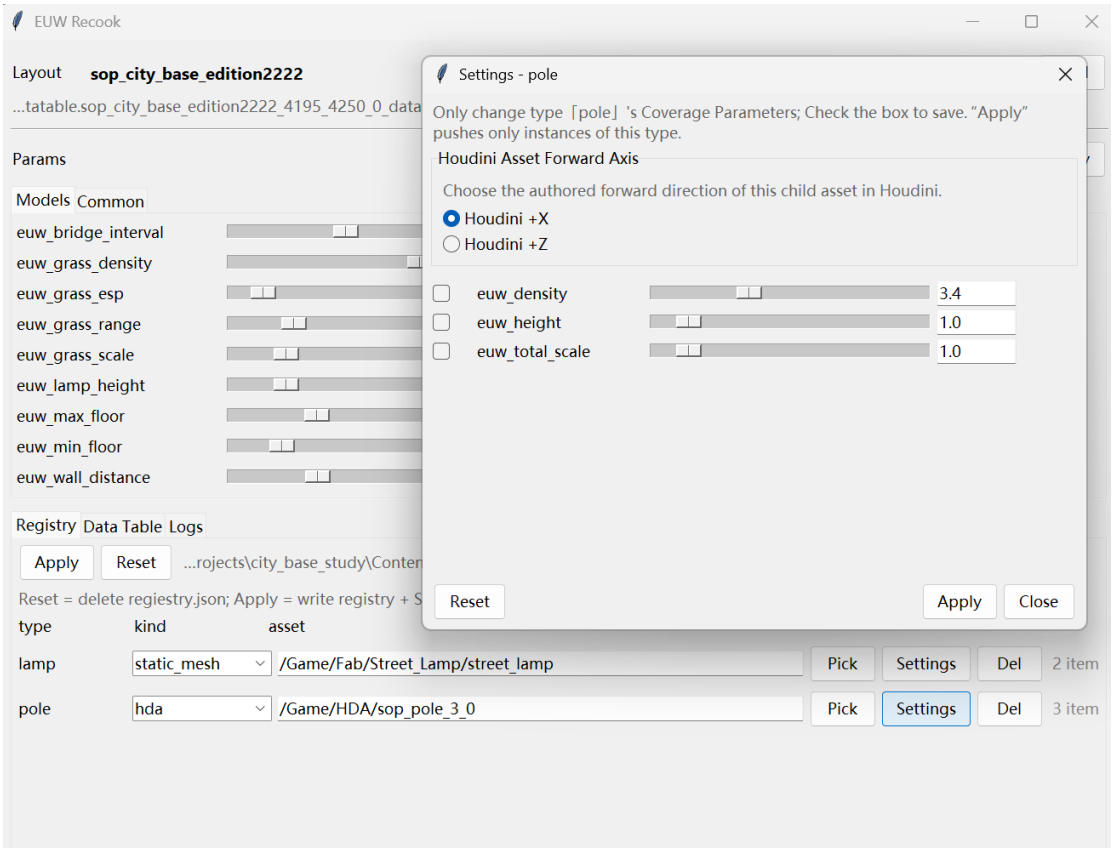
Users can see the internal decision-making process behind updates and distinguish whether the current operation is a high-cost rebuild or a selective incremental update. The image shows the tool’s response when adjusting the density of lights in the layout. As you can see, most lights only undergo a positional offset.

At the same time, the Sync layer has its own protection mechanism. A new snapshot is committed only after the operation has completed successfully and no errors have occurred.

```
if exec_result.get("errors", 0) == 0:

snap_mod.commit_snapshot(rows, cfg=cfg)
```

If an error occurs during execution, the old snapshot is retained to prevent the tool from recording a state that was not actually successfully applied to the scenario. When no snapshot exists, the user executes “Delete All,” or no traceable managed actors are found in the scenario, the tool falls back to the full rebuild path. In this way, the tool uses incremental sync as the default mode while retaining a recovery path.

3.6 Type level Component Control



Item 9 Type parameter UI

In the “Registry” window, click the ‘Settings’ button to open the “Type-Level Component Control” window. For HDA assets, the primary source of parameters is the Houdini Public API wrapper. Once an HDA is successfully instantiated, the wrapper can expose its parameter tuples. However, this information alone is insufficient to generate an artist-friendly control panel. In practical testing, UI metadata—such as slider ranges, labels, folder groupings, and parent-child parameter hierarchies—cannot always be reliably obtained through the wrapper’s standard access methods. To obtain this data and complete the creation of type-level windows, this tool exports HAC to T3D text, parses its contents, and extracts fields related to metadata. For example, the found UIMin and UIMax values can be used to reconstruct parameter ranges, thereby creating parameter sliders.

```
ranges = read_from_hac_t3d()

if not ranges:

    ranges = read_from_actor_t3d()
```

```
if not ranges:
    ranges = read_from_hda_dialogscript()
```

The UI for this window is managed by the registry's param_schema. After the metadata collection described above is successful, the tool creates a schema for the linked assets in a standard format. For HDA assets, the schema is sampled from the live instance in the scene and then written back to the registry. For Static Meshes, the tool uses a standard schema edited by the author, providing controls for scale and color. Therefore, param_schema can dynamically generate type settings controls, rather than manually coding a fixed UI for each asset type.

Users select the parameters they wish to change by checking the corresponding boxes and click "Apply" to sync; the modified values are then stored in the registry as type_params. After deployment, when the user clicks "Reset," the tool clears these overridden values and reconstructs valid parameters based on the default values recorded in param_schema and the original row parameters. It first retrieves the recorded values for each parameter from param_schema to generate defaults, then uses the row parameters to override those defaults.

3.7 Spawn layer

After the registry completes the resolution of assets, the abstract "type" is already bound to the specific assets within Content Folder. The next step is instantiation—that is, converting rows in the data table into instances in the scene. Given the differences between HDA and static meshes, different operations are performed for each at this stage, with the latter being significantly lighter than the former.

For HDAs, this tool uses the asset path stored in the registry to locate the asset's path within the Content, then calls the Houdini API to instantiate it. However, this tool disables the API's auto_recook option, allowing it to independently control parameter writing, input binding, and the timing of the recook. The wrapper returned by the API is saved and associated with the row ID, and the Houdini Asset Actor corresponding to the wrapper is used as an Actor in the scene.

```
wrapper = api.instantiate_asset(hda_asset, transform, enable_auto_cook=False)
```

When linking data to an instance, the "placement" determines how the row is used:

1. A "Point row" represents a single asset instance placed at a specified position

with a yaw value.

2. Line row uses position, yaw, and length to calculate linear placement. The logic is that each point records the direction and distance it should extend. However, in practice, most linearly programmed assets are better suited to being driven by curve input. Therefore, line placement is primarily retained in this project as a compatibility path. All lines used transition to a curve once they meet the criteria. They are spawned using “curve” as the placement identity.
3. The “Curve Row” tool can collect point groups; after processing and analysis, the curve points are converted into a Houdini curve input object and written to the specified input index of the HDA wrapper. This allows the child HDA to regenerate itself based on the curve described by the layout data, thereby enabling the configuration of programmable assets as they currently exist.

Once an instance is generated, the tool sets a label and tags for the generated Actor. Actors managed by the tool are assigned a global managed tag, a type tag, and an ID tag. Even if the snapshot or wrapper cache in memory is incomplete, the system can use these tags to retrieve and update the instance in the scene. For HDA assets, the wrapper is also saved based on the row ID, allowing subsequent parameter updates and recook tasks to locate the correct Houdini instance.

```
type_tag = f"EUW_TYPE_{row.get('type', 'unknown')}"

tags = [str(t) for t in actor.tags]

for t in (managed_tag, id_tag(str(row.get("id"))), type_tag):

    if t not in tags:

        tags.append(t)

actor.tags = tags
```

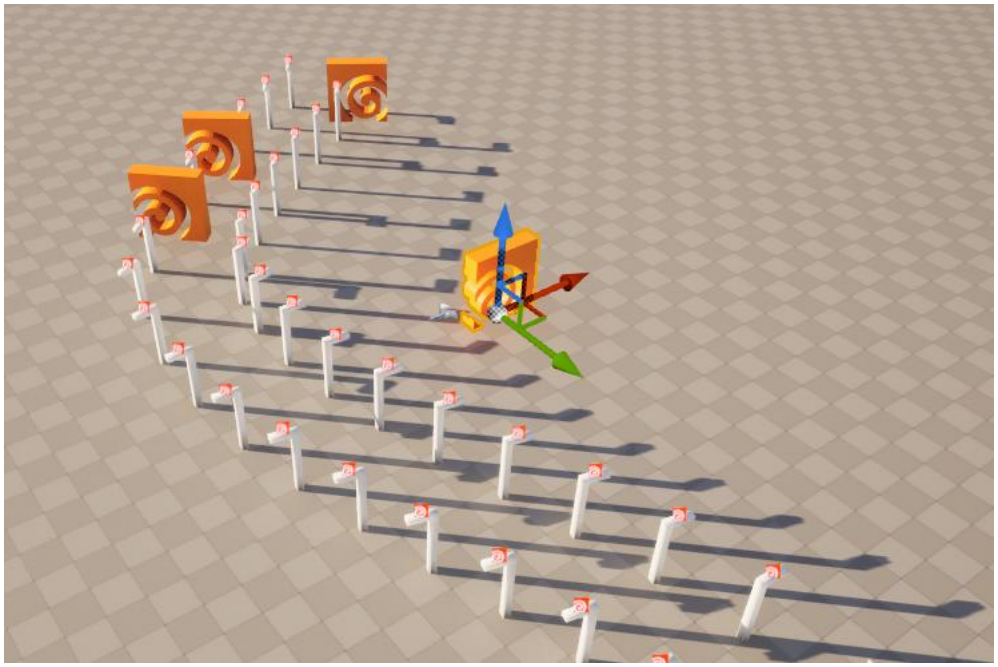
3.8 Cook serial

As mentioned above, the Houdini team considers many of its API operations to be asynchronous, and testing confirms this to be the case. When an HDA is instantiated or “recooked” via the Houdini Engine, a return from a Python call does not

necessarily mean that the generated results have fully and stably appeared in the scene; the return signal often precedes the actual results. During testing, this can lead to some unstable behavior, such as:

1. Instance is missing
2. A Houdini logo placeholder is temporarily displayed in the scene
3. invalid task log
4. The parameters appear to have been written, but the final cook result does not correctly reflect them; the log reports an error stating that the parameters have “no match”.

Therefore, this project introduces a controlled cook layer between scene update decisions and Houdini Engine execution.



Item 10 HDA execution in Unreal cannot be treated as an instant operation.

In early implementations, the tool attempted to process HDA instantiation and recooking in batches. For example, in the RTX 4060 laptop test environment used for this project, cook or instantiate requests were sent consecutively in groups of about 10 HDAs. Test results showed that, in most cases, this approach resulted in multiple HDAs failing to instantiate correctly. Common symptoms included the Houdini logo placeholder remaining in the scene, missing instances, and Houdini Engine returning errors such as “invalid id.” This issue was more likely to occur when the machine was under heavy load or operating at high temperatures, indicating that this batch strategy is sensitive to the state of the runtime environment.

Based on test observations, the problem is not merely a single HDA cook failure, but

rather that multiple Houdini Engine tasks are stacked on top of one another before the previous task has actually completed stably. Earlier versions attempted to mitigate this issue by increasing the task interval (tick), forcing a recook after each batch, and collecting failed instances at the end for remediation. While this strategy could fix some instances that failed in the latter half of the batch, it could not reliably track and recover all objects that failed early on. Additionally, since both the end of the batch and the final recovery phase could trigger additional recooks, some instances would undergo multiple repeated recooks, increasing execution costs and making task status more difficult to determine. Consequently, the project abandoned the “batch cook followed by recovery” strategy.

The final implementation shifted to a more conservative, controlled execution model. HDA spawn and recook tasks are processed sequentially; the next task is not released until the current task produces verifiable completion evidence. After extensive experimentation, the tool intentionally reserves `serial=1` for HDA operations. This is not a UI limitation but rather an execution strategy designed to reduce memory pressure and improve spawn quality—it currently yields the most stable results.

Each cook task is tracked using a combination of evidence, which is ultimately used collectively to determine the “settle” state. First, the tool attempts to bind to the Houdini Engine’s post-processing or post-cook delegate. When the callback is triggered, the system records that a post-cook signal has been received. Second, the tool polls the wrapper’s state to check whether the HDA is still in a state such as cooking, instantiating, loading, or processing. Third, the system checks whether the actor already possesses actual generated geometry (not the Houdini logo), rather than assuming the cook is complete simply because the actor exists. The task is considered settled only when the system receives a sufficient number of post-cook signals, or when the wrapper is no longer busy and the actor has possessed actual cooked geometry for several consecutive ticks. This logic uses variables such as posts_needed, idle_streak, had_real_at_start, and saw_busy. In actual testing, different types of HDAs do not have exactly the same requirements for post-cook signals. For some simple HDAs, a single post-cook signal is sufficient to indicate that their output is ready for use. However, for parameterized point HDAs and curve HDAs, multiple tests have shown that a single post-cook signal does not always result in a fully writable state. The author previously attempted to process point HDAs by waiting for only one post-cook signal; as a result, some instances were generated

but did not correctly apply the parameters passed in by the DataTable. Further investigation revealed that while child HDAs actually had corresponding parameter mappings, the tool had not yet reliably recognized these parameters after the first cook. Therefore, the tool uses “posts_needed=2” in such cases: the first cook is used to generate or expose the HDA’s runtime state, while the post-cook ensure phase rewrites parameters or binds inputs, triggering a second controlled recook to ensure the final output reflects the target parameters.

However, this strategy requires additional safeguards during incremental sync. A full rebuild typically starts from an empty scene, so the presence of a “real mesh” can serve as strong evidence of completion; but incremental sync deals with existing Actors, which often already possess old cooked meshes before this recook begins. In such cases, the presence of a mesh does not prove that the new parameter updates have been completed—it may simply be the result of the previous cook. Therefore, incremental recook cannot rely solely on a geometry check but must combine “had_real_at_start” with “saw_busy”. If an Actor already has real geometry before the recook, the tool must observe that the wrapper has entered the “busy” state at least once during the current task before allowing the task to be deemed settled based on the combination of the idle streak and geometry evidence. This prevents the old mesh from being mistakenly identified as the final result of this recook.

```
if job.get("had_real_at_start") and not job.get("saw_busy"):
        return False
    if has_real and job["idle_streak"] >= cook_idle_ticks(job.get("cfg")):
        job["settled"] = True
        return True
```

If a post-cook signal that meets the specified criteria is not received before the timeout, the tool does not immediately consider the entire operation a success; instead, it logs the instance for a subsequent verification or health

pass. The final check can identify Actors that still have empty outputs or placeholder geometry and perform selective recooking on them. Therefore, the system incorporates both prevention and recovery mechanisms: serial execution reduces task overlap, settle logic prevents premature release, the ensure pass rewrites any potentially missing state, and the verification pass handles any remaining incomplete outputs.

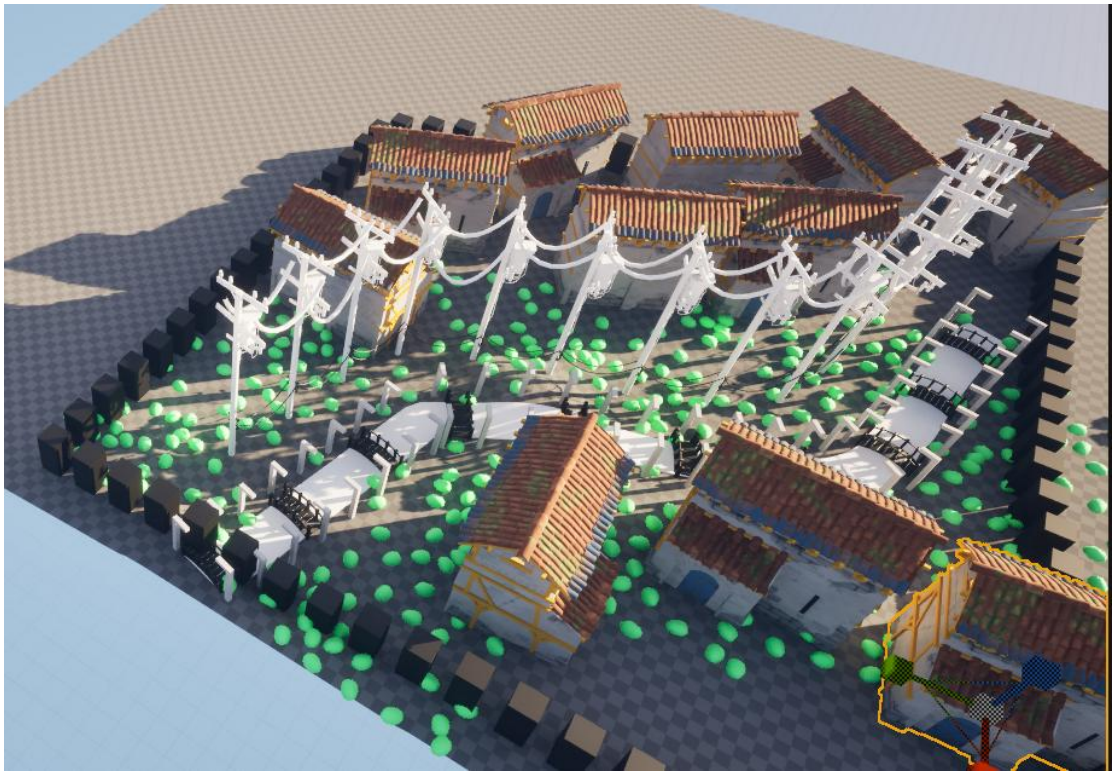
Static Meshes do not enter this complex workflow; instead, the tool handles them directly through StaticMeshComponent operations. This computationally expensive, constant-speed logic is reserved exclusively for HDAs.

4 Validation and Discussion

4.1 Validation

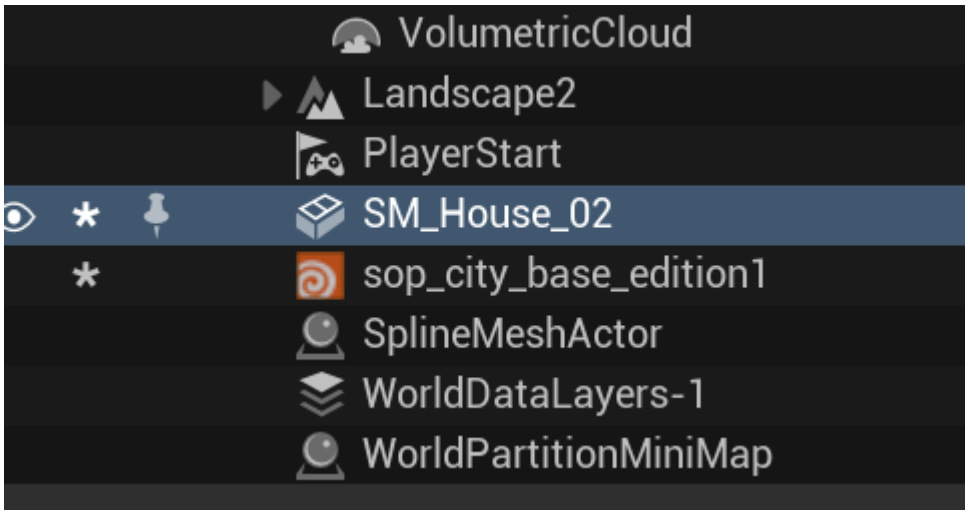
4.1.1 Layout Level

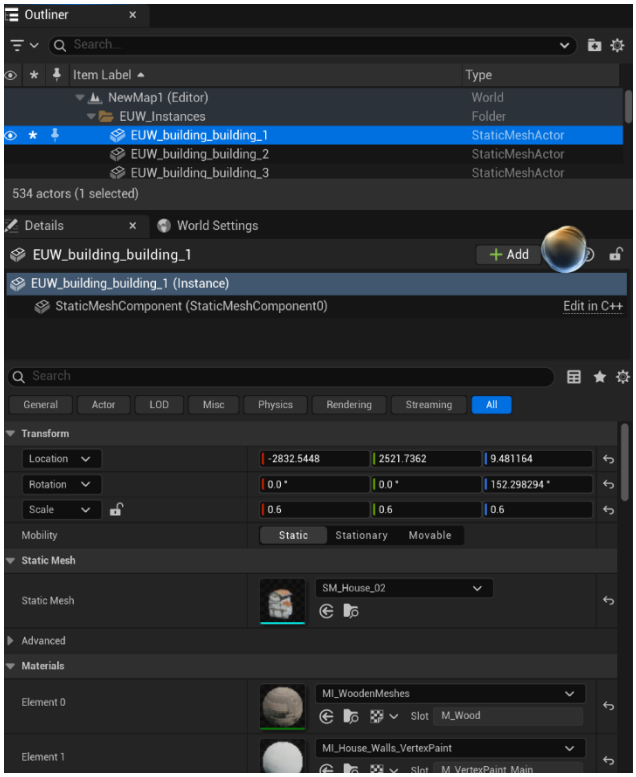
This section verifies whether the tool can complete the basic deployment process from a Layout HDA to an Unreal scene instance. The focus of the test is not on the final visual quality, but rather on whether the data flow is complete: whether the tool can bind to the Layout HDA, read its DataTable output, identify the layout type, bind the type to a Content asset, and ultimately generate HDA and StaticMesh instances.



Current testing shows that this tool can link any HDA or static mesh to a grid-based layout HDA. Even if a child HDA was not created according to the tool's parameter naming conventions—resulting in its editable parameters not being recognized—the core "row-to-asset" deployment process will not be blocked. Unmatched parameters will be skipped and logged, while positioning, rotation, and asset generation will continue as normal. The white streetlight shown in the figure is a simple HDA with no direct nesting relationship to the Layout HDA; since it does not use standardized parameter naming, the tool cannot read its

slider parameters, but the streetlight instances are still correctly placed on both sides of the road and maintain the correct orientation.

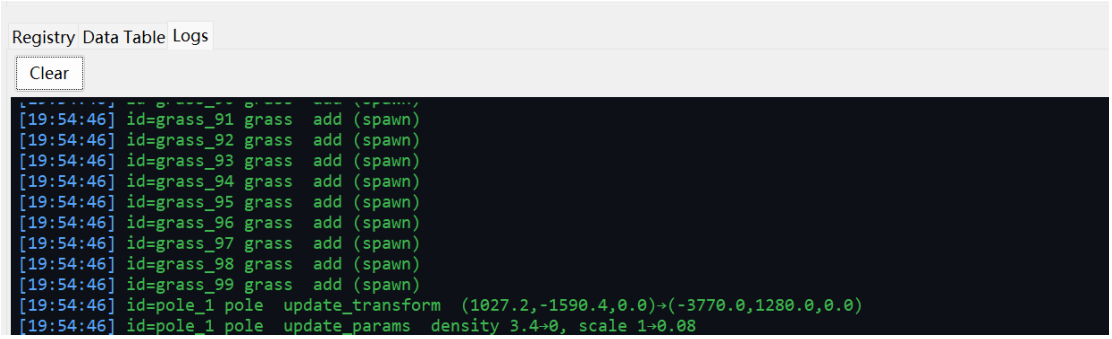




Item 11 ue outline

The naming conventions in the Level Outliner also confirm that these instances were not manually dragged into the scene. If a house StaticMesh is dragged directly from the Content Browser into Unreal, its Actor name will retain the original asset name, such as SM_House_02. Instances generated by this tool, however, are placed in the tool-managed EUW_Instances folder and use names based on internal IDs, such as EUW_building_building_1. This indicates that the house instance was generated through the tool's deployment process and remains associated with its normalized

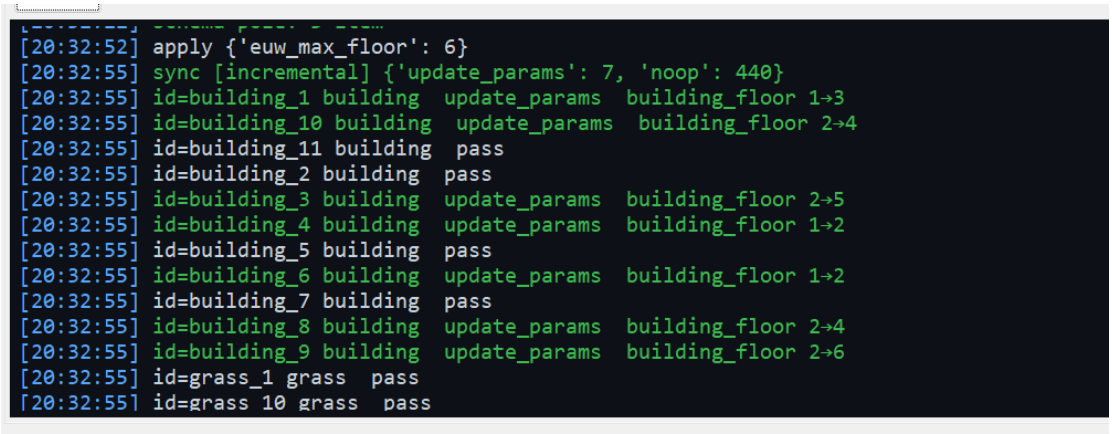
internal ID. Next, the synchronization behavior after updating the Layout HDA was tested. After the author modified the Layout HDA in Houdini and refreshed it in Unreal, asset bindings for certain types required reconfiguration, while pole types with existing valid bindings were still recognized by the system. After re-configuring the missing bindings and performing a sync, the log showed that the pole instances were not fully rebuilt, but rather reused, moved to new positions, and had new parameters written to them. This is consistent with the snapshot -> diff -> sync logic described earlier, indicating that the tool can continue to recognize and update existing instances after layout changes.



Item 12 pole performance

Then further tested the performance of point HDA within this workflow. In this test, we replaced the house that originally used a StaticMesh with an HDA house to verify whether the HDA child asset could receive parameters from a DataTable row and generate different results. As shown in Item 13, the house instances have different heights, indicating that floor or height-related parameters were successfully passed from the layout data to the HDA house and reflected in the final geometry after recooking. This test confirms that the tool can not only place HDA assets in the correct positions but also write row parameters to the child HDA when parameter matching is successful.


Item 13 hda house



Now, adjust the euw_max_floor parameter in the layout hda to increase the maximum number of floors. The logs show that the building HDAs have all been reused. Buildings with unchanged parameters were approved directly, while others had their parameters modified.

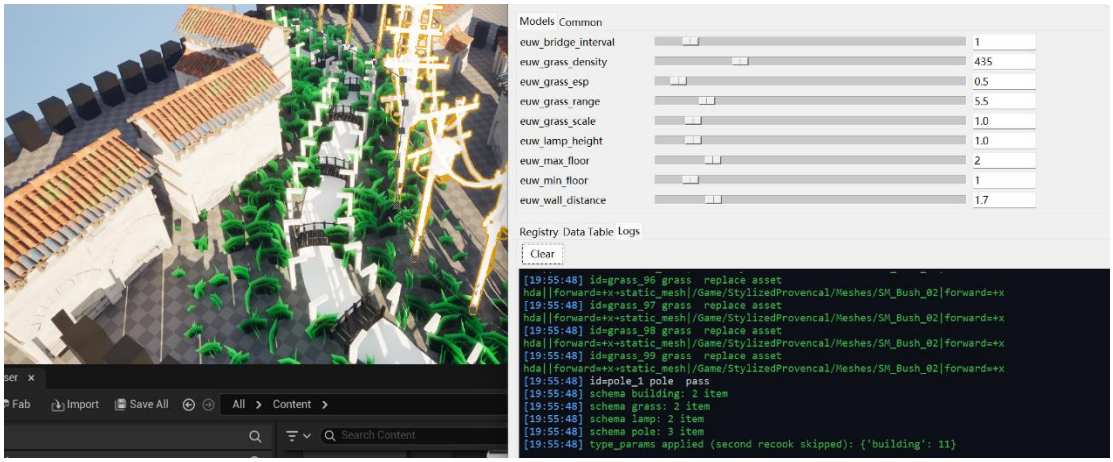
4.1.2 Type Level

This section verifies whether the tool can apply changes at the type level without rebuilding the entire scene. This test focuses on two common operations: changing the sharing settings for all active instances of a given type, and replacing a type's resource bindings, while leaving unrelated types unchanged. In both cases, the expected behavior is that the system locates instances using stored registry entries, tags, and snapshot records, applies the new configuration only to the affected type, and leaves unrelated instances of buildings, utility poles, or vegetation with a "Pass" or "No Action" result.

In the first test, the scaling settings for the Static Mesh house types were changed via the "Settings" window. The log shows 11 house instances were affected. This result is consistent with the implementation: Static Mesh instances do not enter the Houdini cook queue, and their shared type settings are applied directly to the corresponding Static Mesh Actor through scale and material updates.

Item 14 change fbx house scale

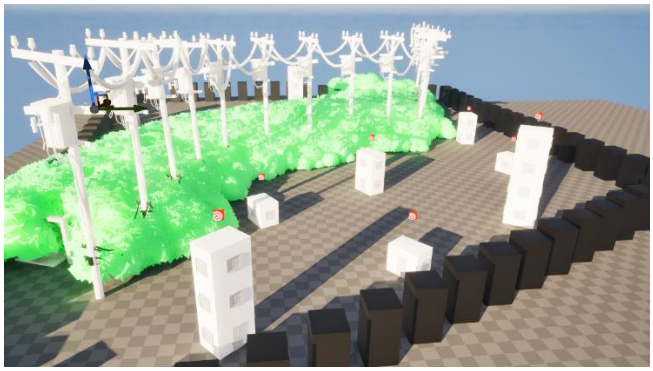
The second test involved replacing the grass resource's bind with another mesh resource. The logs showed that this operation followed the same type-isolated update path discussed earlier: the grass instances were replaced, while the building and utility pole instances remained unaffected. After the replacement, the scaling changes generated by the layout data were still visible; the grass near the road appeared larger than the grass farther away. This confirms that changing the bound content resource does not break the type's row parameter link.

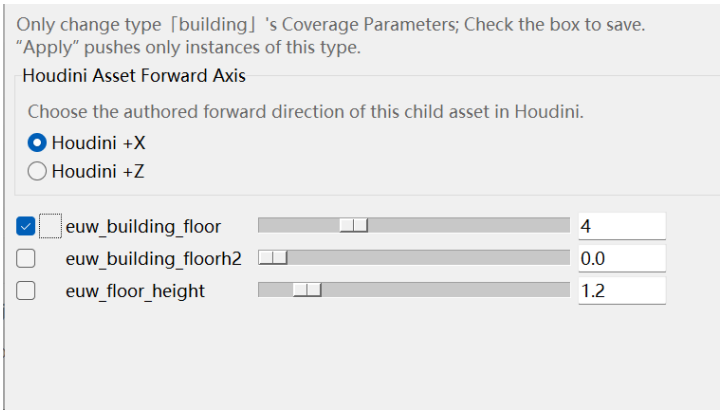


Item 15 change grass

The next test changes the building type from a StaticMesh house to an HDA house, then applies a type-level HDA parameter. In the “Settings” window, select the parameter euw_building_floor and set it to 4. Item19 shows that the building resources have been regenerated based on the new floor height. The log shows that 9 instances require parameter updates and serial recalculation, while the “Outline” view displays a total of 11 building Actors. This difference is crucial: the tool does not simply recalculate a building Actor just because it belongs to the selected type. Instead, `apply_params_to_houdini_wrapper` skips instances whose valid parameter values have remained unchanged, sending only those instances where parameters have actually changed to the serial recalculation queue.

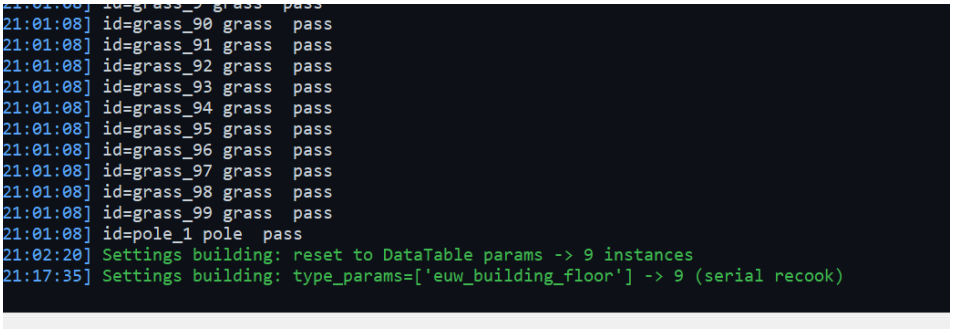
Subsequently, the “Reset” test verifies the reverse path. “Reset” clears the `type_params` overrides stored in the registry and reconstructs the valid parameters for each instance based on the recorded `param_schema` defaults and the original row parameters in the applied snapshot. The “Setup” window displays “Reset; serial recalculation for 9 instances,” and the scene is immediately restored to its original layout-driven state. This demonstrates that type-level control is not a destructive edit to the original layout data; rather, it is a temporary overlay that can be removed and reapplied.





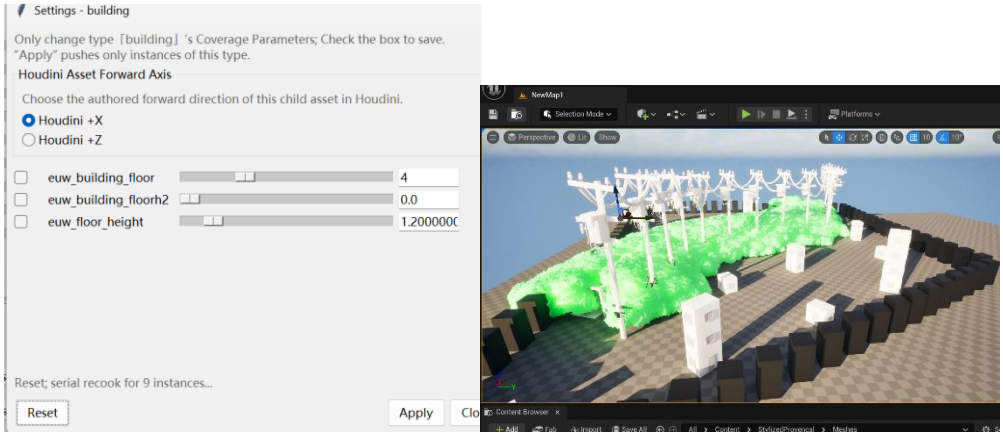
Item 16 change floor to 4

Item 17 outline- 11 buildings



Item 18 logs

Item 19 outcome



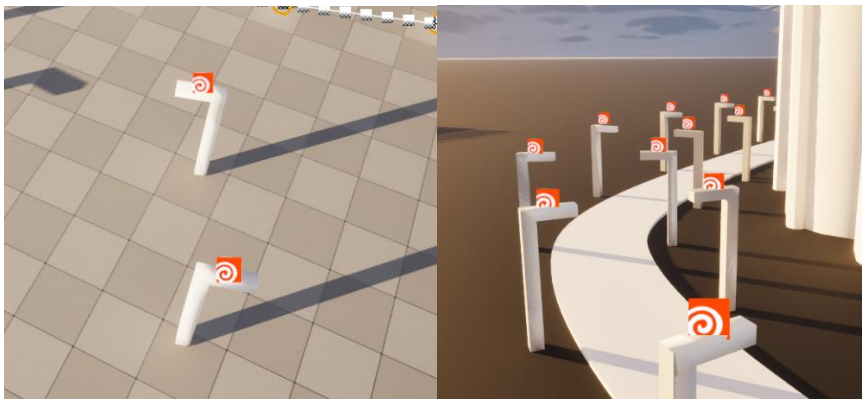
Item 20 reset

4.1.3 Setting Window Permanent Parameters

This section verifies the persistent type settings stored in the registry, which are reused when the same type is deployed again. Unlike temporary inspection parameters such as `euw_building_floor`, these settings describe how the tool should interpret the asset. Key examples include the Houdini forward axis for HDA assets, as well as the origin mode and color/scale controls for StaticMesh or imported FBX mesh assets.

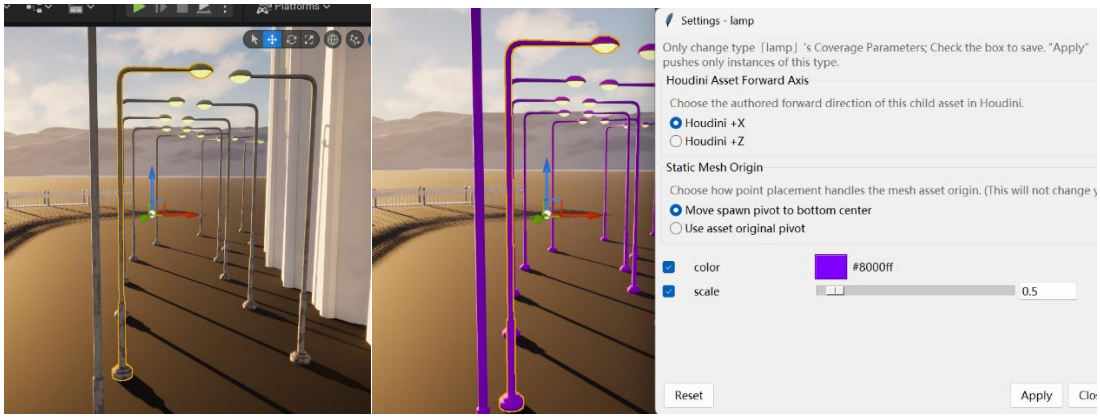
The first example uses two streetlight assets. The white streetlight is an HDA created in Houdini with a forward direction of +Z, rather than the tool's default of +X. As shown in Project 20, when the default direction is used, the generated streetlights do not align with the direction of the road. After changing the type setting to +Z, the tool is able to consistently transform the layout yaw angle so that the streetlights face the

center of the road. This confirms that the direction is not hard-coded for each asset; rather, it is stored as a type-level interpretation rule.



Item 21 hda lamp

The second example uses an FBX light imported as a StaticMesh resource. In this case, the pivot point of the original mesh is located near the center of the object; if the row position is applied directly, the light will intersect with the ground. The tool's default StaticMesh origin handling mechanism compensates for this issue by offsetting the character upward by approximately half the mesh height, so that the light stands firmly on the ground after placement. The same StaticMesh path also supports color changes: this is achieved by creating a dynamic material instance in the first material slot and applying a tint color from the settings or row parameters.



Item 22 fbx lamp

4.2 Discussion, Limitations and Future Work

In terms of functional completeness, this tool has already implemented the main closed-loop workflow proposed for this project: the tool can read layout data from Layout HDA, bind assets in Content to different layout types, add configurations to

assets and types, and update previously generated scenes when layout or type settings change. After the initial full deployment is complete, subsequent updates are typically faster because instances that have not changed are skipped outright, reusable instances are moved to new positions via transforms, and only HDA instances that are actually affected enter the recook queue.

However, if we compare only the execution speed of a single cook queue, this project's current controlled cook workflow is not faster than cooking directly within Houdini. This is an intentional trade-off. Houdini Engine operations in Unreal are asynchronous, and in practical testing, a single callback or status signal is not always sufficient to confirm that an HDA instance has been fully processed. Therefore, this tool employs multiple completion checks, including delegate signals, wrapper busy status, output checks, actual geometry detection, and an `ensure` pass when necessary. This improves the reliability of the generated results but also makes the update process more conservative and increases wait times. Consequently, the current implementation prioritizes correctness and scene stability over achieving the highest possible cook throughput.

This project has not yet undergone systematic benchmarking on particularly large-scale, production-grade scenes. Current validation is primarily based on the author's test scenes, with a focus on verifying the stability of data binding, instance reuse, type-level parameter updates, reset rollbacks, and serial HDA recooks. The authors also conducted stress tests on complex child HDAs: for HDAs that already reference complex FBX geometry and have at least approximately 50,000 vertices, this tool can reliably configure 16 instances. However, this does not constitute a comprehensive performance test at scale. Future work will require systematically recording metrics such as cook time, failure rates, and memory usage under conditions involving larger scenes, more asset types, and more complex layouts.

Another limitation is material control. Currently, the tool reserves channels only for color configuration and can simultaneously support color parameters within the HDA as well as color modifications to StaticMesh and imported FBX mesh assets. For StaticMesh assets, the tool creates a dynamic material instance using the first material slot and overlays a tint color on top of it. While this is sufficient for basic color testing, it cannot yet be considered a complete material management system. Multi-material assets, complex shader parameters, texture replacement, and independent control of each material slot have not yet been included in the current

implementation.

The final unfinished feature is feeding child asset dimensions back to the Layout HDA. The project's initial vision included the ability to adjust the base layout based on the actual dimensions of bound parts, but this functionality was ultimately not completed. Currently, if a bound asset is too large or too small relative to the layout, users can only manually adjust parameters such as scale via the type-level Settings Window; the tool cannot automatically measure the dimensions of the generated child asset and feed that information back to the Layout HDA. This functionality requires more reliable methods for determining when generated geometry is truly complete, for uniformly measuring the bounding boxes of different asset types, and for feeding this dimensional information back to the layout generation phase.

Therefore, future projects can continue to develop in three directions. First, further optimize the cook-control logic to reduce unnecessary waiting while maintaining the stability of the current serial and ensure mechanisms. Second, expand the material system from simple color tints to support multiple materials, multiple shader parameters, and texture-level control. Third, extend the Unreal-side orchestration layer by adding a controlled feedback mechanism, allowing the dimensions of generated child assets to influence subsequent layout generation.

4.3 Declaration of Generative AI Use

Generative AI tools were used throughout this project as development support. Their main contributions included assisting with the implementation of the Unreal Engine panel code, identifying possible causes of errors from code and runtime logs, suggesting debugging approaches, and helping explore Houdini Engine API behaviours. AI assistance was also used during the development of the serial cook system, particularly when investigating available callback signals, runtime states, and possible parameter names.

The author defined the project aims, workflow requirements, system architecture, interface design, data structures, and expected behaviour of the tool. AI-generated suggestions and code were reviewed, modified, integrated, and repeatedly tested by the author within Unreal Engine and Houdini. Technical decisions, including the separation of layout and child assets, snapshot-based incremental synchronisation, type-level control, and the final serial cook and validation strategy, were made based on the author's testing and evaluation. Generative AI was therefore used as an

implementation and debugging assistant rather than as an autonomous designer or evaluator of the project.

5. Conclusion

This project developed a procedural environment creation workflow for artists in Unreal Engine by using Python to manage and edit Houdini Digital Assets (HDAs). The tool provides artists with a centralized platform for binding, configuring, deploying, and updating the HDAs and StaticMesh assets required for procedural layouts, eliminating the need to manually manage individual objects through separate “Details” panels.

By decoupling layout HDAs from their child resources, the system replaces nested resource dependencies with data-driven connections based on layout data, persistent registry entries, and internal snapshots. Its incremental synchronization logic distinguishes between instances that are unchanged, reusable, transformed, newly added, or have had their parameters modified. As a result, existing characters can be retained or updated in place, while only affected HDA instances are submitted to a controlled, sequential recook queue. This establishes a complete workflow from programmatic layout generation to Unreal Engine-side asset deployment, parameter control, incremental updates, and resets.

Although further work is needed in areas such as baking performance optimization, material control, large-scale benchmarking, and sub-resource size feedback, this project demonstrates a viable Unreal-side coordinated workflow for managing heterogeneous procedural resources. It not only provides artists with an easy-to-use control platform but also lays the technical foundation for more stable and selective HDA updates within Unreal Engine.

6. Reference

alexsied (2024) ‘Proper workflow with multiple assets’, *SideFX Forums*, 3 October. Available at: SideFX Forums – Proper workflow with multiple assets

Spahr, C. (2023) ‘Re: Houdini Engine and multi pass asset generation’, *SideFX Forums*, 2 November. Available at: SideFX Forums – Houdini Engine and multi pass asset generation.

SideFX (2026) *PDG Asset Link settings*. Houdini 22.0 Documentation. Available at: https://www.sidefx.com/docs/houdini/unreal/pdgsettings.html

SideFX (n.d.) *The Dialog Script Language*. HDK Documentation. Available at: https://www.sidefx.com/docs/hdk/_h_d_k__h_d_a_intro__dialog_script.html