CookieJar:
An Unreal Engine 5 Physics-Assisted Container Dressing Plugin

Design Document

Author: Julie Swei

1. Project Overview

Project Name

CookieJar: An Unreal Engine 5 Physics-Assisted Container Dressing Plugin

Elevator Pitch

An environment artist myself, there's nothing I love more than a well-dressed and rich environment. Achieving this look, however, often means dressing hundreds of assets and dozens of variations across an entire scene, which may take hours or days of work. CookieJar is an Unreal Engine 5 editor plugin that helps environment artists quickly fill containers with believable arrangements of props using physics-assisted procedural placement. Instead of painstakingly placing dozens of individual objects, artists can generate, iterate, and bake natural-looking results in seconds -- all directly in-level, making it much easier to evaluate the dressing in its final scene context.

TL;DR: I want users to be able to quickly and realistically dress containers with various props.

Goals

- Accelerate the process of dressing containers with props.
- Produce believable, organic-looking arrangements using physics-assisted placement.
- Enable rapid iteration through adjustable parameters and one-click regeneration.
- Allow artists to preview results directly in-level and within context.
- Bake the generated arrangement into editable static props.
- Optionally allow props to overflow naturally onto a user-selected ground surface.

Non-Goals

- Implement a custom or advanced physics engine.
- Simulate complex physical interactions beyond initial placement and settling.
- Support gameplay interactions with generated props after baking.
- Replace general-purpose procedural generation or scattering tools.
- Automatically create or modify meshes; CookieJar only places **existing** assets.

2. Problem Statement

Problem

- Artists spend a significant amount of time filling, arranging, and tweaking props inside containers. While this work is important to the final quality of a scene, it is repetitive and time-consuming.
- In large environments such as markets, shops, kitchens, or warehouses, there may be dozens or even hundreds of containers that need to be dressed. Creating unique arrangements for each one quickly becomes a large production task.
- Manual dressing can also introduce unintended patterns. Props may end up with similar orientations, repeated arrangements, clipping, or unrealistic settling because artists naturally fall into consistent placement habits over time.

Why It Matters

- Creating unique container arrangements by hand is possible, but it takes time. A common shortcut is to duplicate and rotate an existing arrangement, but repeated patterns can become noticeable, especially when many containers appear in the same scene.
- Even if players or viewers don't consciously recognize these repetitions, they can make an environment feel less natural and less visually rich.
- Time spent manually dressing containers is also time that cannot be spent on composition, lighting, storytelling, or other creative work. Because each variation requires additional manual effort, artists may explore fewer ideas before settling on a final result.
- Environment dressing is both a technical and artistic task. Artists often want to experiment with different densities, compositions, and levels of clutter before finding the look that best supports the scene. Faster iteration makes that creative process easier.
- Finally, manual dressing can lead to inconsistencies between artists, making it harder to maintain a consistent visual style across a project.

Existing Solutions

- Manually dress one container, then duplicate and rotate it throughout the scene.
- Create several hand-made variations and reuse those throughout the environment.
- Use generic scatter tools that distribute objects according to procedural rules.

- Use Unreal Engine's PCG Framework, which is designed for a wide range of procedural placement tasks. While powerful, PCG is a general-purpose system, whereas container dressing focuses on generating believable arrangements inside bounded spaces and quickly iterating on those results.
- Use external procedural tools such as Houdini. These tools are highly capable, but typically involve generating content outside Unreal Engine before exporting and importing the results. CookieJar instead focuses on fast, in-editor iteration, allowing artists to generate, preview, adjust, and bake arrangements without leaving the level.

Why CookieJar?

CookieJar is an artist-first Unreal Engine 5 editor plugin focused on a common production task: dressing containers with believable arrangements of props. Unlike more general procedural workflows, CookieJar provides a lightweight, in-editor experience that allows artists to generate, iterate, and bake results without leaving Unreal Engine or managing an export/import pipeline.

By combining procedural generation with physics-assisted placement, CookieJar produces natural-looking arrangements while preserving artist control through intuitive parameters such as prop/container selection, object count, density, randomness, in-container behavior, and overflow behavior. Artists receive immediate visual feedback directly in the level, making it easy to evaluate results in the context of the surrounding environment and rapidly iterate until the desired look is achieved.

As a bonus, CookieJar also supports controlled overflow, allowing props to naturally spill from containers onto a user-selected ground surface when appropriate, creating more organic and visually convincing arrangements.

Rather than attempting to replace general-purpose procedural tools, CookieJar is intentionally designed around a single, frequently repeated environment-art workflow, providing a fast, focused, and artist-friendly solution for everyday scene dressing.

3. Target Users

Primary Users
Environment artists, set dressers

Secondary Users
Level designers, 3D layout artists, lighting artists, art directors

User Pain Points
- Dressing containers with many small props is repetitive and time-consuming.
- Manually placing dozens of objects while maintaining a natural appearance is tedious.
- Small adjustments often require repositioning many objects individually.
- It is difficult to quickly experiment with different densities and arrangements.
- Artists want believable results while retaining creative control.
- Existing workflows can interrupt iteration by requiring repeated manual edits or external tools.

4. Scope

Version 1 Features
- Dockable Unreal Engine editor window for generating container arrangements.
- Support for selecting a target container and one or more prop meshes.
- Artist-adjustable generation parameters, including density, randomness, object count, simulation settings, and ground collision (if props overflow).
- Physics-assisted placement that generates believable arrangements inside containers and supports natural overflow onto surrounding surfaces.
- One-click regeneration to quickly explore different arrangements using the same or updated parameters.
- Bake generated arrangements into the level as editable static meshes.
- Clear previously generated arrangements.

Future Features
- Multi-container selection
- Support for shelves, tables, and other non-container surfaces.
- Intelligent placement rules based on prop type or size.
- Custom placement constraints.

5. User Workflow

User Story (what the "need" sounds like)

"As an environment artist, I want to quickly populate a fruit basket so that I don't have to place every individual apple by hand."

"I'm building a market scene with many different baskets of produce, and I want to dress the produce efficiently so I can save time for other work."

Step-by-Step Workflow

- 1. Prepare the container actor and prop assets.
- 2. Open the CookieJar editor window.
- 3. Select the container actor from the level.
- 4. Select one or more prop meshes from the Content Browser.
- 5. Configure generation parameters.
- 6. Click Generate.
- 7. Allow the physics simulation to settle.
- 8. Review the generated arrangement in context.
- 9. If necessary, click Reset, adjust parameters, and regenerate.
- 10. Click Bake to commit the arrangement to the level (actors will be grouped or converted to ISM).
- 11. Repeat for additional containers.

6. Functional Requirements

Container Selection

- User can select one container actor from the level.
- Plugin validates that the selected actor has a static mesh.
- Plugin reads the container's bounds and transform.

Asset Selection

- User can select multiple prop assets from the Content Browser.
- Plugin validates that the selected assets contain a valid static mesh.
- Plugin reads the bounds and transforms of all selected props.

Placement Generation

CookieJar should generate an initial set of prop placements based on the selected container, selected prop assets, and user-defined parameters.

The system should:

- Spawn props within or above the container's bounds so that when they fall, they all fall into the container.
- Respect object count, density, randomness, and scale variation settings.
- Randomly select between multiple prop assets if more than one is provided.
- Use a random seed so results can be reproduced.
- Avoid starting props in extreme or unusable positions when possible.
- Support natural overflow when the number or density of props exceeds the container space.

Physics Simulation

CookieJar should use Unreal Engine's built-in physics simulation to settle generated props into believable positions.

The system should:
- Enable collision on generated props.
- Allow props to collide with the container.
- Allow props to collide with each other.
- Optionally allow props to collide with a selected ground surface.
- Let props fall, roll, and settle naturally.
- Stop the simulation after a set duration or once objects have mostly settled.
- Preserve the final transforms after simulation ends.

Baking

CookieJar should allow the user to commit the generated result to the level once they are happy with the arrangement.

The system should:
- Turn off physics simulation on generated props.
- Preserve each prop's final position, rotation, and scale.
- Convert temporary simulation props into editable placed objects.
- Group or organize baked props under a clear parent/folder name.
- Keep the baked result editable by the artist.
- Option: convert to ISM for performance

Clearing / Regeneration

CookieJar should allow artists to quickly remove generated results and try again.

The system should:
- Clear generated props from the current CookieJar session.
- Preserve the selected container, prop assets, and parameter values.
- Allow the user to regenerate a new result with the same settings.
- Allow the user to adjust parameters and generate a different result.
- Avoid deleting manually placed objects that were not created by CookieJar.

Error Handling

CookieJar should guide the user when required inputs are missing or invalid.

The system should:
- Warn the user if no container is selected.
- Warn the user if no prop assets are selected.
- Warn the user if the selected container has no usable collision.
- Warn the user if selected props have no usable collision.
- Prevent generation if object count or density values are invalid.
- Warn the user if overflow is enabled but no ground collision target is selected.
- Fail safely without crashing the editor.

7. User Interface

Plugin Window

Needs:

- Container slot
- Props slots
- Parameter section below with checkboxes and sliders
- Ground collision check
- Simulation settings
- Generate, Reset, Bake, Clear, etc. buttons
- Other parameters/checks for ease of use

8. User Parameters

Required Parameters

- Container Actor
    - Defines the container that will be filled. The plugin uses its collision and bounds to determine where props can settle.
- Prop Asset(s) + Counts
    - Specifies one or more meshes to populate the container with.
- Randomness (if objects are dropped at similar angles vs. wildly different angles)
- Spawn Spread
- Scale Variation
    - Controls pre-simulation parameters
- Overflow checkmark
    - If checked, objects will overflow onto collidable objects. If unchecked, overflowed objects will simply be deleted

Optional Parameters

- Initial Drop Height - controls the height that the props are dropped at
- Bounciness + Gravity Scale

Advanced Parameters (Future)

- Asset weighting (60% apples, 40% pears)
- Maximum stack height (props are only allowed to pile up to a certain height)

9. System Architecture

High-Level Overview

CookieJar is organized into a series of small systems, each responsible for one part. The plugin starts by gathering the user's selected container, prop assets, and generation settings. From there, it generates an initial arrangement, lets Unreal's physics simulation settle the props into place, and finally bakes the finished result into the level.

UI System

Responsibilities:

- Display the CookieJar editor window.
- Provide controls for generation settings.
- Allow users to select containers and prop assets.
- Trigger Generate, Bake, and Reset actions.
- Display warnings, errors, and status messages.

Selection System

Responsibilities:

- Read the selected container actor.
- Read selected prop assets.
- Read selected ground actors (if overflow is enabled).
- Validate user selections.
- Retrieve mesh bounds, transforms, and collision information.

Asset Management

Responsibilities:

- Store and manage the selected prop assets.
- Validate asset compatibility.
- Randomly choose prop assets during generation.
- Maintain asset lists and future weighting information.

Placement Generator

Responsibilities:

- Calculate valid spawn locations.
- Determine the initial spawn volume.
- Apply generation parameters such as density, randomness, and scale variation.
- Generate the initial prop arrangement before simulation.

Physics Manager

Responsibilities:

- Enable Unreal's physics simulation.
- Configure collision between props, containers, and optional ground objects.
- Allow props to fall and settle naturally.
- Monitor simulation progress.
- Determine when simulation has completed.

Bake System

Responsibilities:

- Stop physics simulation.
- Preserve the final transforms of generated props.
- Convert temporary simulation actors into editable level actors.
- Organize baked actors within the level hierarchy.

Cleanup System

Responsibilities:

- Remove generated props.
- Preserve manually placed actors.
- Clear temporary simulation data.
- Support regeneration without resetting user parameters.

Validation System

Responsibilities:

- Verify required inputs are present.
- Ensure containers and props have valid collision.
- Validate user parameter ranges.
- Prevent generation when invalid settings are detected.

10. Generation Pipeline

Placement Algorithm

1. Read the selected container and its bounds.
2. Generate an initial spawn volume above or within the container.
3. Randomly choose prop assets from the selected asset list.
4. Generate spawn positions based on object count, density, and randomness settings.
5. Apply initial rotation and scale variation.
6. Spawn temporary physics-enabled actors.

Spawn Algorithm

For each object:

1. Randomly select a prop asset.
2. Generate a valid spawn position.
3. Apply random rotation.
4. Apply scale variation.
5. Spawn the temporary actor with physics enabled.

Physics Workflow

1. Enable collision and physics simulation.
2. Allow props to collide with the container, each other, and optional ground objects.
3. Run the simulation until either:
- the maximum simulation time is reached, or
- the props have come to rest.
4. Store each prop's final transform.

Baking Workflow

1. Disable physics simulation.
2. Preserve each prop's final transform.
3. Convert temporary simulation actors into editable level actors.
4. Organize baked actors into a parent actor or World Outliner folder.
5. Clean up temporary generation data.

11. Unreal Engine Integration

UE Systems Used
- Unreal Editor
- Chaos Physics
- Static Mesh System
- Collision System
- Actor System
- Asset Registry
- Content Browser
- Slate UI
- Editor Utility Framework

Classes / APIs
- AActor
- AStaticMeshActor
- UStaticMesh
- UStaticMeshComponent
- FBox
- FTransform
- GEditor
- FAssetData
- SButton
- SSlider

12. Development Milestones

Milestone 1
Working plugin:
- Plugin loads successfully.
- CookieJar window opens.

- UI is functional.
- Generate/Reset/Bake buttons exist.
- User can select a container, prop assets, and ground plane assets.

Milestone 2

Initial prop arrangement can be generated:

- Spawn positions are generated.
- User parameters affect generation.
- Props spawn correctly.
- Multiple asset types work.
- Overflow support exists.

Milestone 3

Physics produces believable arrangements:

- Props fall using Chaos Physics.
- Props collide correctly.
- Overflow settles naturally.
- Simulation stops correctly.
- Results are stable.

Milestone 3.5

Overflow behavior:

- Overflow checkmark disabled: objects that fall out of the container don't collide with whatever is beneath it, and are automatically deleted once they leave the container space. (Container exit detection)
- Overflow checkmark enabled: objects that fall out of the container collide with anything that has collision underneath it -- normal physics behavior (Container exit detection)
    - However, objects will not fall past a fall-distance cap, dictated by 4x (tentative) the container's height. They will disappear once they pass this limit.
- "Overflow Horizontal Roll Damping" parameter, only affects objects that have fallen out of the container. 0 = rolls freely/naturally; higher values = stops rolling sooner. It is gated to only be enabled when Overflow is checked. Note: It's hardcoded for now, but can become a parameter later if manual tweaking by the user is needed.

Milestone 4

Polish to be a usable artist tool:

- Bake works. Bake to editable groups (ISM later).
- Reset works. Reset sets the mesh transforms back to original pre-sim positions (need to cache those positions in Generate, before RUN kicks off physics)
- A maximum limit should exist for the number of assets that can be generated.
- Error handling exists.
- UI is polished.
- Generation is reliable.
- Demo scene completed.

Milestone 5

Future Features:

- Batch multi-container generation, select several baskets/containers at once and dress them all with one click of Generate. The containers should have different seeds as well, so it's not the same-looking pool of props in different containers.
- Include Instanced Static Meshes as an option for the baked result/output. When count is high, bake into an ISM/HISM component instead of individual actors, to reduce actor overhead and improve performance at high object counts.