Modular Brutalist Office

Product Link: Brutalist Architecture Office
Trailer: Brutalist Architecture Office
Demonstration Video: Brutalist Architecture Office - Breakdown
Screenshots: ArtStation
Breakdowns: ArtStation
Demo Build: Download demo

Release Notes

1.0

- Initial version

1.1

- Window decoration system refactored to be more robust
- Window decorations now supports multiple mesh sockets
- Building system supports options to add doors for each floor
- Old door spawning logic is now removed and replaced with the new “FloorData” array

1.2 (Unreal Engine 5+)

- Meshes that can work with Nanite system are converted to Nanite meshes
- Example map lighting tweaked to work better with Lumen lighting system
- Blueprints are converted and tested to work well in UE5

1.3 (Unreal Engine 5.3+)

- Building system refactored/optimized
- Building system bug fixes
- Spline system refactored/optimized
- Spline system bug fixes
- Turned various meshes into Nanite meshes
- New example player and overall overhaul
- All of the inputs are now using Enhanced Inputs

Important information

Make sure you are not manually copying files with file explorer but instead use the migrate workflow. More information here. This way you will avoid possible issues that might occur if you move files outside of Unreal Engine.

If the editor asks to import files, choose "Don´t Import". This is because the pack can contain source files for those who like to edit them and importing them will reset some settings and cause issues.

This pack will not work with static lighting because blueprint assets are creating instanced mesh components that are not supporting baked lightmaps. In order to use static lighting, you need to create elements by hand instead of using provided blueprints.

Blueprints in this pack are very advanced and in order to edit them, you should have a good understanding about the Unreal Blueprint system first.

Before you buy this product, remember that you can always download a test build before that to see how it performs. You can find a link for that at the beginning of this documentation.

F.A.Q

Q. Is this pack going to work with static lighting?

A. Short answer is no. Because this pack is very dynamic and optimized, it's not possible to bake lighting for instanced meshes. In order to use static lighting, you need to create elements by hand instead of using provided blueprints.

Q. Level is missing shadows and what I need to enable in order to get all of the features working?

A. This package requires that you have the "Generate Distance Fields" setting enabled in your project settings.

Q. I'm seeing greenish cards on top of surfaces. What are those?

A. You need to enable the DBuffer Decals option to use d buffered decal materials. You can find this setting in *Project Settings -> Engine -> Rendering -> Lighting*.

Q. I'm using a custom player but for some reason I can't use items inside buildings?

A. Building system is using various box colliders to scatter and place assets but if your custom player/project contains custom collision channels, that might cause issues. In that case I would advise to open the **BP_P_Building** blueprint (parent class for every building) and then change collision settings for the "MainBBox" and "InteriorCheck" colliders.

Q. I can't move when I press play?

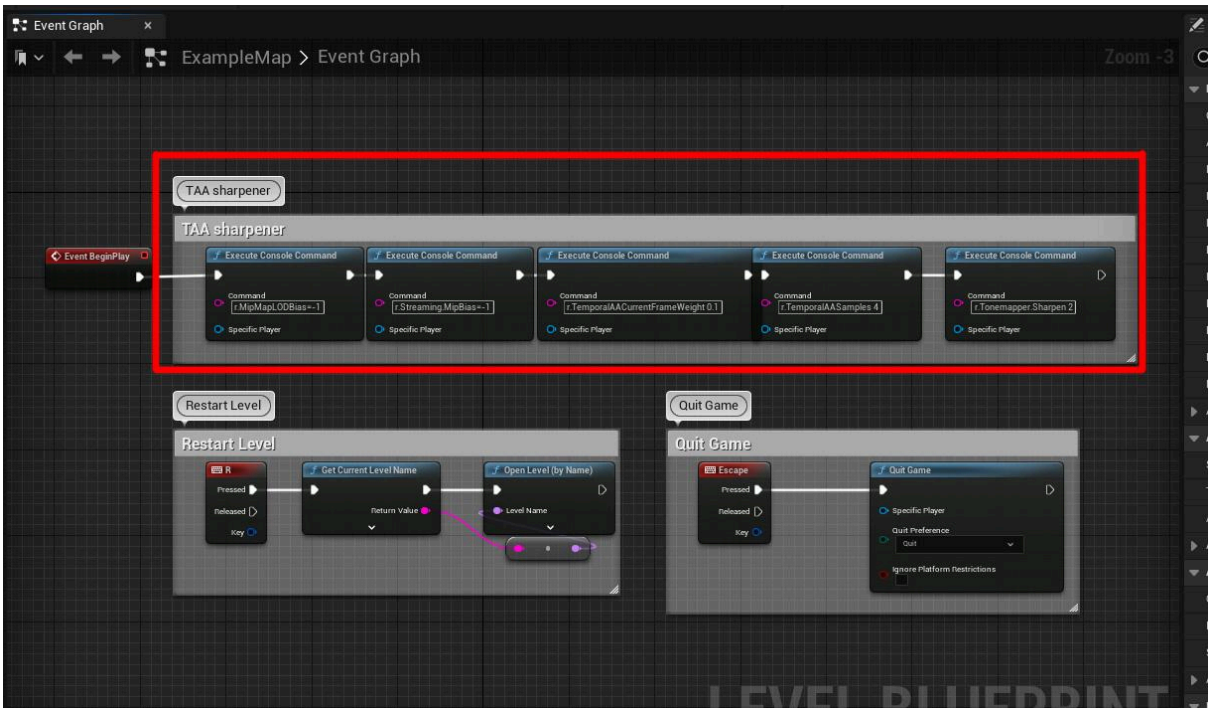
B. This can happen if you don't have correct input settings specified in your project settings. Look for the "*Example Player*" section to see what action and axis mappings you need to set up in the project settings. Alternatively you can also download the input.ini file.

Q. I'm getting errors and can't load levels in my game with these buildings?

A. If your level contains lots of building actors then you might encounter this issue. This is because Unreal needs to load buildings each time when you open levels and there is a certain loop limit that will cause issues if you go over that. This issue is more related to how Unreal Engine works but here are few workarounds to solve this. Look for the Level Load Error section for possible fixes.

Q. Picture is very sharp. How can I disable that?

B. Level blueprint contains some console commands that help to make the picture sharper when using regular TAA solution. If you don't like that effect then you can simply open the level blueprint and remove nodes from the *EventBeginPlay*.



Q. Opening the demo level takes a long time. What is causing that?

A. At the moment it's still unknown why UE5 versions of marketplace packs take so long to open but they will open eventually. I will update things when I'm sure what is causing this or if Epic

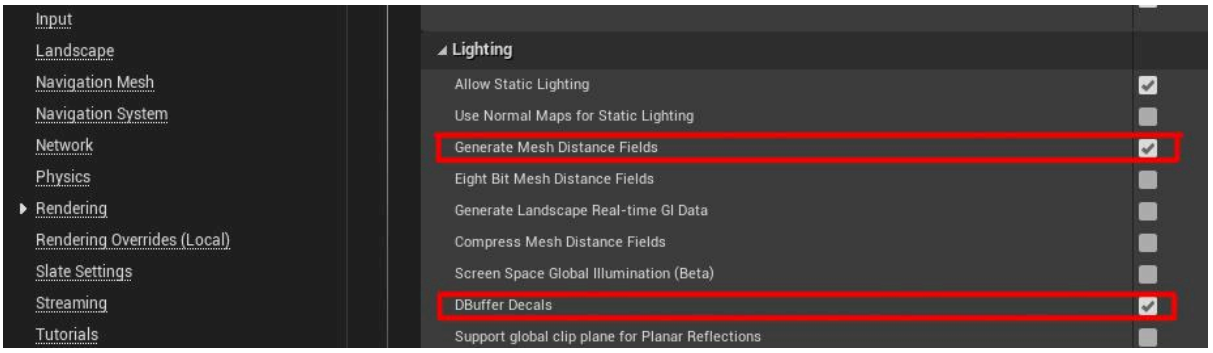
Games change something on their end. This issue is not happening in the UE5.1+ so it would be advised to update your engine and project before adding this pack.

Project Setup

Required Settings (UE4)

In order to avoid issues and use all of the features this pack can offer you should enable the following settings.

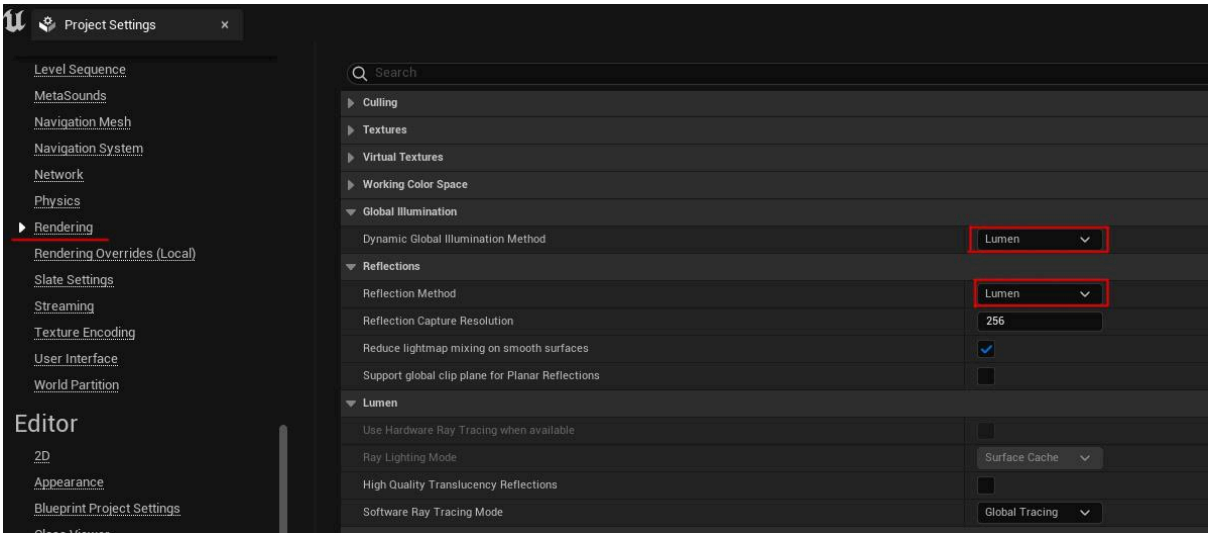
Example level is using distance field shadows and ao. In order to use these lighting features you need to enable the "Generate Distance Fields" setting. Some materials need the "DBuffer Decals" setting to be true too. Both of these settings can be found in **Project Settings** > **Rendering**. I would also suggest enabling SSGI (*Screen Space Global Illumination*) to get better visual results in Unreal Engine 4.



Required Settings (UE5)

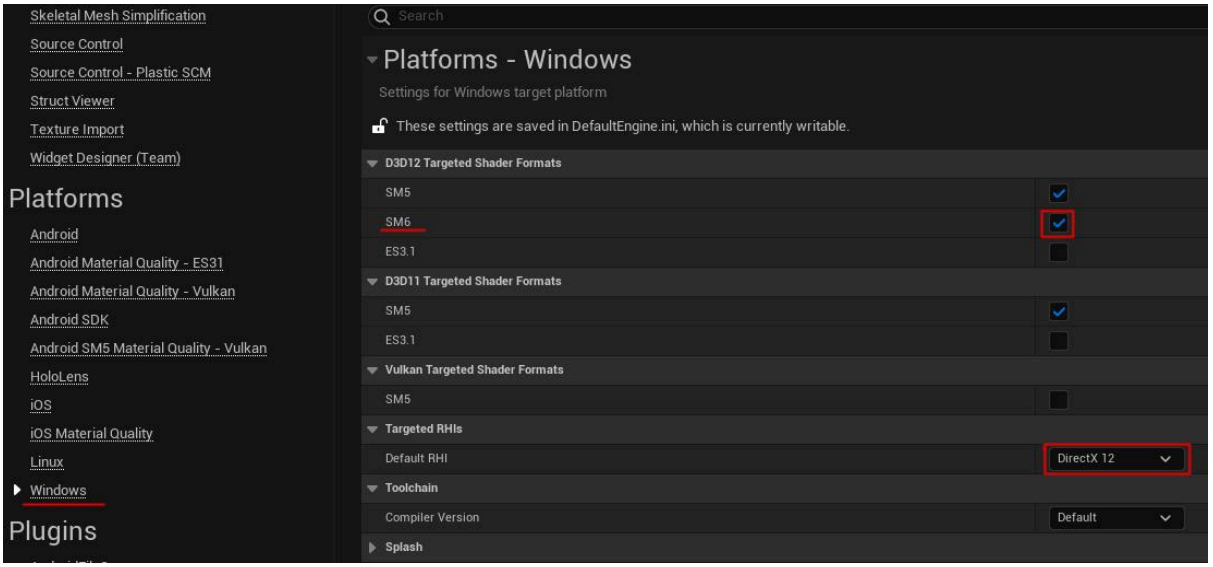
Lumen

Lumen is enabled from the Project Settings under the **Rendering > Dynamic Global Illumination** and **Reflections** categories. You can read more about Lumen in the documentation page. I would also advise to enable Virtual Shadow Maps to achieve a better shadow quality with Nanite meshes. You can enable it in the same place.



Nanite

If you created a new Unreal Engine 5.1 project, you should already have Nanite enabled. If that's not the case then you need to change a few settings in Project Settings under **Platforms > Windows > Targeted RHIs** and change *Default RHI* to DirectX 12. You also need to enable SM6 under **D3D12 Targeted Shader Formats**. After that you can restart the editor and Nanite should now work. You can read more about Nanite in the documentation page.

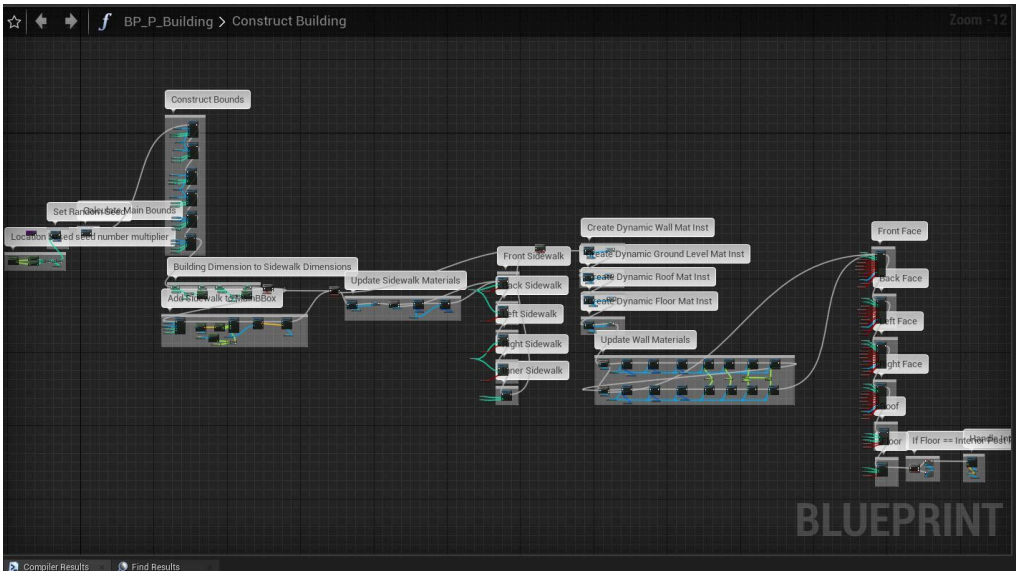


Building Systems

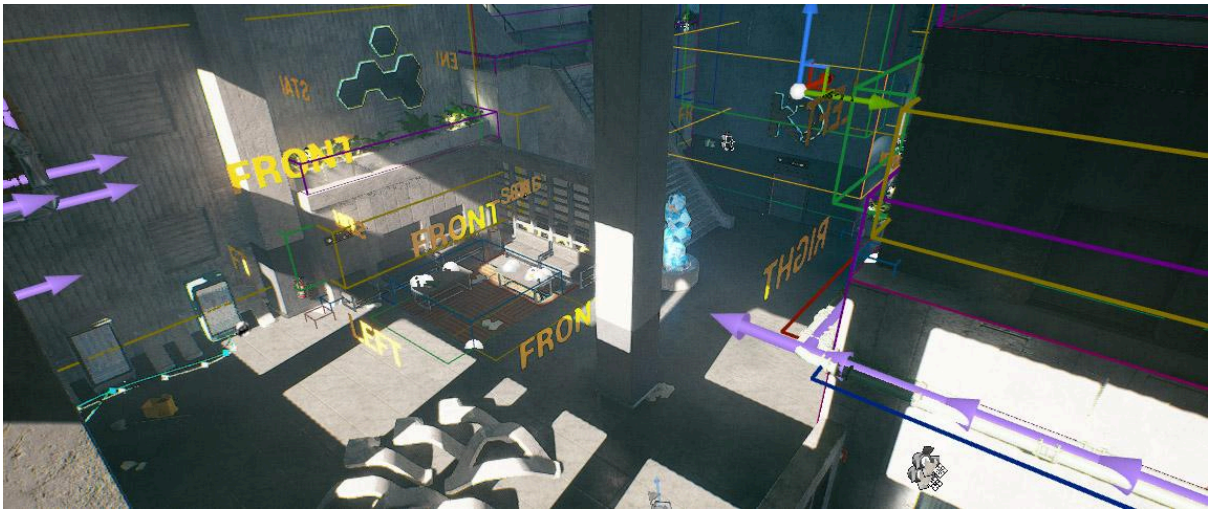
Overview

This system is constructing buildings using different building library sets. These sets contain building meshes that are modeled in a modular fashion to work well together. This way you

can control building size, colors, damage, windows/roofs/doors etc… Blueprint will then use HISM (*Hierarchical Instanced Static Mesh*) components to construct the actual building in an optimized way. You will also have an option to make it generate static meshes instead but this will have a huge performance impact in terms of draw calls.



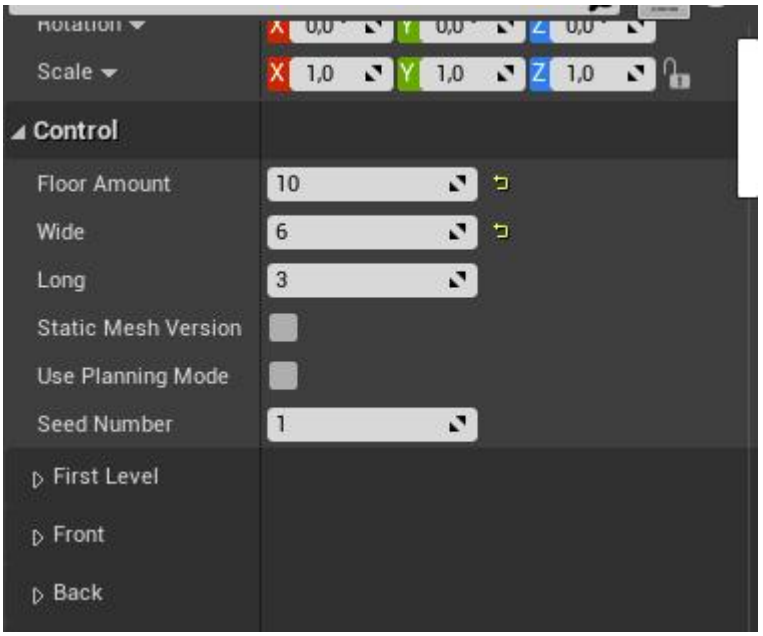
Main logic is in the **BP_P_Building** parent class. Every building is then inheriting that logic from this class. Some common variables are public so you can tweak them in the level and create more variations between buildings. Others are private that you need to change inside that child blueprint. Most of the time you don't need to do that unless you are creating new ones or want to make changes to each building instance.



Main Control

System is divided into different parts. First stage is the planning stage. In this part the system will calculate building dimensions and generate bounds that are later used for the actual building. You can find a variable called “Use Planning Mode” under “Control”. Turning that on will make it faster to work and edit buildings but you want to change that to false

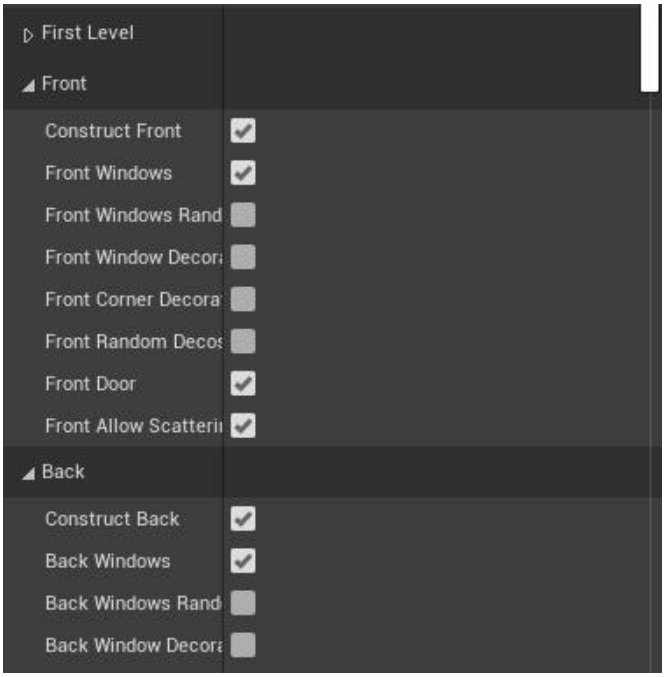
when you are done. This will then run the actual logic that generates HISMs and assign materials.



In the same place you find variables called “Floor Amount”, “Wide” and “Long”. These are used to control the building dimensions and are the most important variables in this system. Everything else is then created based on these. “Seed Number” is a handy way to generate different variations and keep those changes using a pseudo random logic. It will affect every random function that the system uses. This seed number is also changing based on the building world location so every building can be a unique one.

Building Walls

When bounds are calculated, the next stage is wall building. This is done for left, right, front and back sides and you can find individual control for every side under “Control”. You can choose to disable constructing specific sides totally, enable/disable windows and so on. These settings are the most used ones and can have a huge impact on how the building looks.

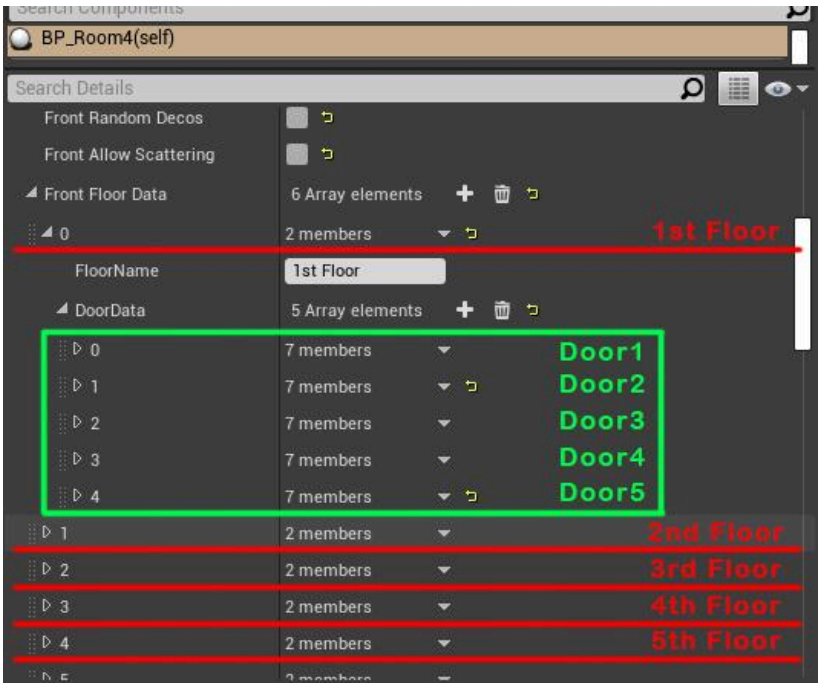


There is also a separate control for the “First Level”. You can enable/disable this part and control where the door would appear or just disable doors totally. “Door Offset” controls the door blueprint position that is different for every building due to different building dimensions. You can also spawn actors and controls for them can be found under “Actors”. Most buildings are spawning actors like vending machines, trash containers etc… If you don't want to spawn any actors you can turn “Spawn First Floor Actors” to false. Otherwise the system will try to find actors from the actors array.

Building Doors

During the wall building process, the system can also add doorways and doors based on the door array. This is basically a list of rules where the system will replace wall pieces with doorways. You have an option to specify different door blueprints or leave it to “*None*”. On top of this, you can also change door colors, locked status, names etc.. Every building face will contain their own door arrays so you can have the most optimal amount of control. Door location is figured out by using mesh sockets. Doorway meshes contain a socket with prefix “*Door_*” and you then use door offset values if you need to move door actors for specific door types.

Update 1.1 will bring a change that allows adding doors for each floor. You can control this for each building face by specifying values in the “Floor Data” array. This array first contains items to describe each floor that the building has. If your building's “Floor Amount” value is 5 for example, you can add 5 items into this array where item 0 = first floor, item 1 = second floor and so on. You can then expand each of these items and see that all of them contain an array called “DoorData”. You can then expand this array and control where to spawn doors for that specific building face on that specific floor.



Building Roofs and Sidewalks

Then there are controls for roofs. You can disable roof construction or choose to scatter roof decorations that are specified under “Meshes”.

If you wish you can also construct sidewalks. This will construct sidewalks using the building dimensions and then you can specify multipliers that will shrink or grow that system.

Floors/Interiors/Furniture

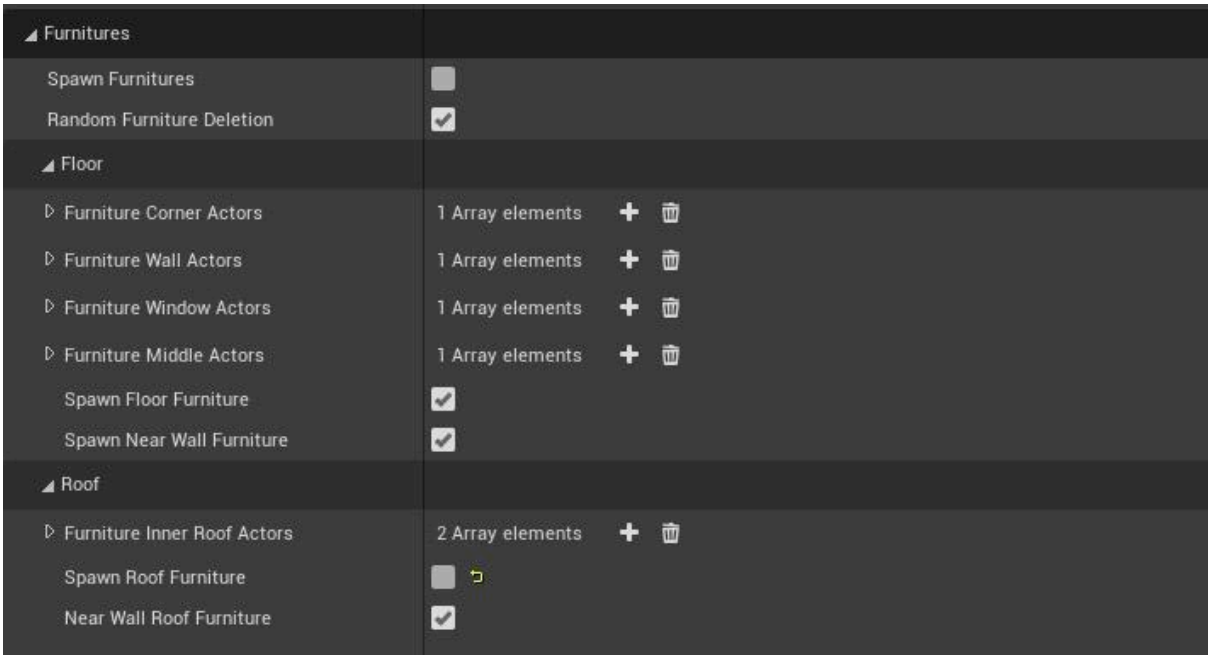
System can generate multiple floors, dynamic stairs, pillars and even some basic furniture placement. In order to use it you need to open the **Control** tab and then enable the "Construct Floor" boolean. This will tell the system to run the floor function. Then you can enable/disable the option "Allow Multiple Floors" that will construct floors based on the "Floor Amount" number. You can also control floor start and end cutoff values that are useful to specify how many floors the system will skip from start or end. "Floor Height" will determine the distance from the previous floor to the next one and usually you want to keep this same with the wall height that is the default value for each building type. "Floor Elevation" is useful to offset all of these floor levels. In some specific cases you might also need to offset the floor but most of the time you should leave "Floor Offset" to 0,0,0.

Under Floor settings you can also control Stairs settings. You can enable/disable stairs generation and control the stairs offset. "Random Stairs Location" will randomly choose where stairs will be placed for each floor but you can disable it and manually specify this location using the "Stairs Location" X and Y settings. System will spawn a child stairs actor with correct values to match the floor height value and you can change this actor to something else if you so choose.

Buildings can be very large so you might want to enable pillar generation. This will find each floor tile, measure its center point and then trace the pillar height based on the floor height. It will then spawn pillar start, middle and end meshes using the same HISM functions that the rest of the building system is using to optimize draw calls. Pillars are fully modular to fill any room height but you can control things like "Min Allowed Distance" to avoid adding pillars when the room height is too small. Variables like "Inner Pillar Tracing Distance, Inner Pillar Fill Offset and Inner Pillar Start Offsets" will control how the pillar system will handle its fill scaling. "Pillar Density" is useful to control different pillar patterns/density to get more interesting results and with "Random Pillars" and "Random Pillar Rotations" you can get even more unique looks.

System also contains a basic furniture spawning logic. Main idea is to have a control on what furniture actors are spawned in different locations. This system is divided into floor and roof systems that each have their own settings. "Spawn Furniture" is the main switch to enable or

disable the main furniture spawning logic. "Random Furniture Deletion" will randomly skip spawning furniture to achieve a more natural look. Then you can specify actor arrays for different cases like window, wall, corner, middle actors and roof. You also have finer control to offset furniture from walls, roofs and floor.

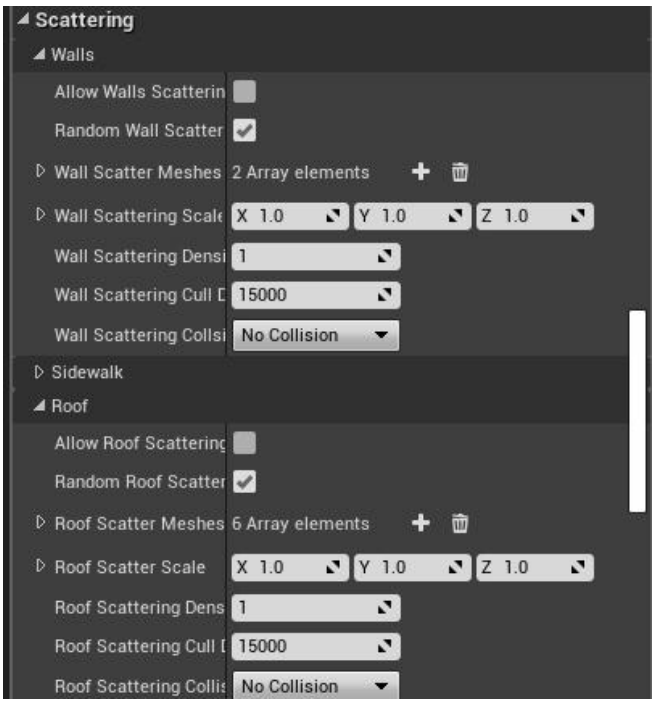


Traced Based Scattering

Scattering settings are controlling different layers of meshes that are scattered on top of buildings. You can choose to enable this system for walls, sidewalks, roofs and floors. Every layer is using the same structure variable so settings are identical for all of them.

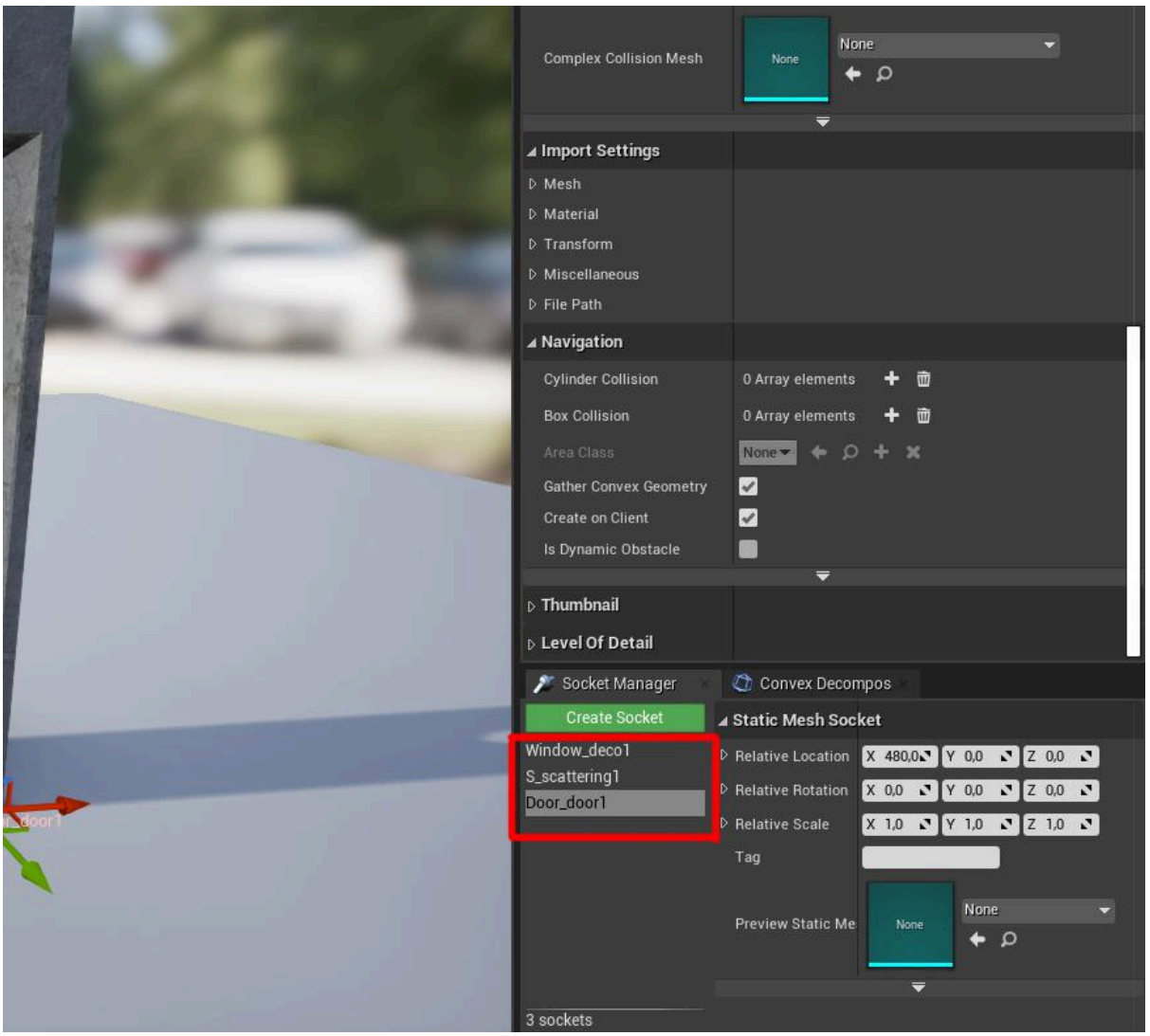
You can choose to use random rotations, apply different scales for scattered meshes, change culling distances, scatter density and even collision settings. System will look for meshes from the “Scattered Meshes” array and randomly choose different ones and put them into correct HISM components to save draw calls. You can simply drag & drop meshes and don´t have to worry about mesh instancing. System will handle all of that for you.

For example you can spawn some air pumps on walls with a specified scale and density, allow collisions and correct culling distance and then spawn smaller debris on sidewalk with smaller scale, without any collision, random rotations enabled and smaller culling distance and maybe larger scatter density. This way you can add lots of procedural details with minimal performance impact and still have enough control.



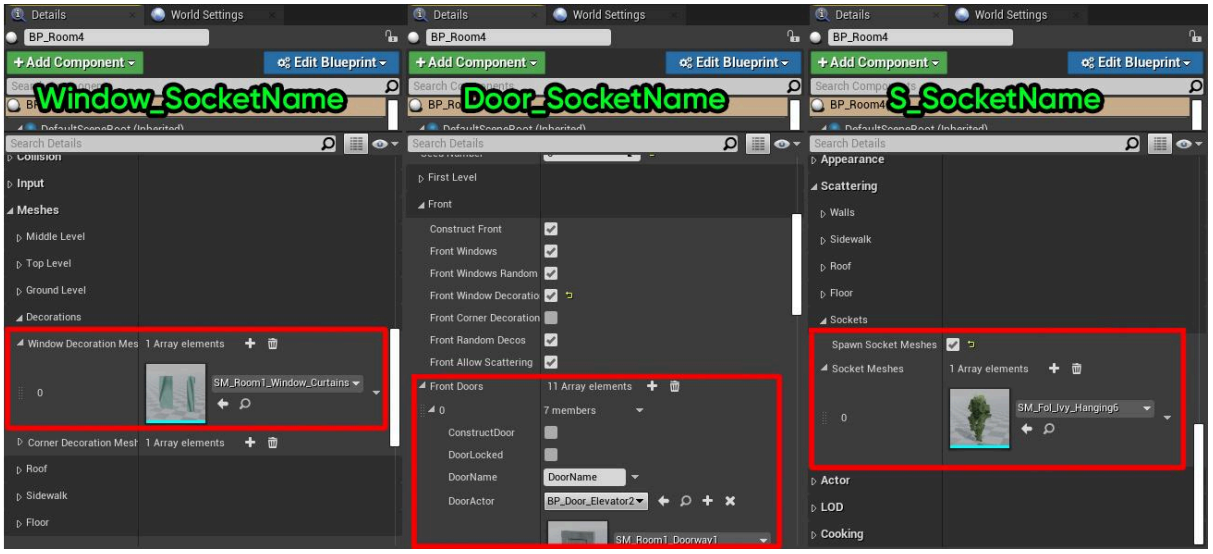
Mesh Socket Based Scattering

You can also use mesh socket based scattering. This can be found in the same “Scattering” tab under “Sockets”. You can specify a mesh array and the system will then check for certain building static meshes to see if those contain any mesh sockets. If that’s true then the system will start to generate instanced meshes into those socket locations. You can edit/add/remove mesh sockets simply by opening static meshes and this way control where to spawn those instanced meshes. This is perfect for scattering things like ivy, hanging foliage and such.



Socket names can also matter because one mesh can have many sockets that each spawn different things. Basic socket scattering (*Scattering/Sockets*) requires that the mesh socket name is **S_SocketName**. System will look for that **S_** prefix and you can have as many of those sockets as you like.

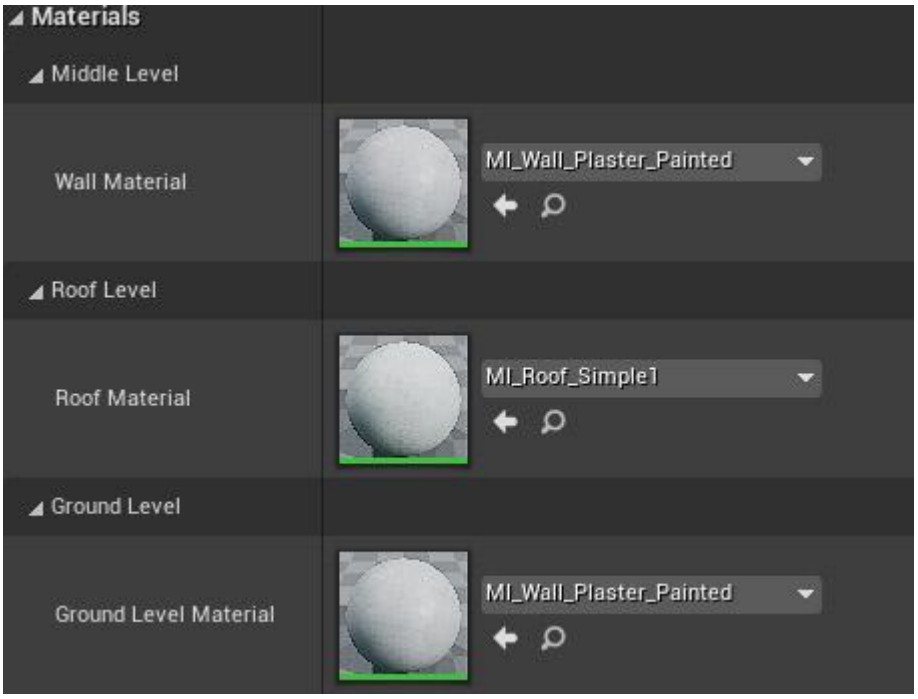
If you want that mesh to spawn window decorations (*Meshes/Decorations*) then that socket name should be **Window_SocketName**. Mesh sockets are also used to figure out door spawning location and that mesh socket name should be **Door_SocketName**.



This image will show what socket names correspond to different building system features.

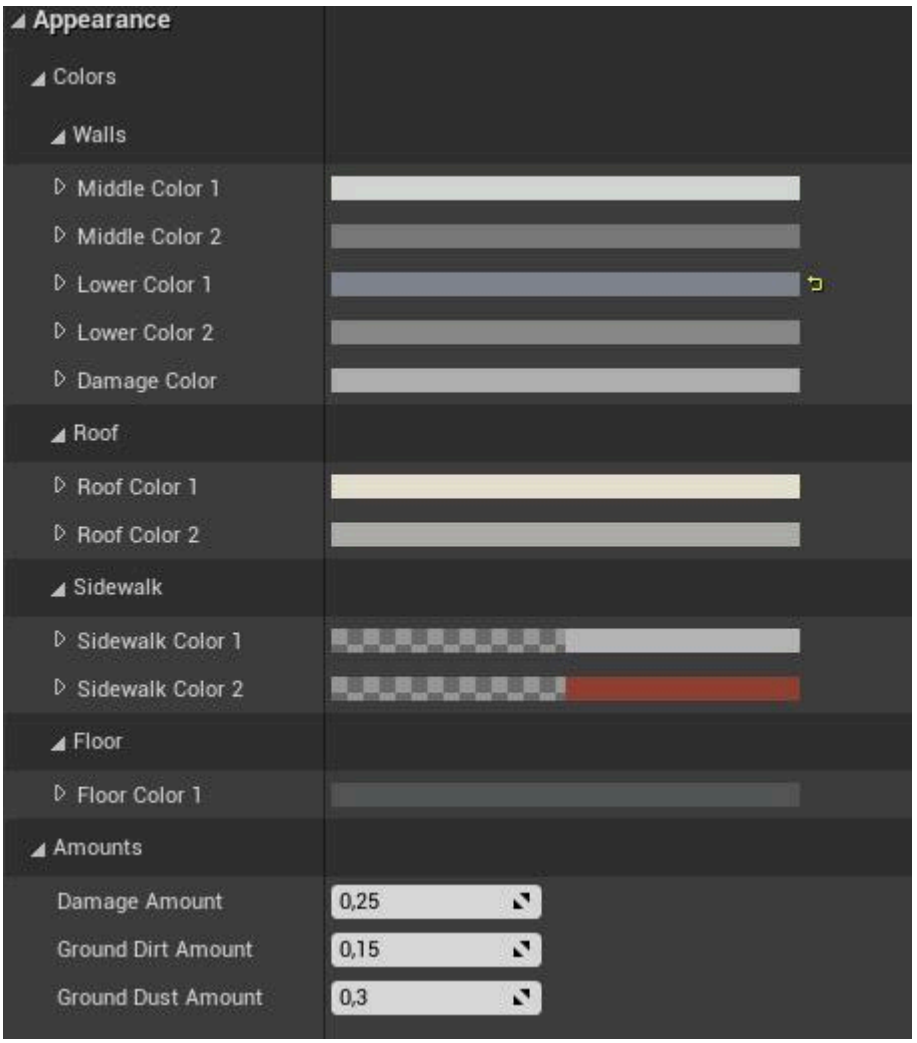
Building Appearance

Buildings are using materials that are specific for that building type. You can however change those if needed under “Materials”.



Remember if you are changing building materials, make sure those materials contain the right parameter names in order to change colors and damage values with this system.

Changing building colors is very easy. You can do that in the “Appearance” section and specify colors for walls (Middle Level), first level (Ground Level), roof and sidewalks. Under “Amounts” you can find settings to control the amount of damage and dirt buildings have.



Distant City System

Even though this building system is optimized it's not optimal to use this for far away city creation. If you need to add distant cities then you can use **BP_DistantBuildings** blueprint. It will scatter merged building meshes and randomly choose different building colors using material functions.

These merged buildings are usually costing only a few draw calls and the system is also instancing these meshes so it´s possible to create very large cities for background use with minimal performance footprint. Merged buildings are low poly meshes and are using low resolution textures so they are not optimal for situations where the player is able to get close to them.

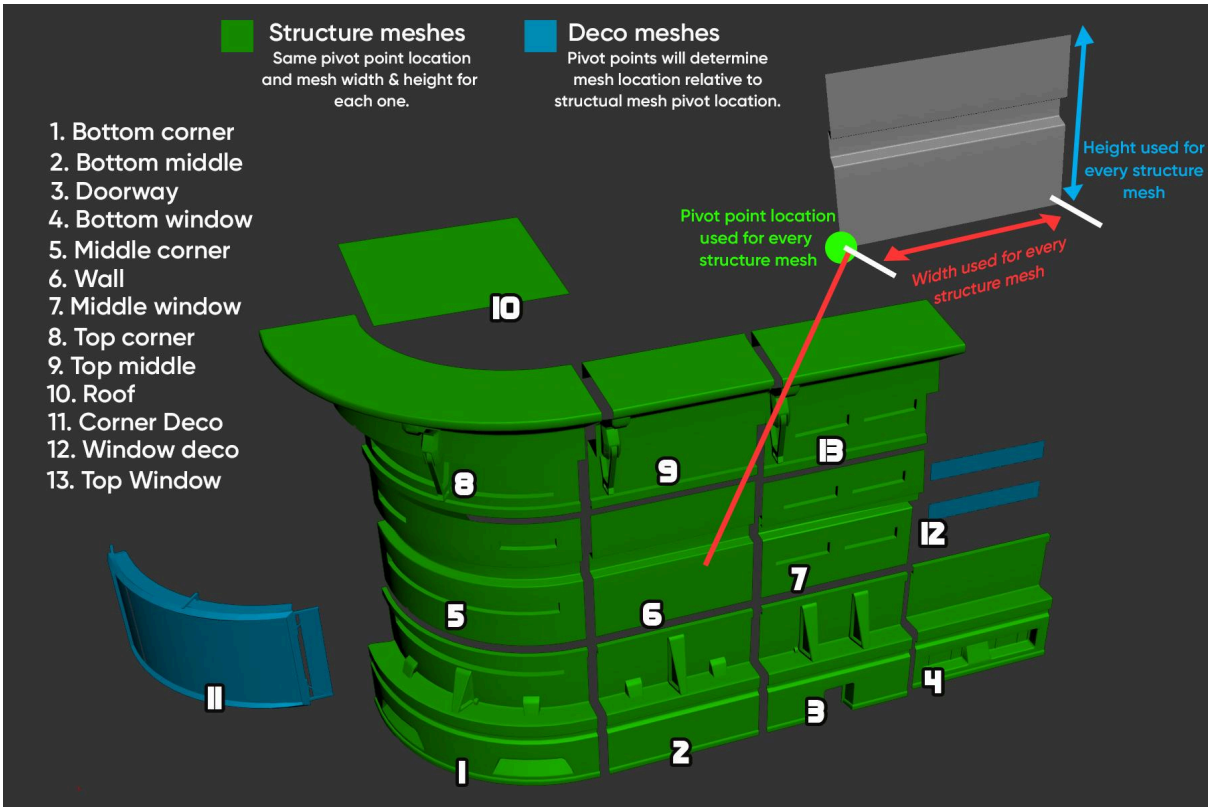
Custom Building Meshes

If you want, you can create custom building meshes to be used with this building system. It would be a good idea to create a new child class from **BP_P_Building** class and use it as a base for this new building type.

Buildings are using few structure meshes (green) and few decoration meshes (blue). This system is using wall mesh bounds (6) to figure out width and height for a single building slot. Building dimension variables are then controlling how many building slots wide and tall the building is going to be. Wider wall mesh means the building will be larger compared to narrower wall mesh using the same blueprint dimension values.

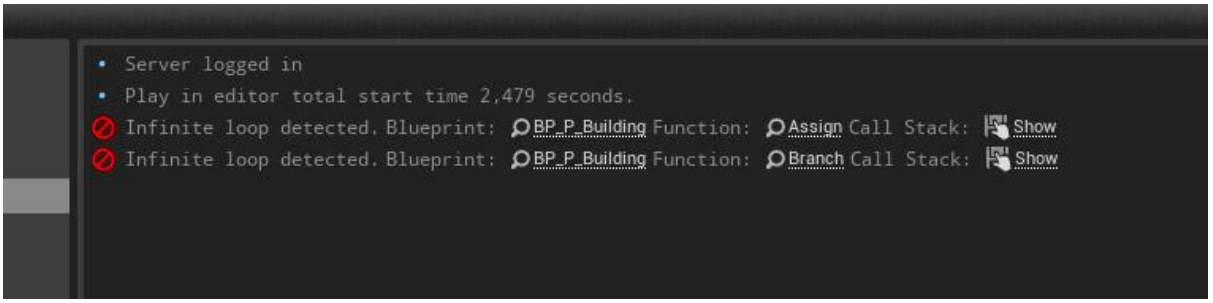
Every structure mesh needs to be modeled in the same width and height and pivot point location needs to be at the bottom left corner of the mesh. Make sure these meshes are also tiling correctly both vertically and horizontally. I advise you to use a grid when you are modeling so width, height and pivot point locations would match perfectly with each other. This way buildings can scale in every direction without issues.

Roof and corner meshes should have the same X and Y scales. For example if the wall mesh is 4 units wide the X and Y values should also be 4 for corner and roof meshes. This way there would be no overlapping or gaps between meshes. Roof mesh pivot point Z location will determine where the roof is placed and can be used to raise or lower the roof.



Decoration meshes are then placed on top of these structure meshes. That means decoration mesh pivot locations will determine the actual location where it will be placed relative to structure mesh pivot location. Best practice would be to model these deco meshes on top of structure meshes and then use the same pivot points with both of these. This way the results in Unreal Engine would match with results in our DCC application.

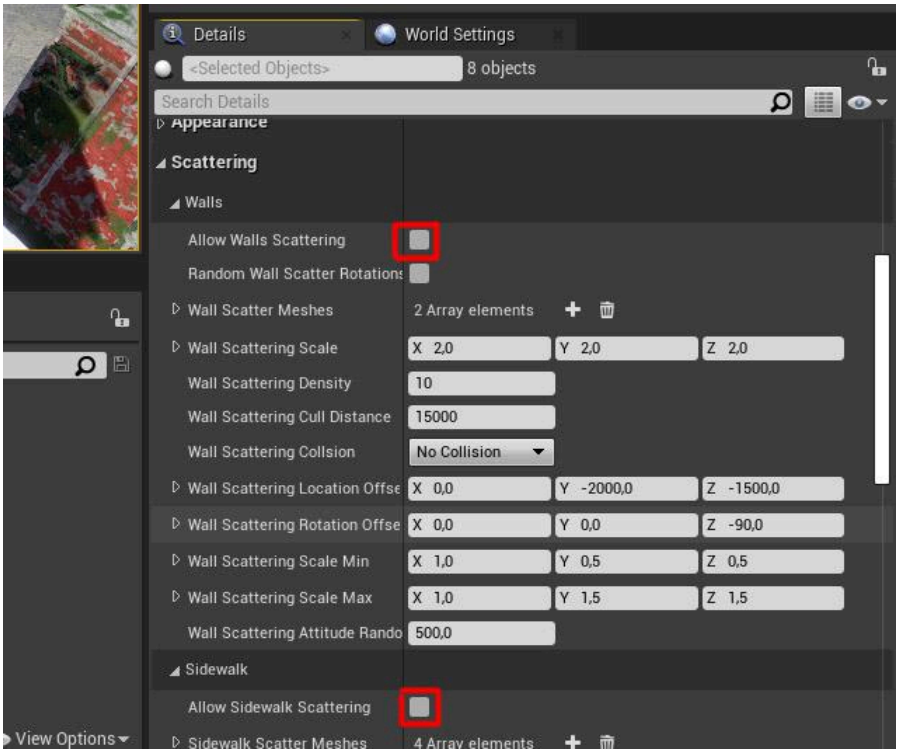
Known limitations



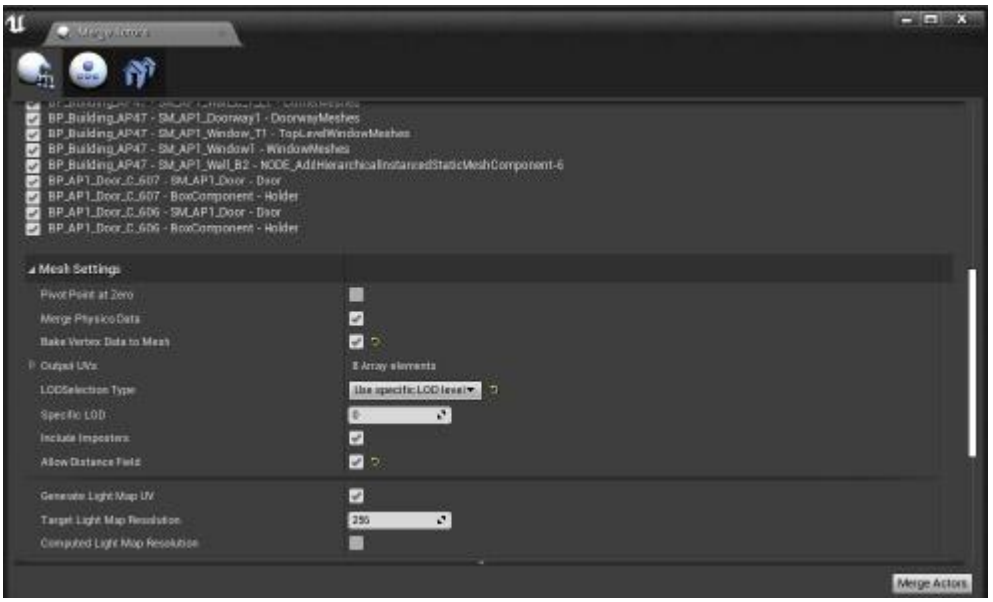
If your level contains lots of building actors then you might encounter this issue. This is because Unreal needs to load buildings each time when you open levels and there is a certain loop limit that prevents Unreal from going over it. This issue is more related to how Unreal Engine works but here are few workarounds to solve this.

1. *Reduce the amount of buildings you have in your level to reduce construction script loops.*

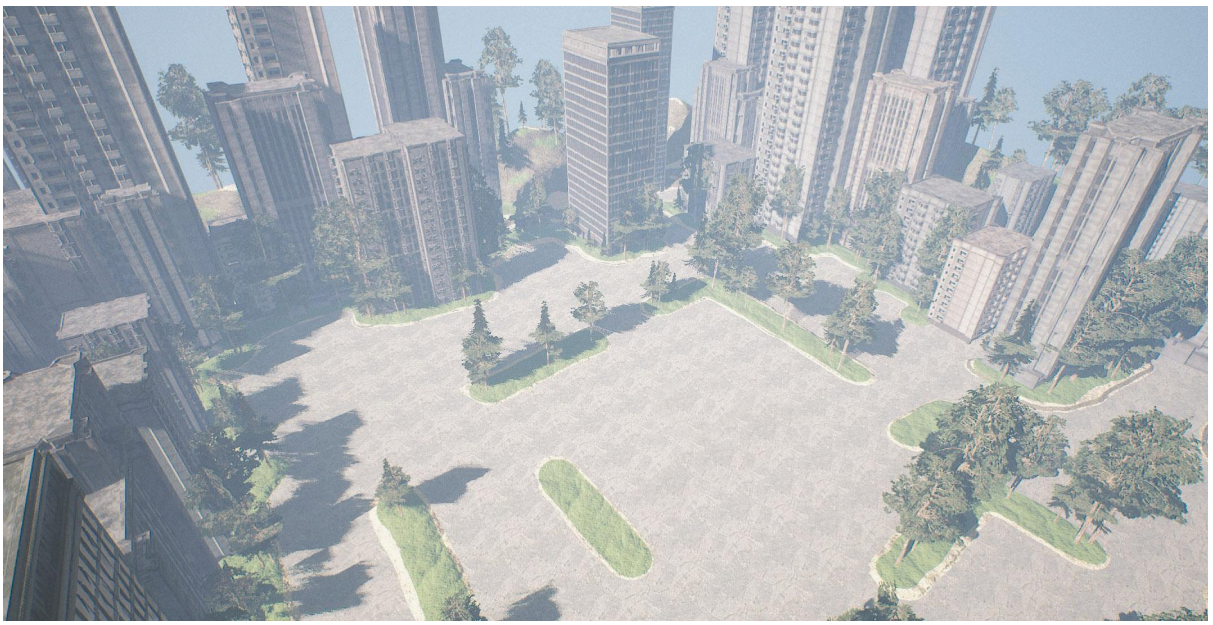
2. *Disable building features that you don't really need. Disabling wall, roof, floor and sidewalk scattering systems will reduce loops drastically. If this doesn't help then disable floor and socket based scattering.*



3. *Merge building actors down to simple static meshes. This way Unreal Engine doesn't have to load building actors or run any loops. You should first enable "StaticMeshVersion" because Unreal's merge tools works better that way.*



4. *For background/distant buildings you should use merged buildings or **BP_DistantBuildings** system that will generate and scatter simple meshes rather than more complex and heavier building actors.*



Lighting

This project is heavily relying on the dynamic lighting with distance field shadows and ao. Without distance fields enabled, lighting can look very dull and low quality so make sure to enable the “Generate Mesh Distance Fields” option in the project settings.

Splines

Overview

Spline blueprints are inheriting main functions from **BP_P_Spline** class. This system is using a spline curve to spawn different types of meshes. This system can be divided into different parts and it's possible to use all of them at the same time to have a very advanced system.

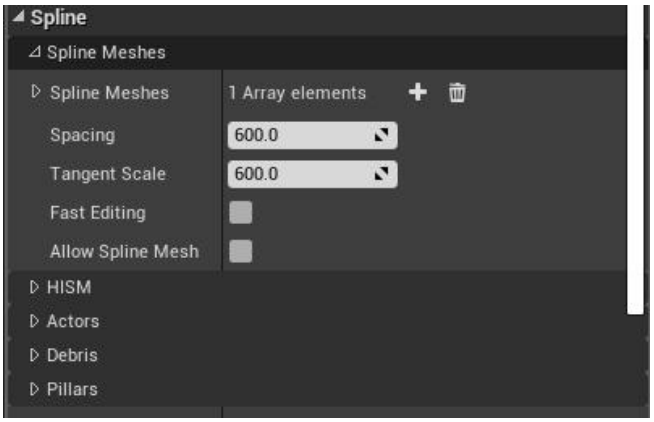
Spline Meshes

Basic use for this blueprint is to spawn spline meshes along the spline. This way meshes will deform and follow the spline curve accurately. You can read more about this system here.

“Spline Meshes” array will store meshes that are used for this system and you can specify them as much as you need. “Spacing” will control how many meshes the system will place along the spline. Using too low values will stretch meshes and too high values will flatten them. “Tangent Scale” will control the overall smoothness of the curve that we use for those meshes. Usually you want to keep this pretty close with the “Spacing” value.

“Fast Editing” is a way to make it faster to edit splines in the editor but you want to change that to false when you are done with editing. “Allow Spline Mesh” boolean will tell the system whether we use spline meshes or not.

Keep in mind that spline meshes can increase your draw calls dramatically. The more spline meshes you use the more draw calls it will generate. Using small meshes with high spacing values is the worst case scenario.



HISM Meshes

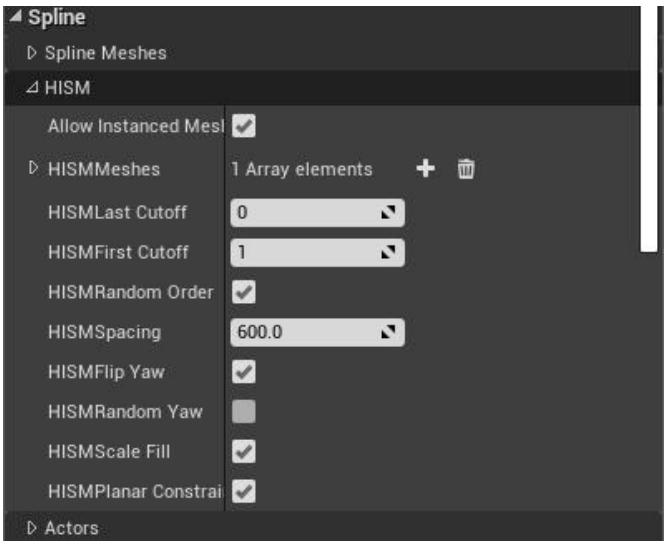
Using instanced meshes is a more optimized way because then we can batch meshes better. Unfortunately HISMs are not supporting mesh bending in the same way that spline meshes do. However there are lots of cases where we don´t really need to bend meshes so HISMs are a more optimal choice then.

You can find similar settings here like those that are used for Spline Meshes. You can enable/disable this function, specify what meshes to use, random order to use these. Then there are some settings to control rotations.

Cutoff settings allow you to cut mesh instances from start or end spline points. This is useful when there are unique meshes in start and end points and we don´t want to add HISM instances on top of those.

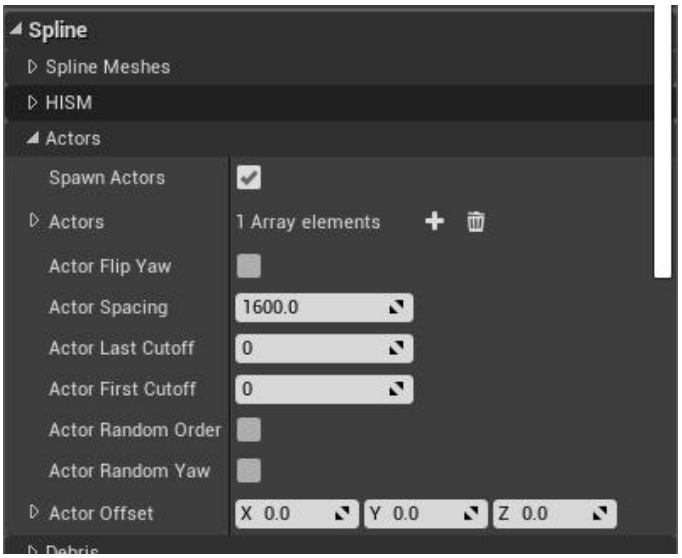
“HISMScale Fill” will help to fill gaps in cases where the mesh length will not match with the spline length. When this setting is true the system will scale the first mesh to fill this gap.

“HISM Planar Constraint” setting will constrain HISM instances onto a plane. This way it will keep z location the same for every instance. Useful for situations where meshes need to be on top of surfaces. Suitable for situations like roads, ground, floor and roofs.



Actor Spawning

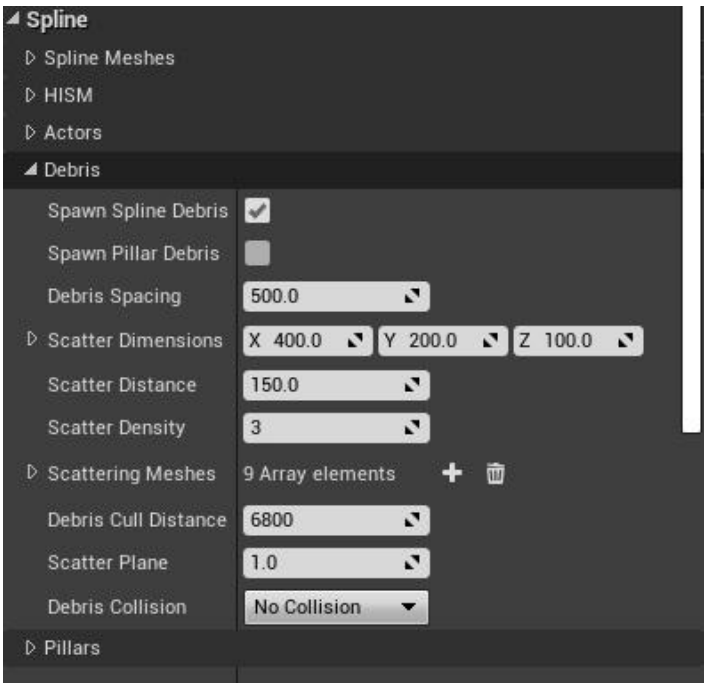
You can also spawn actors along the spline. These settings are almost identical with HISM settings. There is an array where to specify what actors to spawn and actor spacing value will control the density of how often these actors are spawned based on the spline length.



Debris Spawning

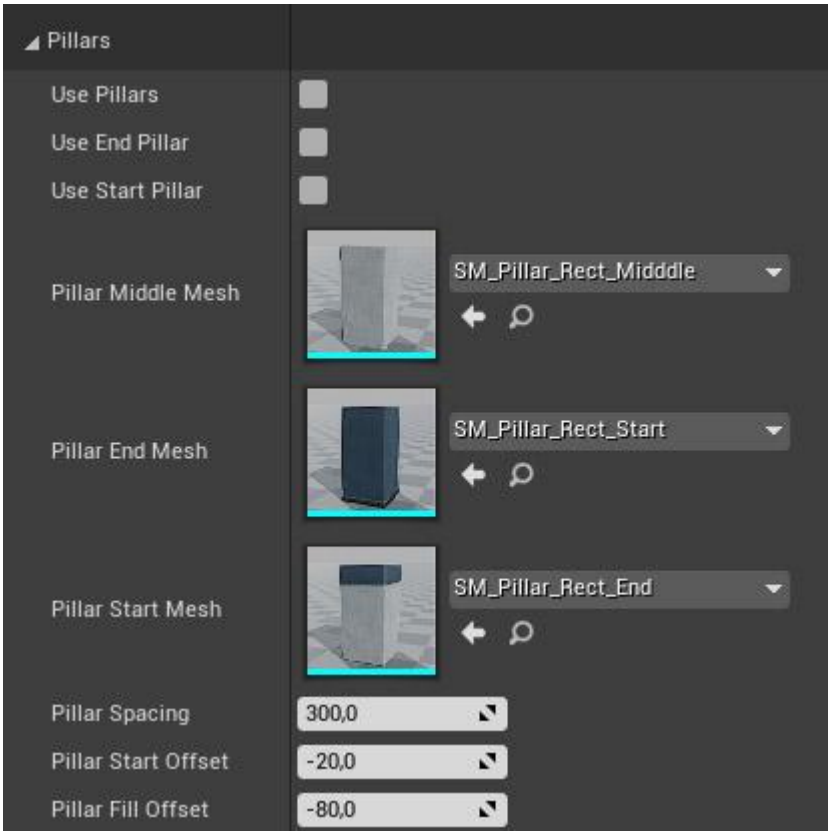
Spawning debris is also supported with this system. “Scattering Meshes” array will contain meshes that are going to be scattered. “Debris Spacing” value is controlling how often the scattering will happen along the spline and “Scatter Density” will control the amount of meshes that are spawned when that happens. “Scatter Dimensions” will specify how large the scatter area will be and “Scatter Distance” is controlling how far traces are going to go.

This system is creating HISM components under the hood to optimize performance footprint as much as possible.



Pillar Spawning

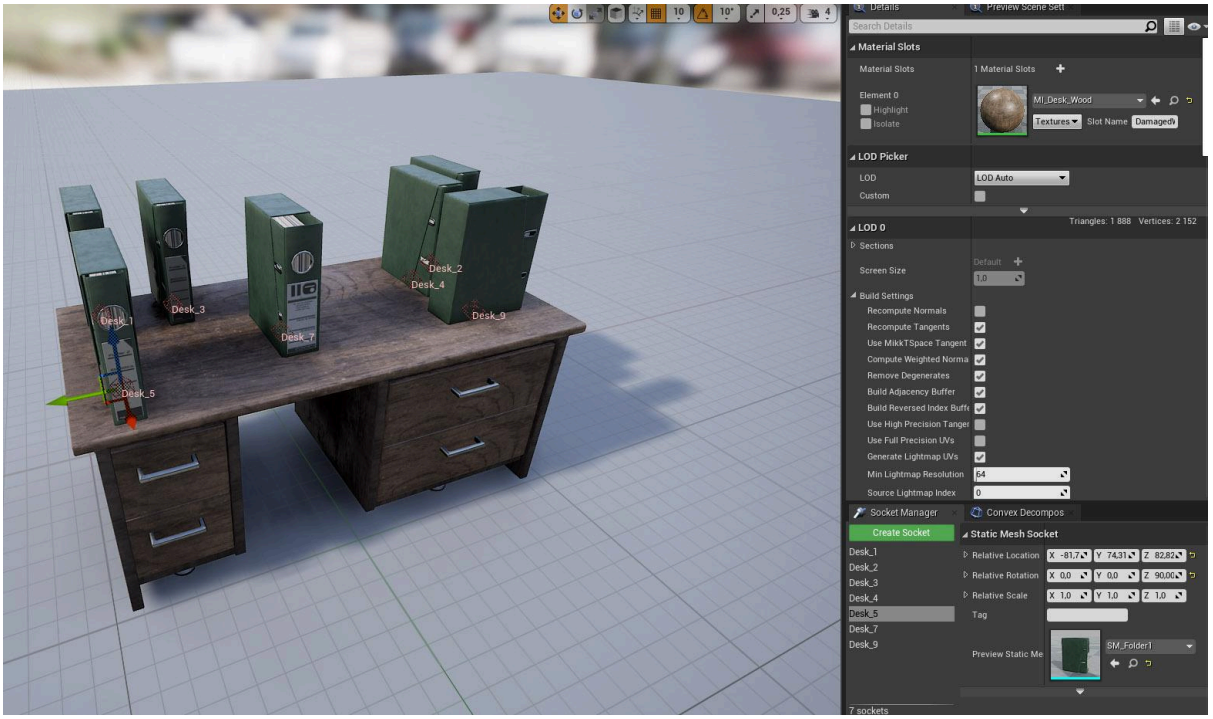
If you need to add supporting pillars you can turn the “Use Pillars” option on. This system will trace lines from locations that are taken from the spline based on the “Pillar Spacing” value. You also have extra control to specify whether or not to use end and start pillar meshes.



Start offset will allow the pillars to move up/down from the spline and “Pillar Fill Offset” is used to control mesh scaling to avoid gaps or mesh overlapping.

Mesh Socket Based Scattering

Spline system also includes the same mesh socket based scattering system that is also in the building system. You can specify a mesh array and some basic control like min and max scale etc.. System will then check for each building HISM component static mesh to see if that contains any mesh sockets. If that’s true then the system will start to generate instanced meshes into those socket locations. You can edit/add/remove mesh sockets simply by opening static meshes. This is perfect for scattering things like ivy, hanging foliage and such.



Scattering Blueprint System

Scattering blueprint **BP_P_Scattering** is basically using two different systems to find transforms where to add instanced meshes. These systems are line traced based and plane based.

Line trace based is more accurate but will be a bit slower to update when changing values in the editor. Plane based will just take random points from a plane and will ignore the actual environment. This is faster and works well when you need to scatter something on a flat surface. You can change between these two modes with the “ConstraitPlaneMode” variable.

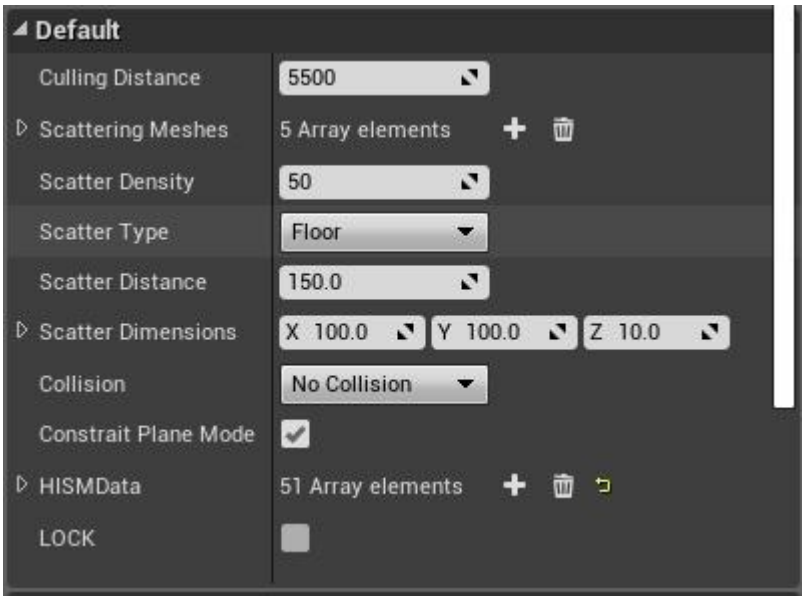
“Culling Distance” is controlling when to cull mesh instances. Smaller value means that instances will be culled faster and a value 0 means instances will not be culled unless culling volumes or systems similar to that culls it.

“Scattering Meshes” array will be used to figure out what meshes the system will scatter. It will automatically create HISM components for each unique mesh type and then store the same type of meshes there. This will keep draw calls at minimum.

“Scatter Density” will control how many times the system will run its loops and scatter meshes. “Scatter Type” enum controls how to align meshes onto surfaces.

“Scatter Dimensions” vector variable is specifying the actual scatter area where the system will take points for tracing. You can also change collision settings whether or not to use collision.

Instanced meshes need to be initialized every time the level loads and this system also uses pseudo random functions.



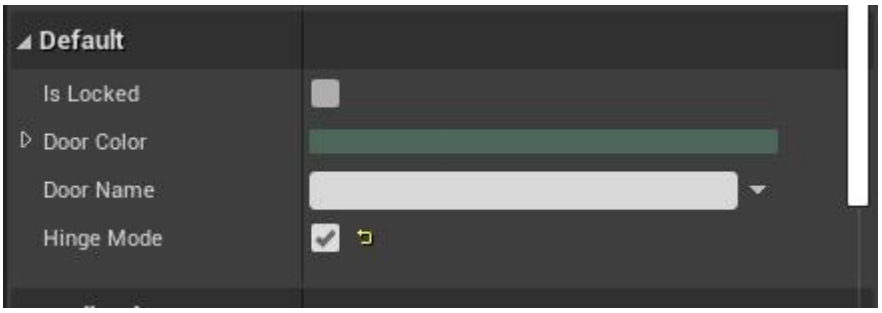
You can also use this blueprint as a child actor. This way you can for example scatter cans from vending machine blueprints etc.

Doors

Door Modes

This pack contains different door blueprints. All of them are inherited from the **BP_P_Door** actor that contains systems to handle door lock status, colors etc. By default, doors will use a timeline node to play open/close animations. This way it’s possible to control how that animation will play. Doors can also use hinge mode. This means that the door will be a

physics object that is using a constraint so the player is able to push them open. You can enable this by checking the “HingeMode” boolean to true.



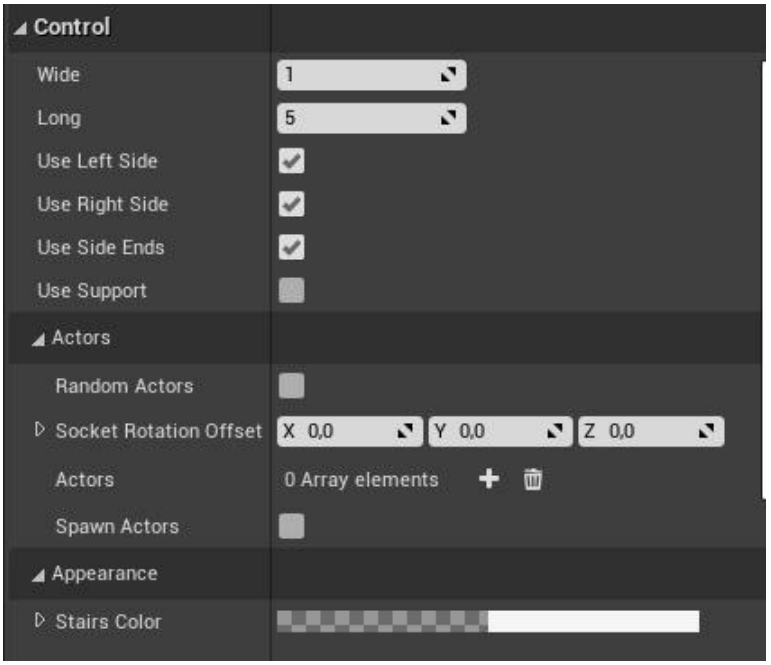
You can also change door colors and disable doors from opening by enabling the “IsLocked” boolean. You can use these doors alone or you can use them with the building system.

Stairs

Main Logic

Stair system is constructing stairs using a basic logic that will allow the use of different stair mesh libraries. Basic unit is one meter so it’s possible to cover all sorts of scenarios with this system. System will automatically handle mesh instancing so stairs will be as optimized as possible in terms of rendering. You can find these stairs actors in *Blueprints/Environment/Stairs.*

Wide and Long variables are controlling the overall dimensions. Pivot point is automatically adjusted to be in the correct place to make it easier for level designers to place stairs. You can also choose to construct left (UseLeftSide) and right (UseRightSide) sides like handrails and event support (UseSupport) that will fill the underlying part of the stairs.



Actor placing

In some cases you might want to add actors like lights. You can do that by enabling the “SpawnActors” boolean. Then you can specify an “Actors” array and the system will randomly choose those actors. Placement is using mesh sockets that you can specify for every stair static mesh using the Unreal static mesh editor.

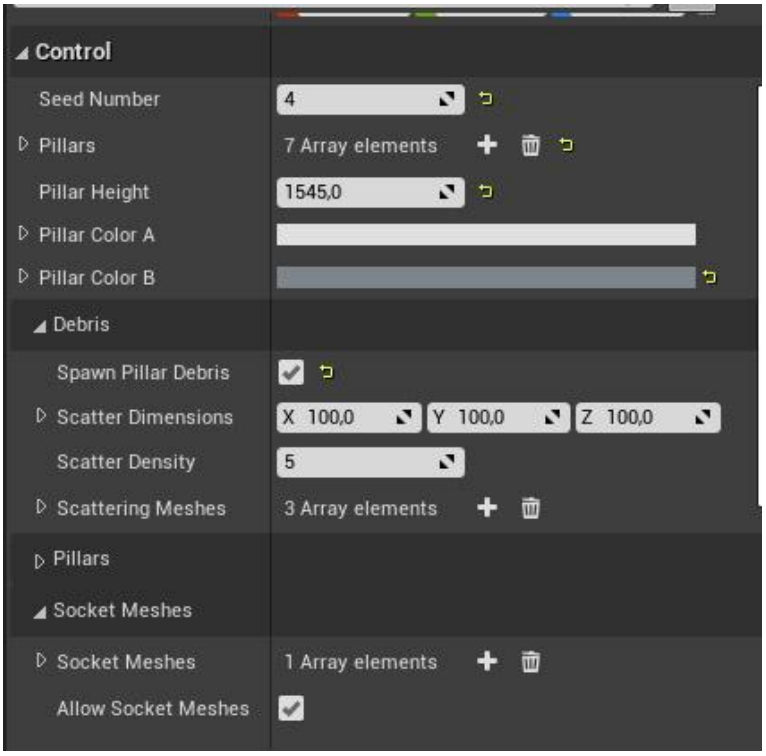
Appearance

Stair colors are also possible to change depending on the situation. This will affect the whole stairs actor and requires that stair meshes are using the correct material that is inherited from the M_Structure material.

Pillars

Main Logic

Building system already includes an automated pillar system but this pack also includes separate actors to manually add pillars (*Blueprints/Environment/Pillars*). It will basically use an array of points (Pillars) in the level to spawn pillars and pillar height will be controlled with the “Pillar Height” variable. You can change pillar meshes under the “*Pillars*” tab.



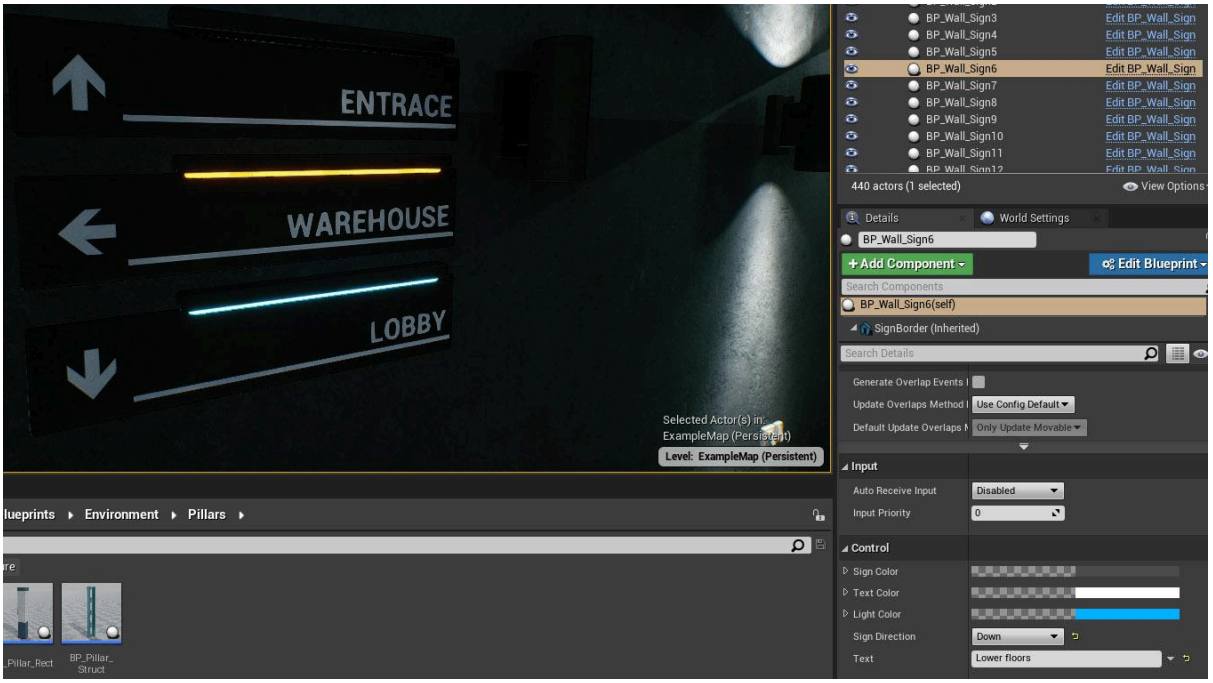
Extra Control

You can specify debris meshes that this system will scatter near pillars. Enable the “Spawn Pillar Debris” boolean and specify “Scattering Meshes” array. After that you can increase/decrease debris amount with the “Scatter Density”. “Scatter Dimensions” will control how large the scattering area will be around pillars.

System can also add socket meshes (SocketMeshes) that will use static mesh sockets that you can specify for every pillar mesh using the static mesh editor. “Seed Number” will help to generate different outcomes and it will affect every aspect in this system.

Signs

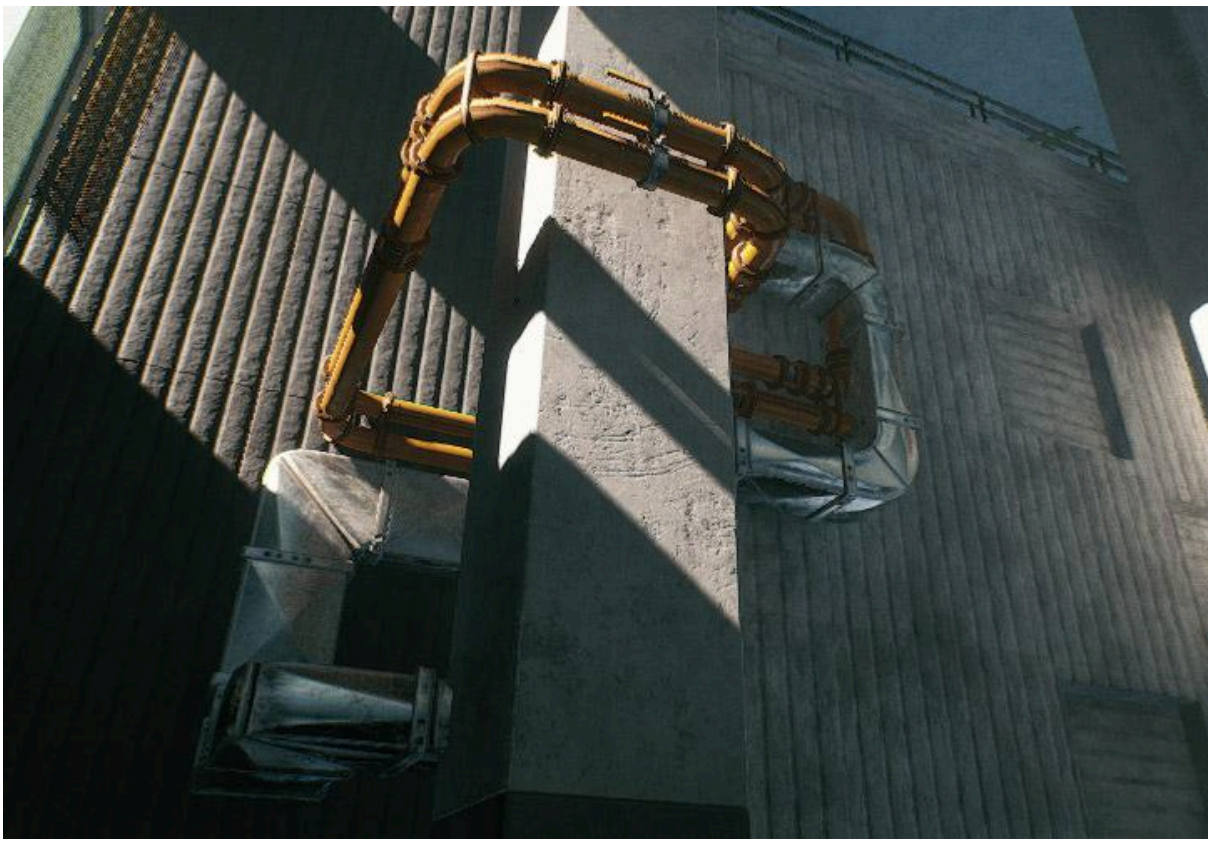
This pack includes a generic sign system. It will allow you to add signs that help users to navigate in the environment. You can specify sign color, text color, light color, text and arrow direction. This can be found in *Blueprints/Environment/Signs.*



Pipes

Main Logic

Pipe blueprints will help to add and construct pipes. It allows you to use fully random and automatic mode (Automatic Mode) but you can also have a manual control. Most of the settings are easy to understand by changing values and seeing how the system changes outcomes based on that.



Control

“Locked” boolean will allow locking current results when “Automatic Mode” is enabled. “Iterations” variable will be used with automatic mode and that will control how long the pipe will be. Higher values will make it more expensive. “Tracing Mode” will use line tracing to see if something will block it when using automatic mode. Enabling this will make it more expensive. “Alignment Mode” will constrain pipes, free will allows pipes to go in every direction, planar will force pipes to be on a plane etc. This option will only work in automatic mode.

“Static Mesh Version” will force the system to construct basic static mesh components instead of HISM components. “Start Rotation” will help to rotate pipes when using automatic mode. “Trace Distances” array will be useful to add random pipe lengths. “Given Directions” array will contain given rules on how the system will construct pipes (left, right, up, down, straight). Manual and automatic modes will both result in these rules that you can manually change, add or remove.

Manual Control

When you select a pipe actor in a level, you can find buttons like *LEFT, RIGHT, UP, DOWN* under the "*Default*" tab. This way you can control how the pipe will be constructed. *UNDO* button will remove the last action and *CLEAR* will clear every action you have done. You can first use automatic mode and then continue with this manual control by disabling the "Automatic Mode" boolean under "*Control*".

Optimizations

If you need to optimize performance even more, here are a few tips for that.

Remember that in this case optimizing will disable some features that can have a huge visual impact.

One thing to consider is grass density. Grass is using landscape layers and is a part of the landscape grass system that you can see in the distance. Foliage setting in Engine Scalability is also affecting how dense the grass is and this can have a huge impact on overdraw. You can find more info here.

Drawn distance will increase/decrease the amount of visible meshes on the screen. This can have a huge impact on the performance in terms of the draw calls and tris count. Default values are set up to work well with the "Epic" view distance option but lowering that can give you a good fps boost with minimal object popping.

Master materials are already using quality switches to disable some heavy features when material quality is changed to low. You can also turn various layers off from the material instances to save instructions. You can first do this in the material instances and then maybe even bypass those nodes in the master material.

Post process effects can eat performance and you can turn off features that you don't need. Screen space reflections can cost a lot so you can decrease the “Quality” setting or disable this feature completely by changing “Intensity” to 0. You can do the same thing with ambient occlusion. Both of these settings can be found in the post process volume (PostProcessVolume).

Depending on your needs you might want to use ray tracing. This is something that will increase the performance footprint dramatically.

In Unreal Engine 5, Lumen can cause some performance issues with lower end systems so you might want to lower its settings down or disable the whole system completely. You can do this by selecting the post process volume in the level and then set “Global Illumination Method” to “None”. Alternatively you can also do this in the Project Settings. You can also disable Virtual Shadow Maps in the same place.

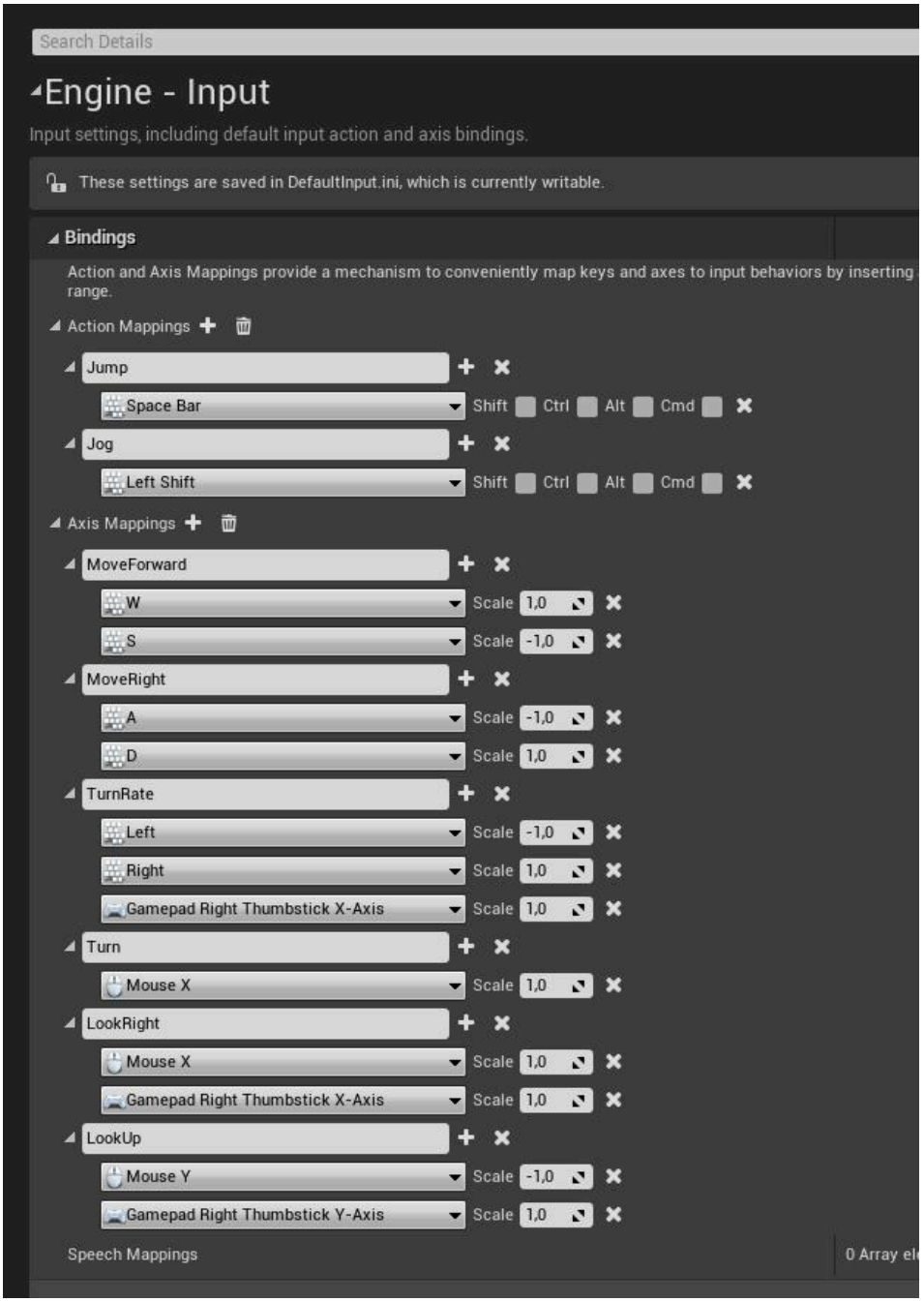
It’s always a good idea to profile if you are mixing these assets with something else to see what is the part that is causing performance issues. In that case you can use Unreal’s debugging tools. You can read more about that here.

Example Player

This pack also comes with an example player blueprint. It is based on the basic Unreal Engine first person pawn with some small changes like an option to switch between first and third person views.

It will support basic WASD movement, sprinting (Left Shift), crouching (C), object picking (E) and interaction.

Because this product is an asset pack, you need to manually specify these key mapping settings in the “Project Settings” in order to control the player. Alternatively, you can also download input settings and simply import them into your project (Choose “**Import**” on the top right in the Project Settings > Key Binding tab).



Demo

In the demo you can test the whole example map. It will contain everything that comes with the product. Inputs are:

Movement: **WASD**
Sprinting (**Left Shift**)
Crouching (**C**)

Enter vehicle (**E**)
Object picking (**E**)
Restart level (**R**)
Exit game (**ESC**)
Developer console (**TAB**)

Any feature requests or questions? Please feel free to ask them
(kimmo.koo@hotmail.com)

Join to the Discord server
KK Design support policy

More Unreal Engine assets be found here
