Post Apocalyptic City Pack - 5.1+

Product Link: Post Apocalyptic City - Unreal Marketplace
Demonstration Video: Post Apocalyptic City - Demonstration
Screenshots: ArtStation
Demo Build: Demo Build

Release Notes

To see older release notes, please visit the older documentation page.

2.0 (Unreal Engine 5.1+)

- Added new example player (based on UE5 mannequin) + Enhanced Inputs
- Added rock cliffs + new advanced rock master material
- Added packed level actors (buildings, rocks, props)
- Added pipe generator blueprint
- Added pillar generator blueprint
- Added ventilation shaft assets
- Added more detailed ground assets
- Added more detailed prop and debris assets
- Added more house furniture assets
- Added a drone pawn

- Example map is now using World Partition
- Example map lighting re-iterated
- Example map foliage re-painted
- Example map local exposure tweaks
- Example map area/landscape expanded

- Building system now allows to add any number of doors
- Building system now allows to spawn static mesh actors
- Building system now handles HISMs more optimal
- Building system allows to add individual windows (better for Nanite)
- Scatterer system re-iterated + static mesh actor option added
- Stairs system re-iterated + static mesh actor option added
- Spline system re-iterated + static mesh actor option added
- Number of building material instances reduced (optimization)

- Grass meshes are now using individual grass strands
- Tree meshes are now using more detailed leaves and trunk assets
- Trees are now using pivot painter based wind for extra realism
- Meshes with masked materials are now turned into Nanite meshes

- Pre-made packed level actor assets created for buildings
- Pre-made packed level actor assets created for rocks
- Pre-made packed level actor assets created for certain splines
- Pre-made packed level actor assets created for outer assets

- Vehicle assets re-modeled to achieve next gen results
- Vehicles are now split into parts to work better with Nanite
- Vehicles now contains a modular system that allows to open doors
- Vehicle wheels are split for extra modularity
- Vehicle material is using sub layers for better results and extra control
- Vehicle parent class is more unified, appearance options collapsed

- Foliage material re-iterated + specular, base color control added
- Parent structure material re-iterated + added more unified top layer

- Fixed wooden stairs missing polygons
- Fixed tent material occlusion material bug
- Folder hierarchy re-organized

2.0.1 Hotfix (Unreal Engine 5.1+)

- Material sampler counts reduced to avoid issues in UE 5.2
- Sidewalk blueprint now construct light pole colliders
- Better procedural tree settings

2.0.2 Hotfix (Unreal Engine 5.1+)

- Parent building blueprint system refactored and null checked to avoid warnings/errors
- Parent spline blueprint system refactored and null checked to avoid warnings/errors
- Parent scatterer blueprint system refactored and null checked to avoid warnings/errors

Important information

This version of the pack is not 100% backward compatible. If you wish to update your existing version of this pack, it is highly recommended to clear your Vault Cache, remove

any existing Post Apocalyptic City assets and then download/import this version of the pack into your project. This way you will avoid issues that can happen otherwise.

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.

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 in the beginning of this documentation.

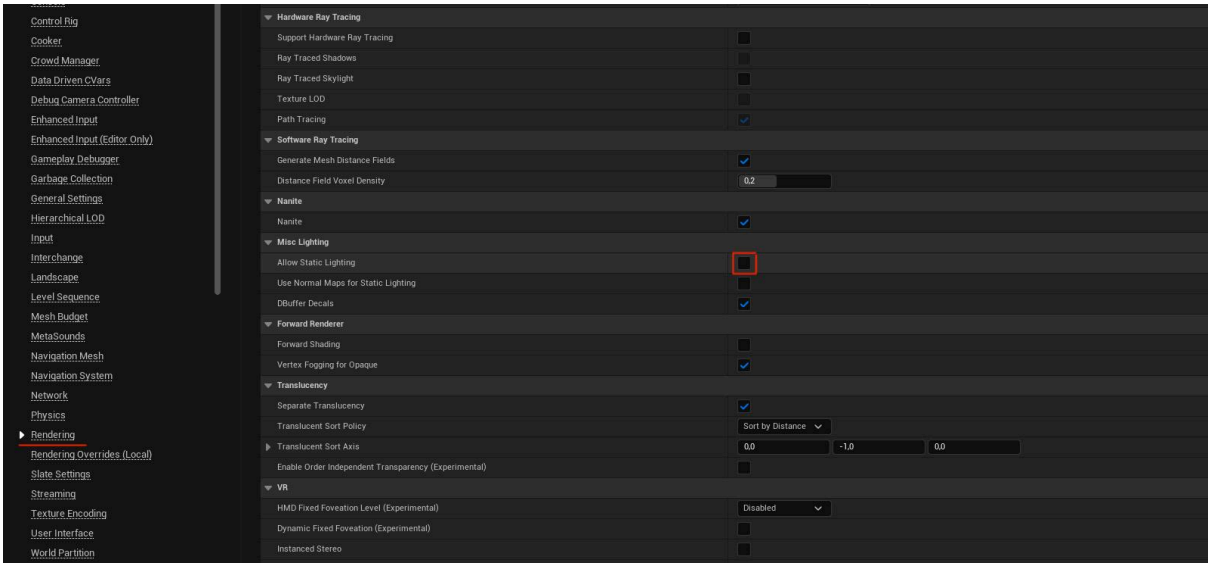
If you have any questions, issues/problems, please, let me know first before you leave reviews. You can contact me at (kimmo.koo@hotmail.com) or join our Discord server.

F.A.Q

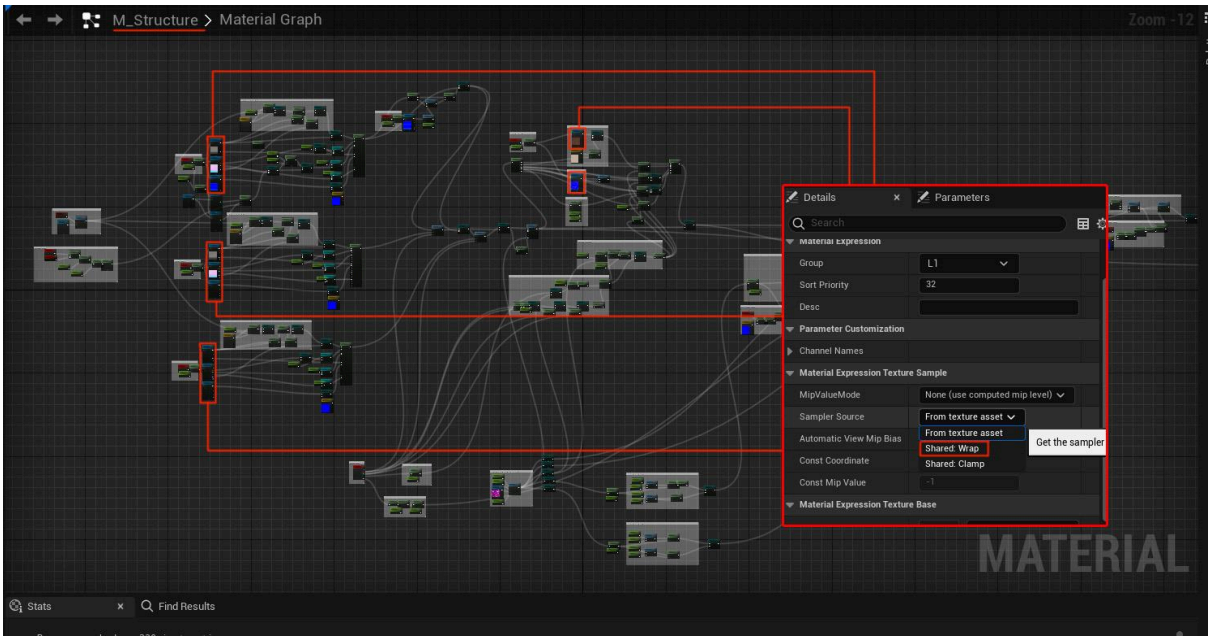
Q. I'm getting too many samplers error in some of the materials. How to solve that?

A. Something changed in the Unreal Engine 5.2 how it handles texture samplers so it will add two extra samplers per material. If you have certain project settings enabled, it will cause some materials to go above the supported 16 samplers. This sounds like a bug in the engine but here are a few ways to fix that right now.

Disable "**Allow Static Lighting**" option in the "*Project Settings > Rendering*". That will free up a few more samplers in every material.



Open the **M_Structure** master material (the heaviest material in this pack). Select all the highlighted texture sample nodes and then change "*Sampler Source*" to "***Shared:Wrap***". After that, recompile that material.



Q. Will this pack work with the rest of your post-apocalyptic packs?

B. Yes it will. Most of the materials are using similar concepts and the visual style is consistent. There are some shared assets between these products like cars, some props and debris. Different visual themes can be also easily changed from desert to more “forest” type because sand/moss are using separate material layers. That way you can mix these assets better with each other.

Q. Is this pack already containing Post Apocalyptic Vehicles and Post Apocalyptic Rifle packs?

C. Yes, those assets are included with this pack with some tweaks to fit with the theme so you don't need to buy those packs. Future vehicle updates will also appear in this pack too.

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. Also it's not optimal to use static lighting with large environments like these.

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 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. Vehicles are not working and I'm getting blueprint errors in BP_ExamplePlayer.

A. That looks like an issue with the Chaos Vehicle system. By default, UE5 disables Chaos Vehicle Plugin and that will then cause those issues. You can try to enable that Chaos vehicle plugin in the Plugin window (Edit > Plugins > Vehicles, enable ChaosVehiclesPlugin) and restart the editor again.

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. ExampleMap is empty. What can I do to solve that?

A. ExampleMap is a huge level and that's why it's using the Unreal World Partition system. You need to load cells in order to see the level. Open **World Partition** tab and then left-click + hold and mark areas in the minimap and right-click to choose "Load Region from Selection".

Q. How large is the example map?

A. Example map is quite large, around 6,2 square kilometers and contains different areas like city center, suburbs, nature areas etc..

Q. Does this pack contain any audio?

A. Unfortunately this pack does not contain any audio files.

Q. I saw a player character in showcase videos. Is that provided with this product?

A. You will be getting that player blueprint. It's based on the third person player blueprint with various additions and tweaks like a first player camera mode, object picking, driving and such. You won't be getting that exact player mesh because that is using a MetaHuman character. You can always create your own or decide to use the basic Unreal mannequin asset.

Q. Why can't I climb up the stairs? It feels like something is blocking it.

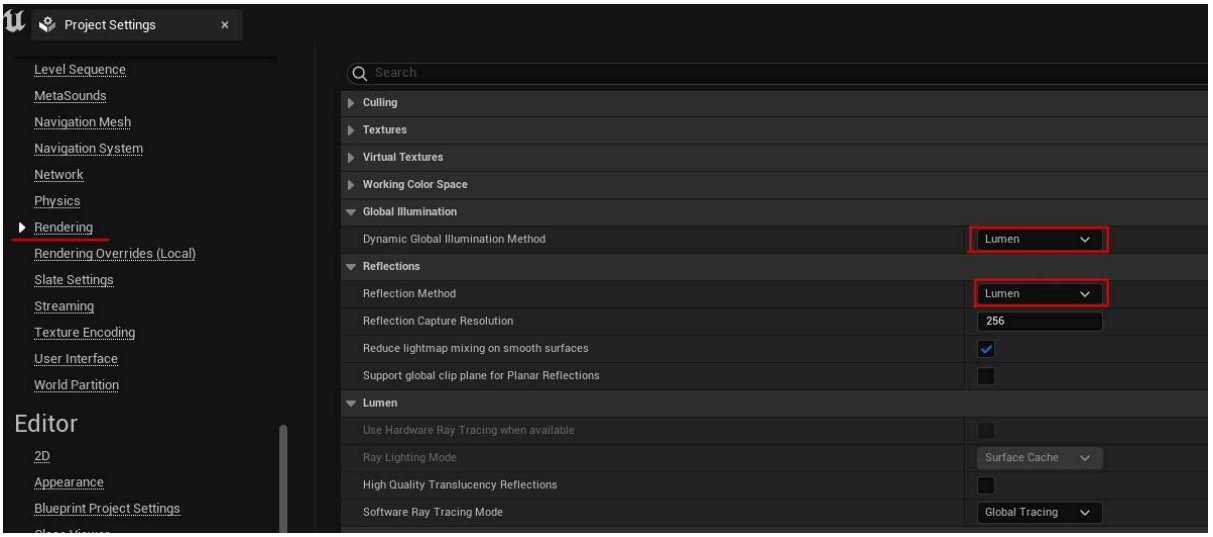
A. That sounds like your character is not allowed to climb those steep angles. You can change that by opening your character blueprint and then select the "Character Movement" component. Then there you can find a parameter called "Walkable Floor Angle" that you can change to something like "*60*".

Required Settings

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

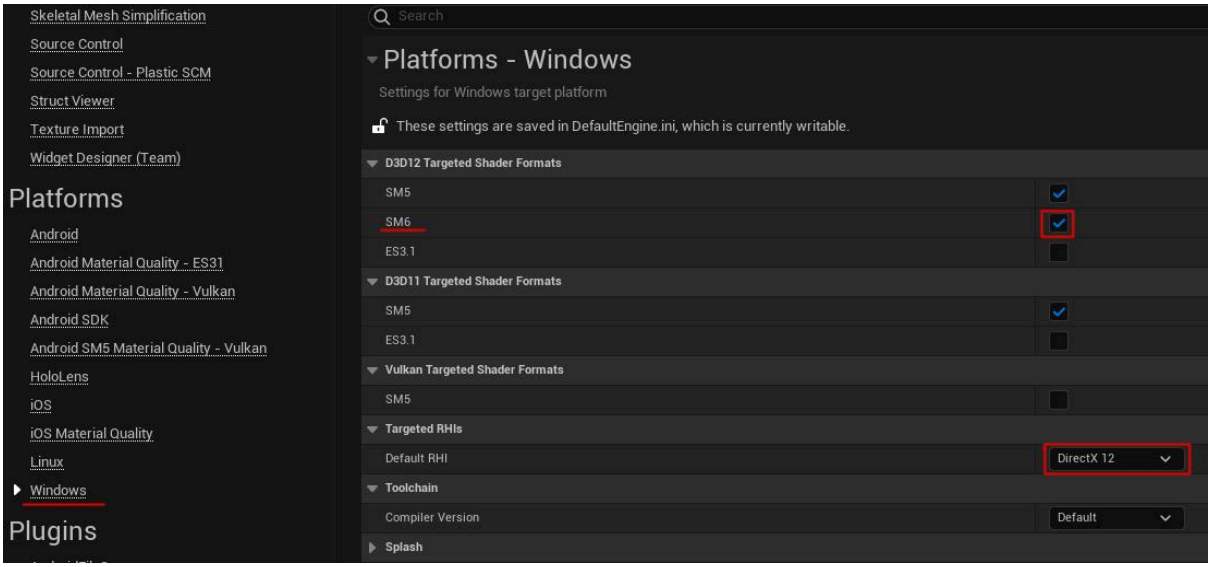
Lumen

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.



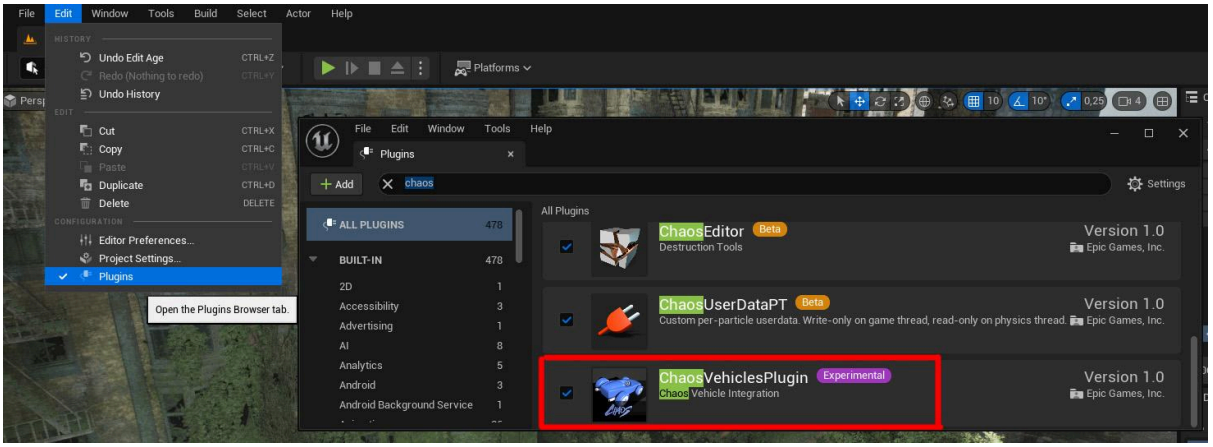
Nanite

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.



Plugins

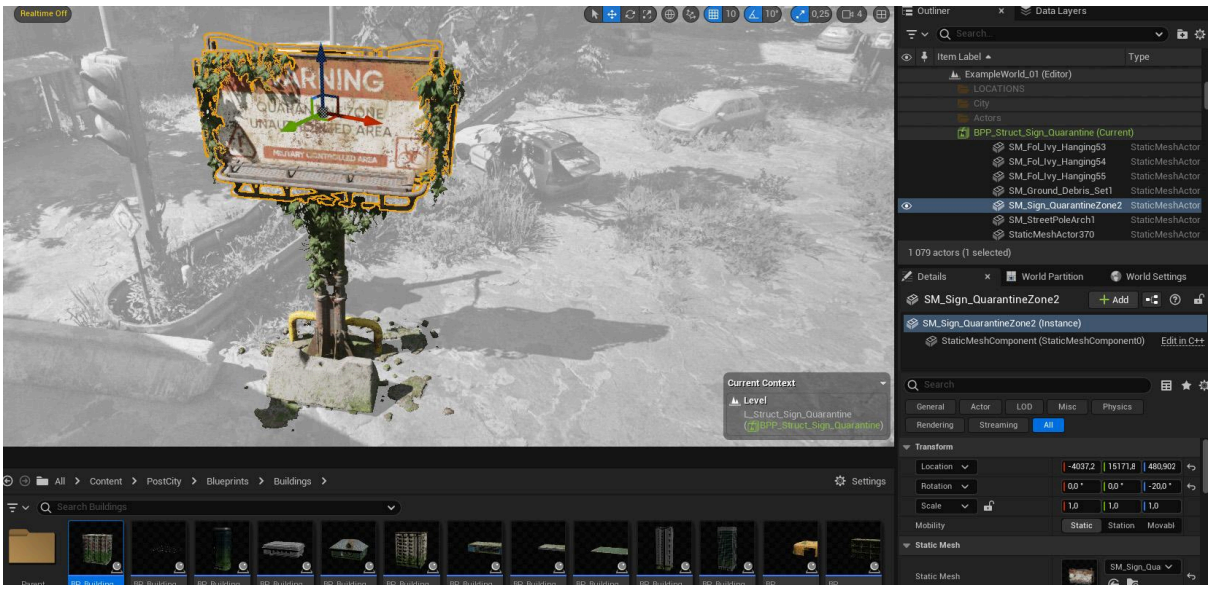
By default, UE5 disables Chaos Vehicle Plugin and that will then cause those issues. You can try to enable that Chaos vehicle plugin in the Plugin window (Edit > Plugins > Vehicles, enable ChaosVehiclesPlugin) and restart the editor again.



Packed Level Actors

Overview

This pack comes with pre-made sets of various assets that form larger structures. Rock formations, buildings, prop sets etc.. This allows to easily add different sets into levels and it will also handle mesh instancing under the hood. You can read more about that here.

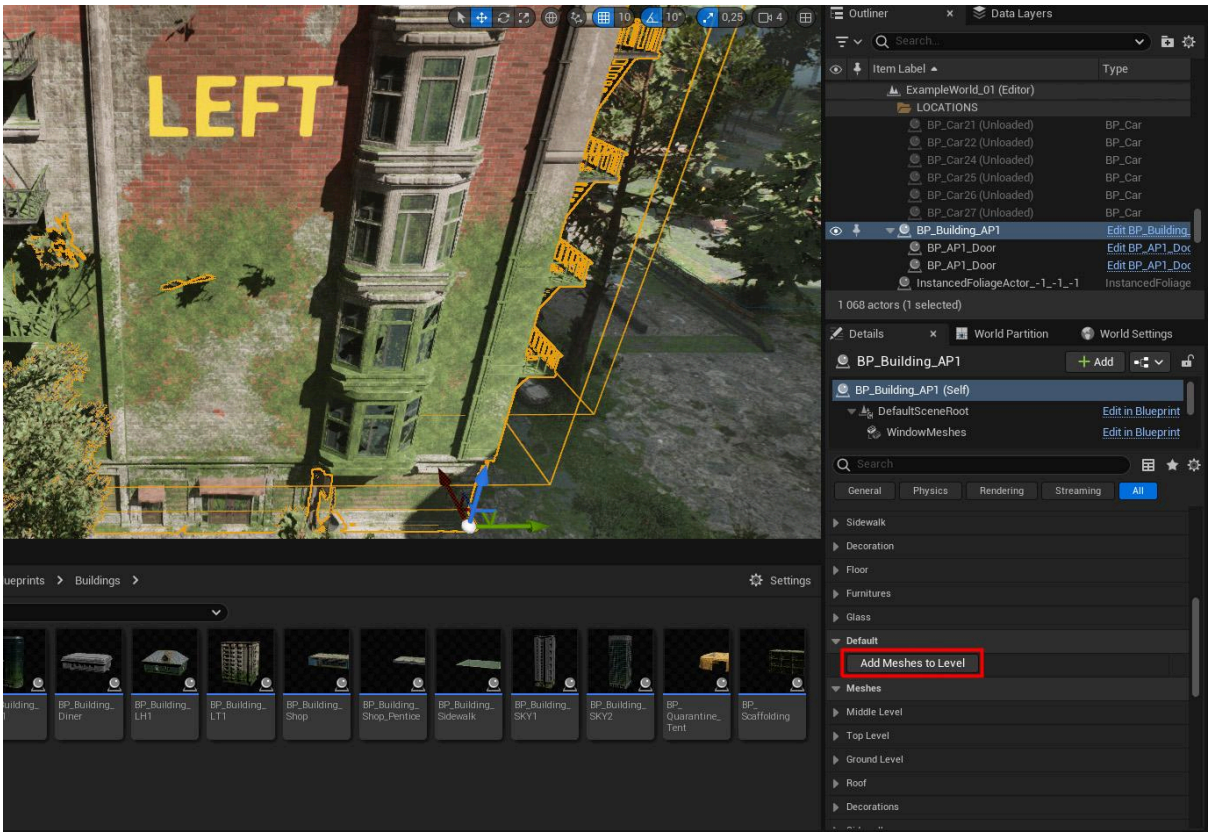


In a nutshell the system works like this. You can select different meshes and then convert them into a single actor by right clicking and choosing "*Level > Created Packed Level Actor*". Then you can specify where to save that level instance and after that the actual packed actor. It will try to find similar meshes and then turn them into instanced static mesh components to save draw calls.

You can always edit these actors afterwards if needed by selecting them and pressing the "Edit" button in the Details panel. When you are happy with the changes, you can press the "Commit Changes" button and it will update the level instance that will then update that actor as well.

How to make your own?

This pack comes with a set of different blueprints that helps to generate, place and scatter all sorts of things. The drawback is that these blueprints will run their logic in the construction script that might end up being a problem if you have lots of them in your level. That's why the optimal approach would be to use them at first and lay down things like buildings and such. Basically help you skip the time consuming, manual work. Then turn those blueprints into basic static mesh components and convert those into a packed level actor.
Most of the blueprints contain an editor event called "Add Meshes to level" that you can find in the *Details* tab. That way you will get the best of both worlds.



Why use Packed Level Actor instead of building Blueprints?

Blueprint systems in this pack are built to be as efficient as possible but the reality is that this will always introduce some overhead. Even though most functions will output instanced meshes and try to limit draw calls as low as possible, there are still some fundamental limitations when it comes to blueprints. Optimal situations would be to calculate everything offline, store that into a disk and this way reduce the runtime cost but unfortunately that would be out of scope for this product.

In order to understand these limitations, it would be a good idea to get an overall picture of how the system works. Building system is running a function in the construction script that will construct each wall, roof, floors and such. All of these will then use a few generic functions that will generate HISMs to batch draw calls. Then there are more "sub systems"

built on top that can be toggled on/off to save performance. In total there are over two hundred variables that can affect what the system will build.

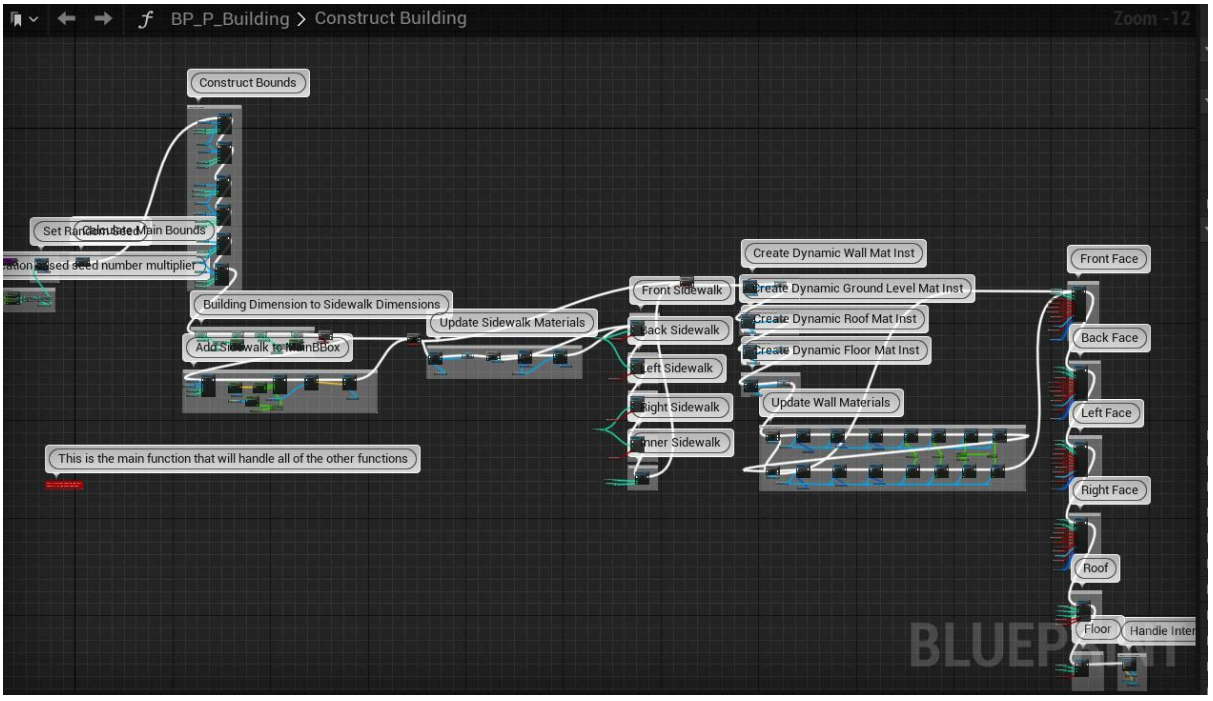
There is a basic minimum cost for a building if it only constructs walls and roof but the worst case scenario would be that it will use all of the features where it will scatter things on walls like ivy, construct interiors and add stairs, furniture etc.. The issue is that the more of these buildings there are, the more Unreal needs to run these functions that will run other functions and all of this will add up in the total loop count. Worst case scenario would be that where Unreal crashes because there are way too many loops. Another issue is also the amount of dynamic material instances that makes it harder to batch buildings in macro level.

That's why a library of premade packed level actors would be more optimal and would work better with Nanite to limit draw calls and such. That will strip all the extra calculations and will use basic ISM components. This combined with the new World Partition system allows to optimize a lot and allows to add more hand placed details like ivy and such.

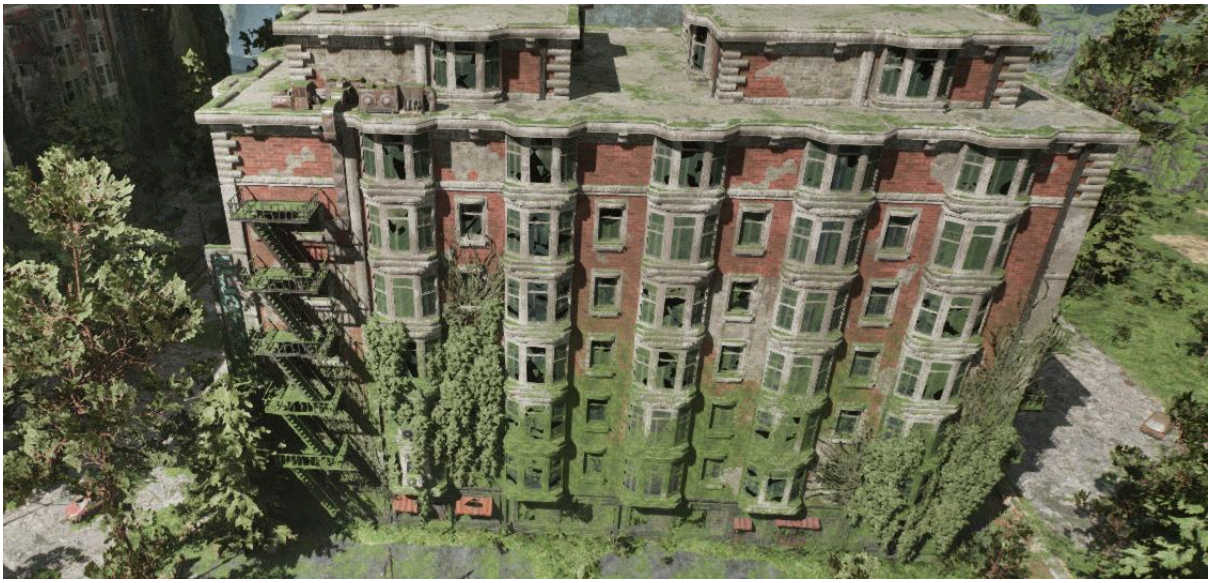
Building System

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.

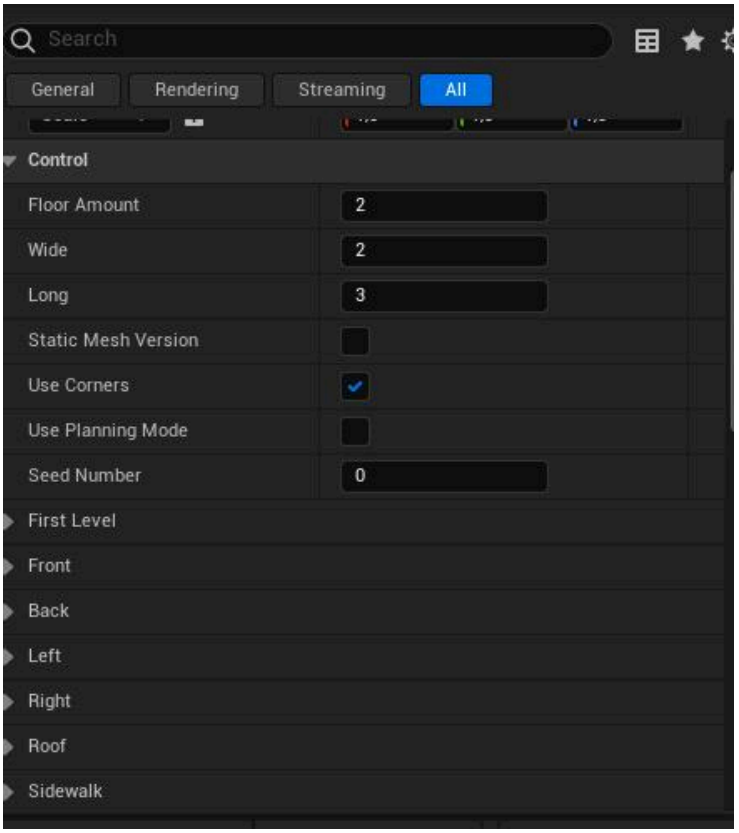


Main logic is in the **BP_P_Building** parent class. Every building is then inheriting that logic from this class. Some common variables are public so you can tweak them in the level and create more variations between buildings. Others are private that you need to change inside that child blueprint. 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

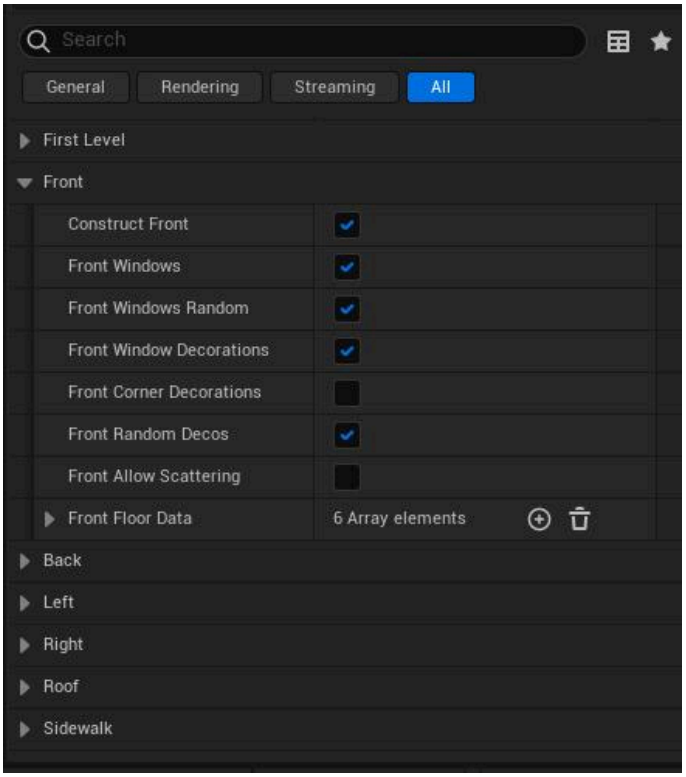
System is divided into different parts. First stage is the planning stage. In this part the system will calculate building dimensions and generate bounds that are later used for the actual building. You can find a variable called “Use Planning Mode” under “Control”. Turning that on will make it faster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

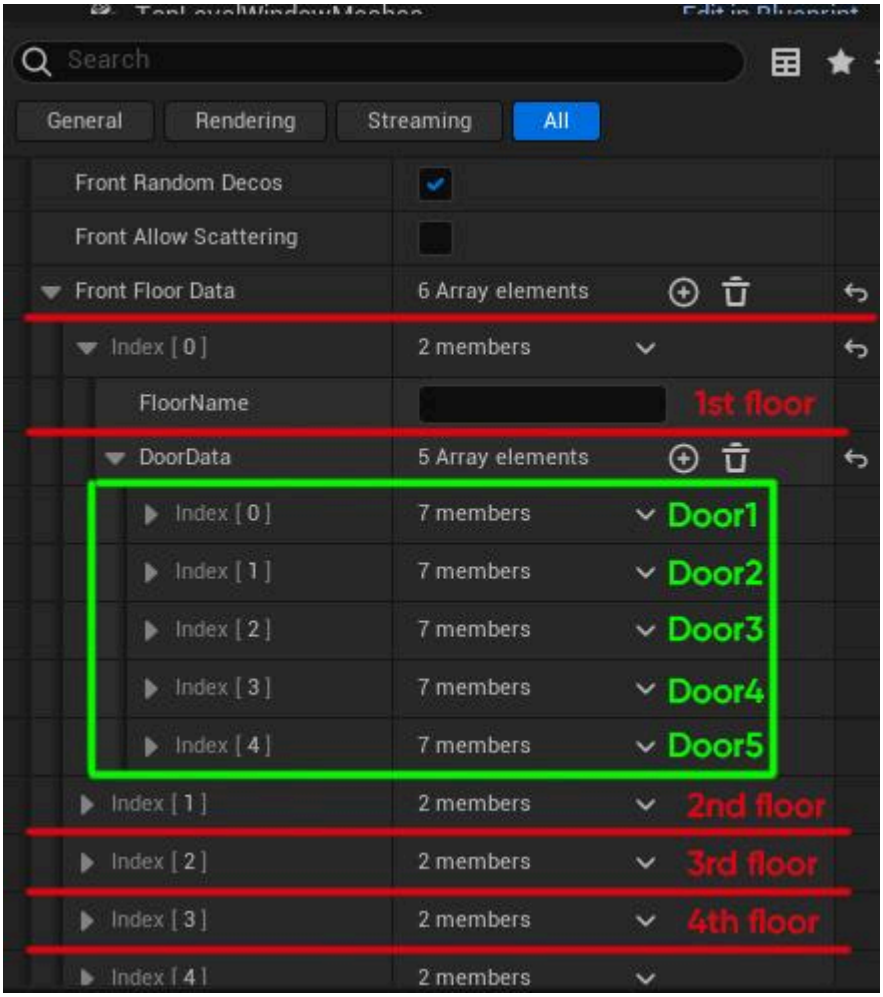


There is also a separate control for the “First Level”. You can enable/disable this part and control where the door would appear or just disable doors totally. “Door Offset” controls the door blueprint position that is different for every building due to different building dimensions. You can also spawn 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.

You can also add doors on every building 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”.

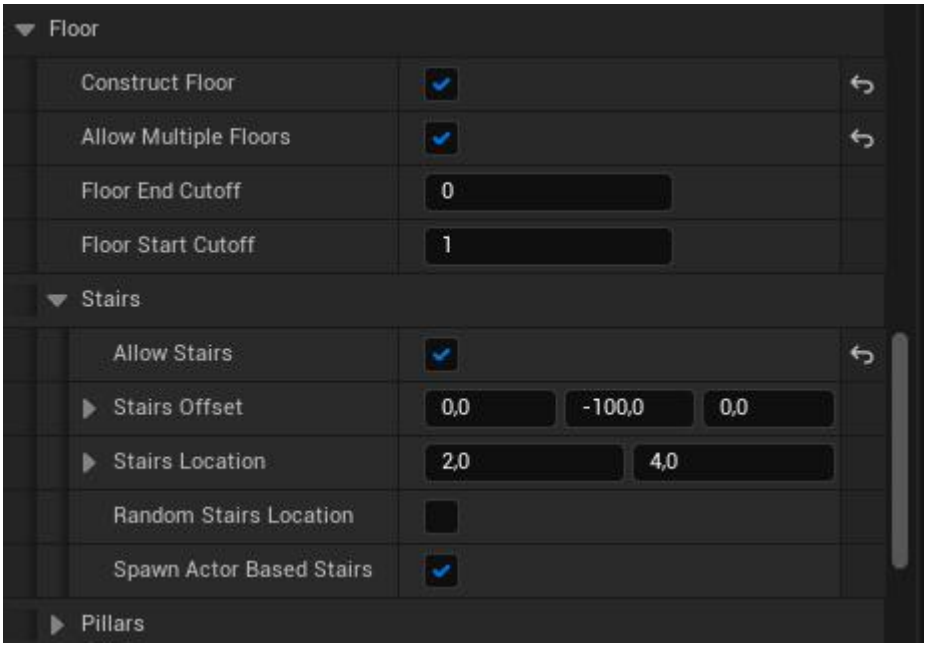
If you wish you can also construct sidewalks. This will construct sidewalks using the building dimensions and then you can specify multipliers that will shrink or grow that system.

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

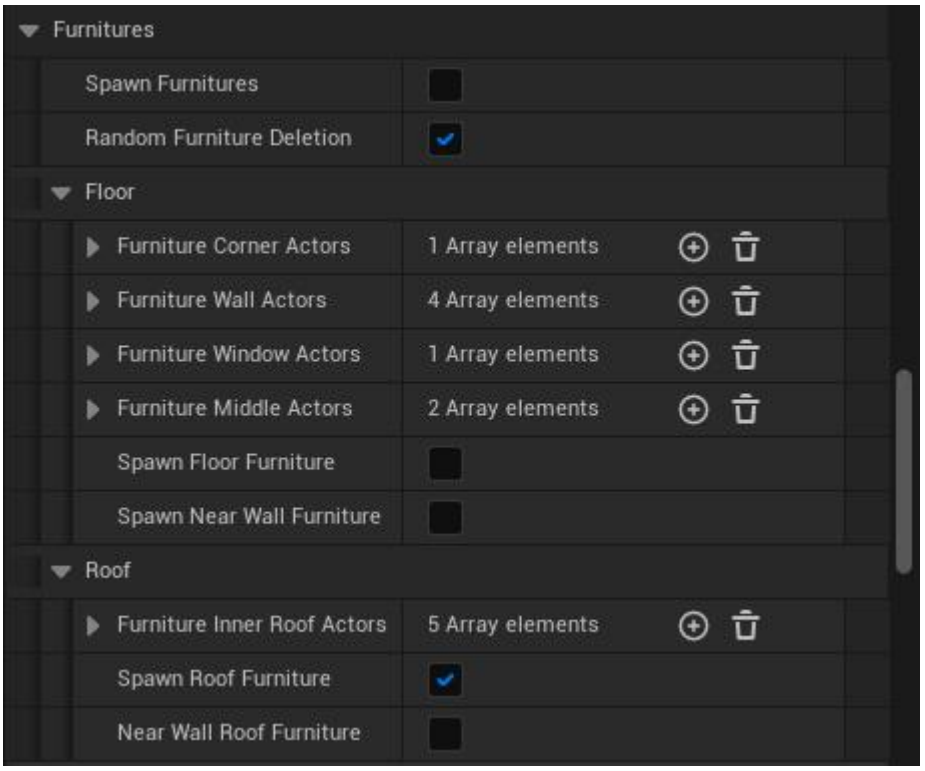
Amount" number. You can also control floor start and end cutoff values that are useful to specify how many floors the system will skip from start or end. "Floor Height" will determine the distance from the previous floor to the next one and usually you want to keep this same with the wall height that 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.

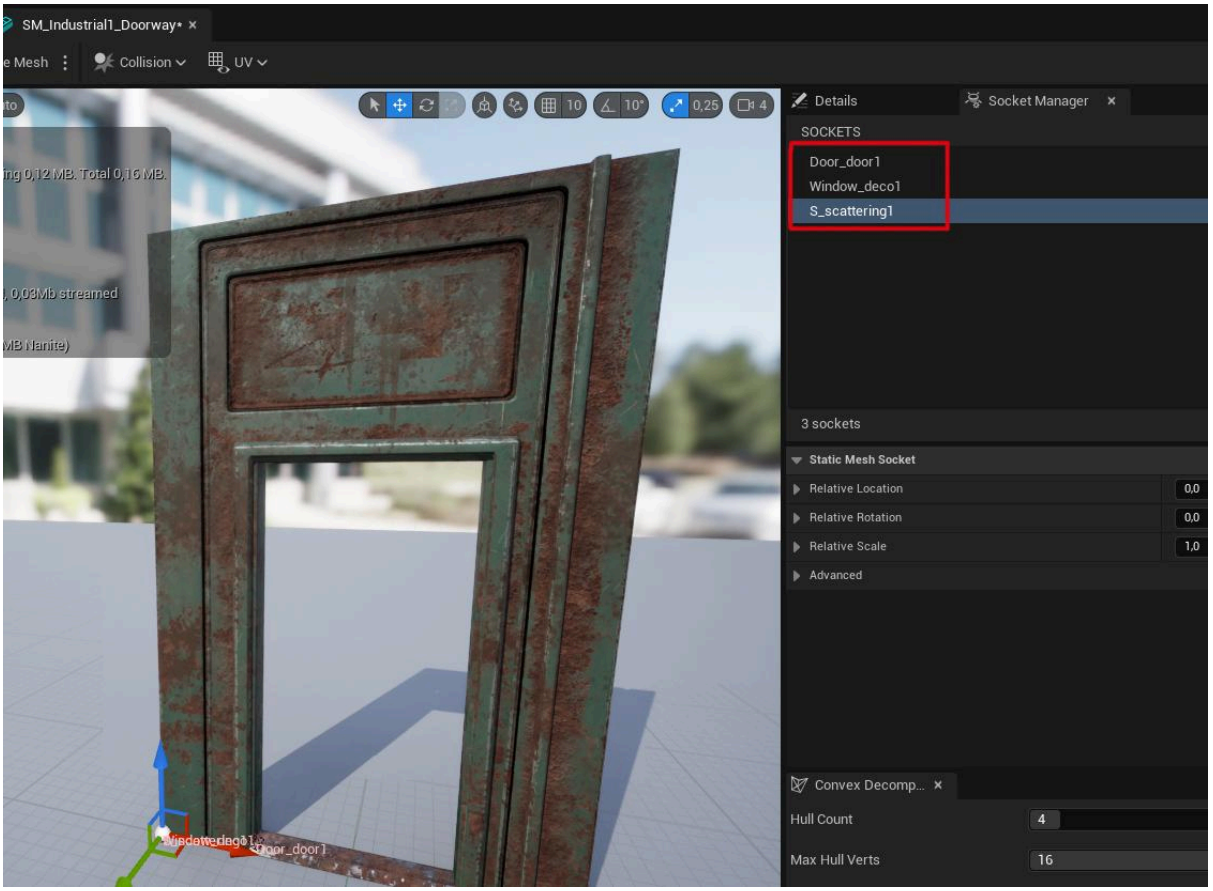
You can choose to use random rotations, apply different scales for scattered meshes, change culling distances, scatter density and even collision settings. System will look for meshes from the “Scattered Meshes” array and randomly choose different ones and put

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.

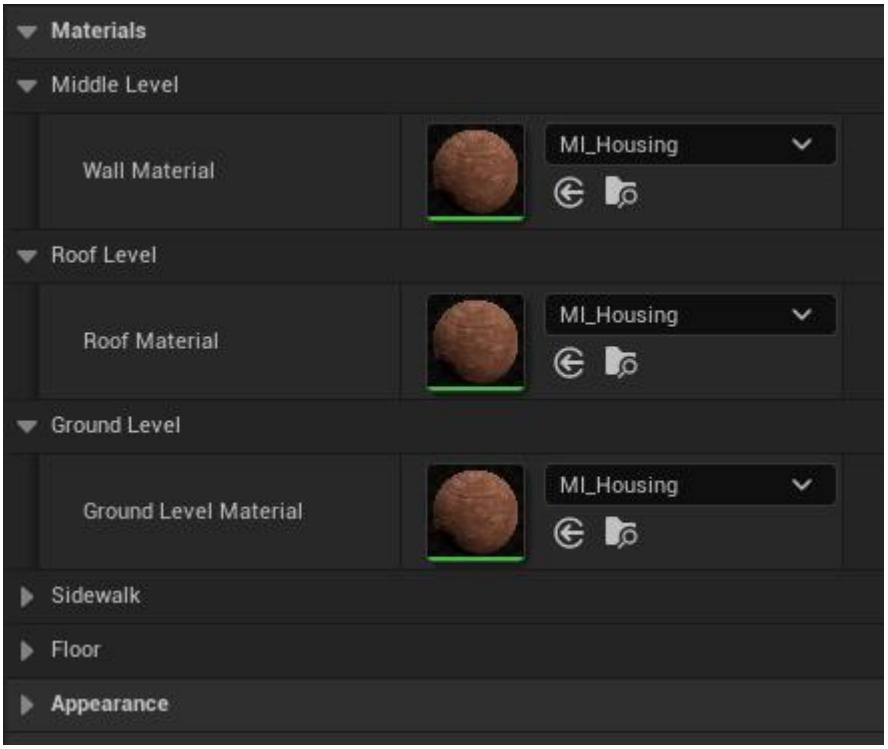


Socket names can also matter because one mesh can have many sockets that each spawn different things. Basic socket scattering (*Scattering/Sockets*) requires that the mesh socket name is **S_SocketName**. System will look for that **S_** prefix and you can have as many sockets as you like.

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**.

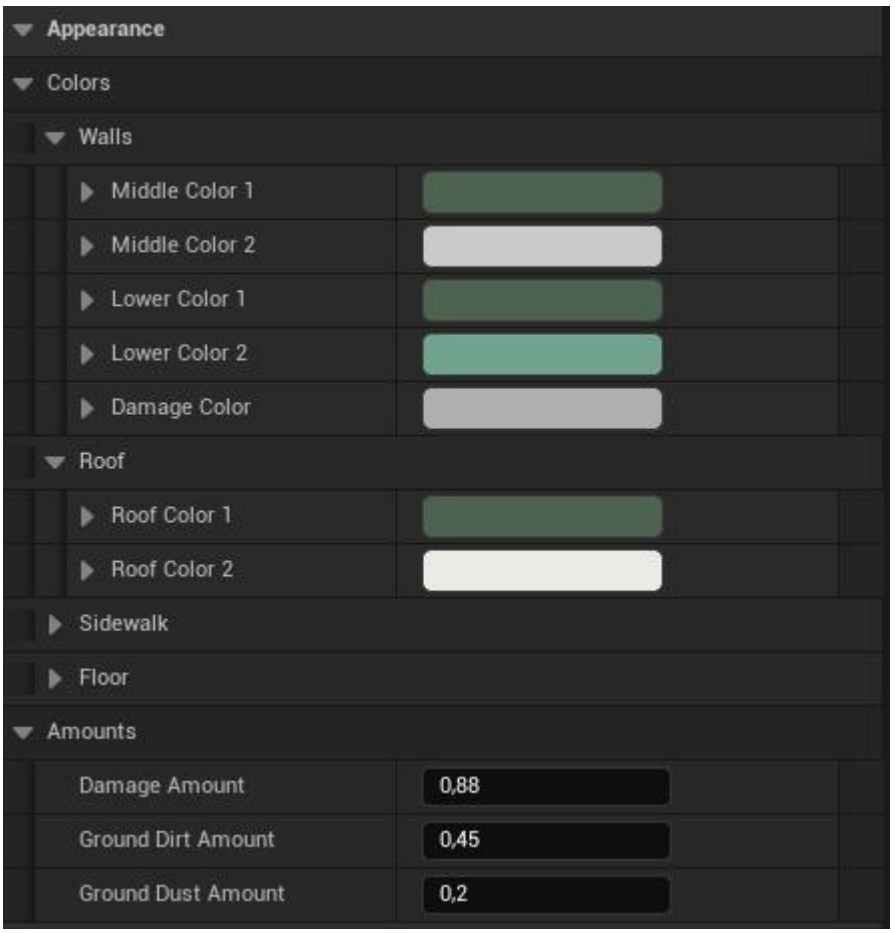
Building Appearance

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



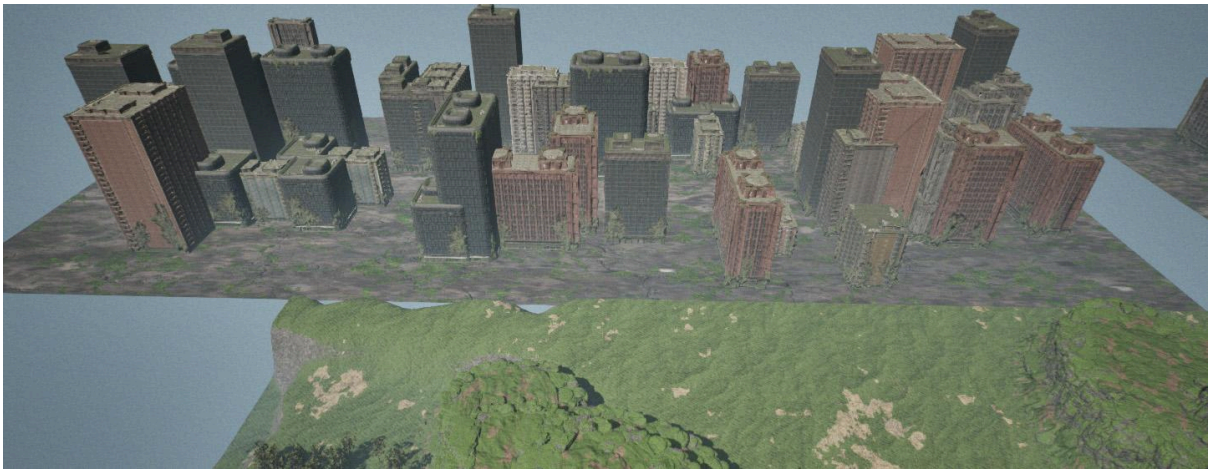
Remember if you are changing building materials, make sure those materials contain the right parameter names in order to change colors and damage values with this system.

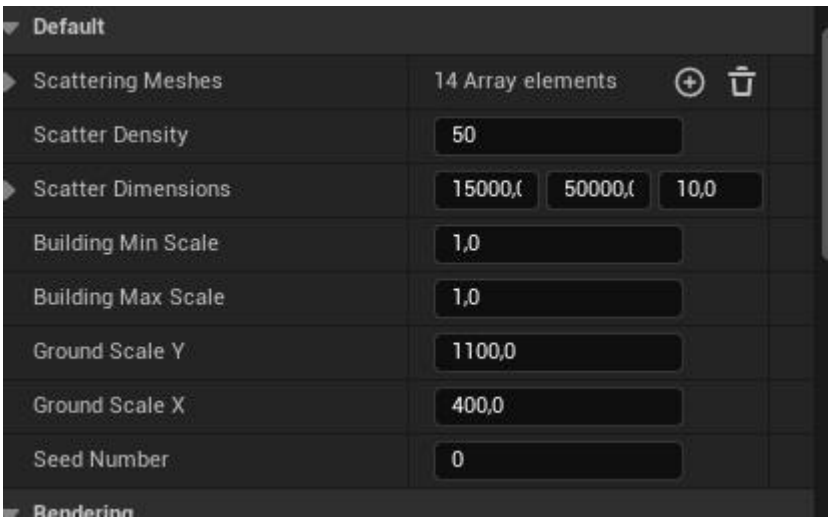
Changing building colors is very easy. You can do that in the "Appearance" section and specify colors for walls (Middle Level), first level (Ground Level), roof and sidewalks. Under "Amounts" you can find settings to control the amount of damage and dirt buildings have.



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.

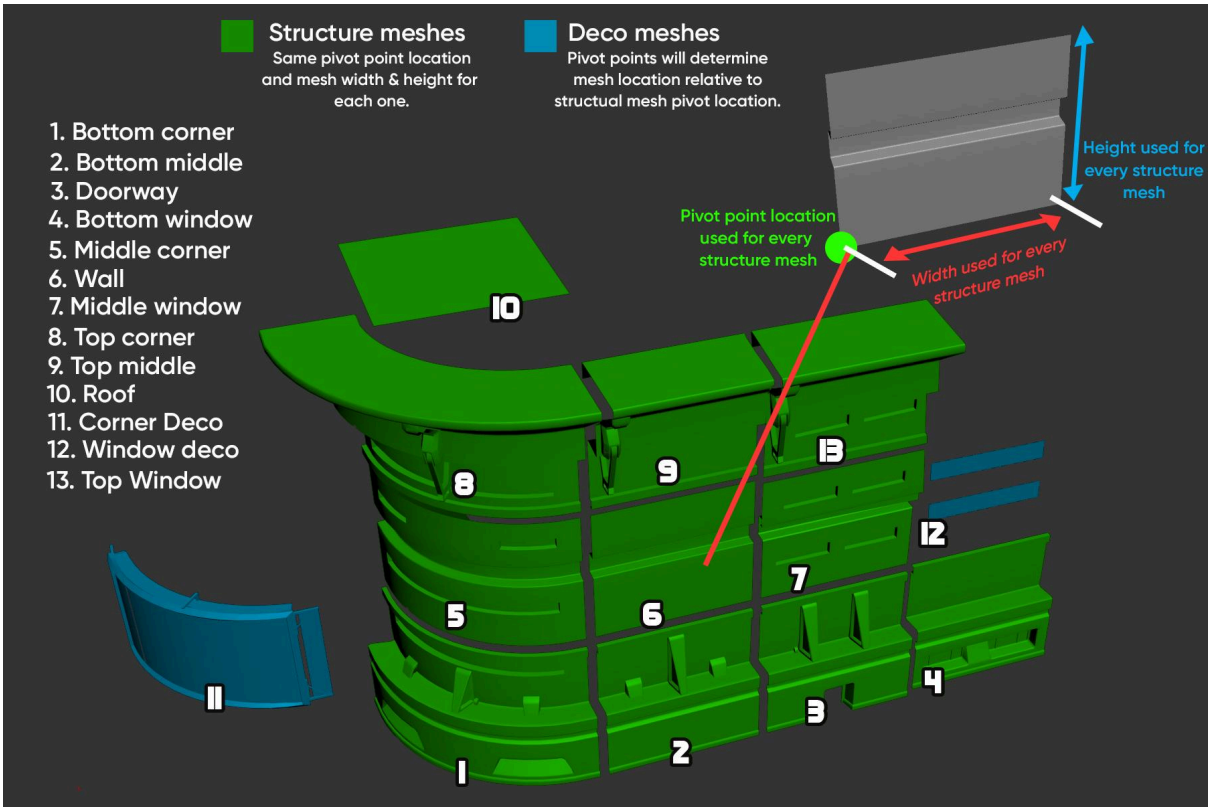
Custom Building Meshes

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.

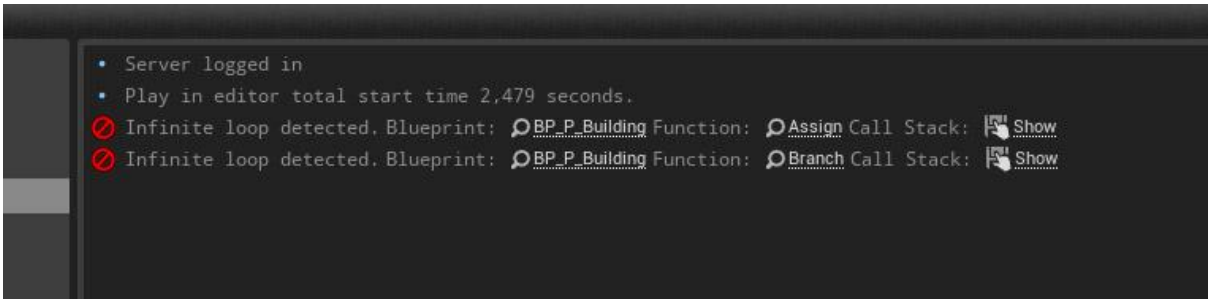
Roof and corner meshes should have the same X and Y scales. For example if the wall mesh is 4 units wide the X and Y values should also be 4 for corner and roof meshes. This way there would be no overlapping or gaps between meshes. Roof mesh pivot point Z location will determine where the roof is placed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

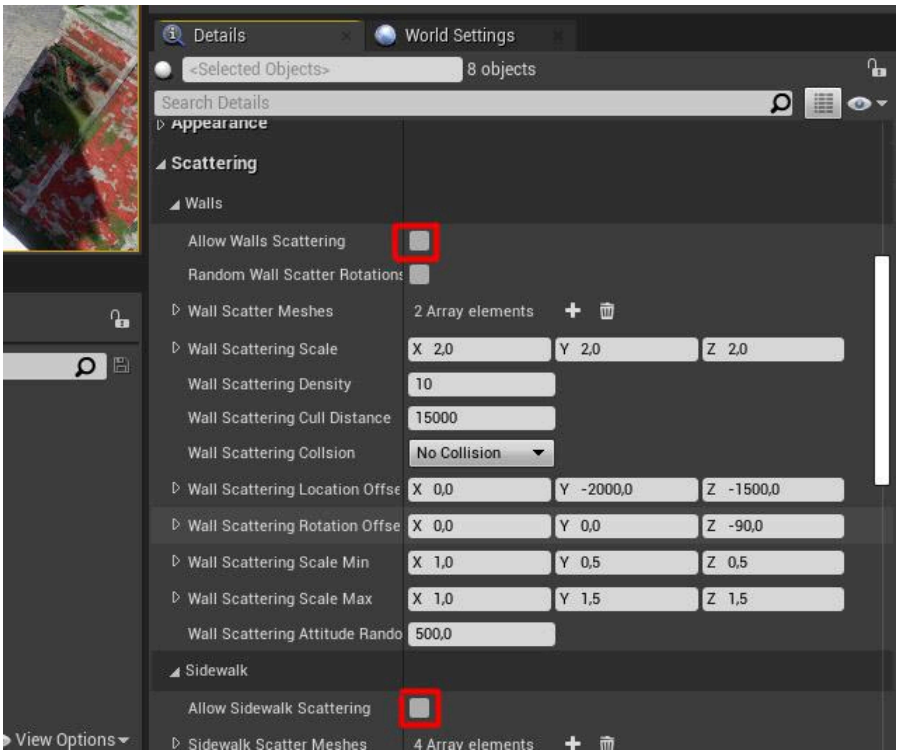
Level Load Error



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.*



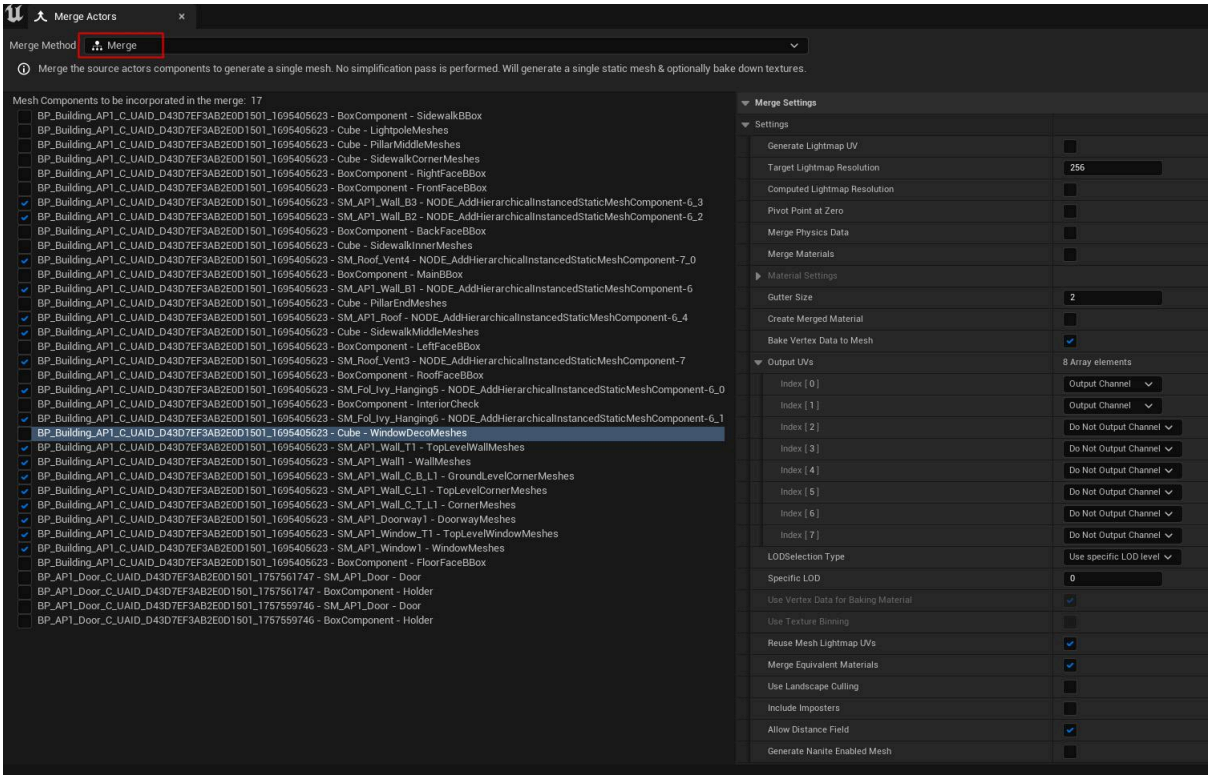
3. *Merge building actors down to simple static meshes. This way Unreal Engine doesn't have to load building actors or run any loops. There are few things to keep in mind.*

    1. *Unreal's merge tool is not handling dynamic material instances right and in some cases, it can produce wrong results. That's why you need to manually assign correct wall materials for the merged mesh.*

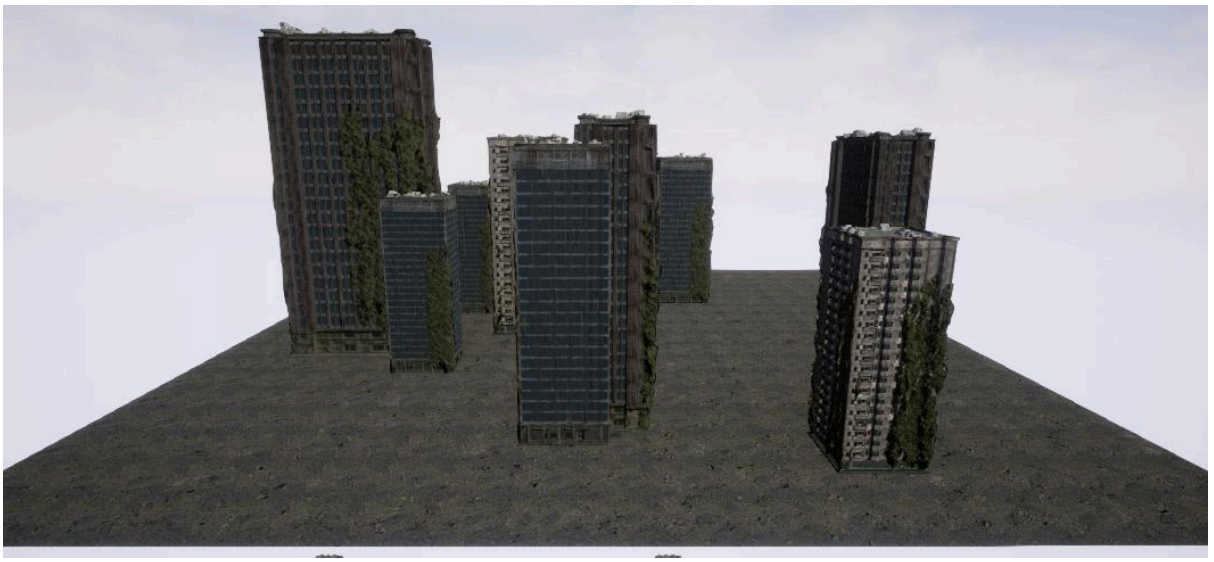
    2. *Some building materials are only working right with instanced static meshes so merged buildings might not look exactly the same. You can see this in windows and such where "PerInstanceRandom" node is used. Another thing that will not work is custom building colors, damage etc. that are set in the building blueprint but are not going to work with the merged building version.*

    3. *In Unreal Engine 5, It would be advised to use the "Merge", "Simplify" or "Approximate" modes. Select only components you wish to merge because*

any extra components can cause issues. You can do this by unchecking components from the components list. Recommended settings are in the picture below.



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. **BP_EnvironmentController** is then changing light values based on the time of day system’s clock values.

If you want to manually change light colors then you can either disable the “Controller Active” boolean variable or even open the **BP_EnvironmentController** and tweak values there. You can also replace this controller with something else so it’s safe to remove it totally. Just make sure that if you still want to change materials wetness values then you need to edit corresponding material collection parameters.

Spline System

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.

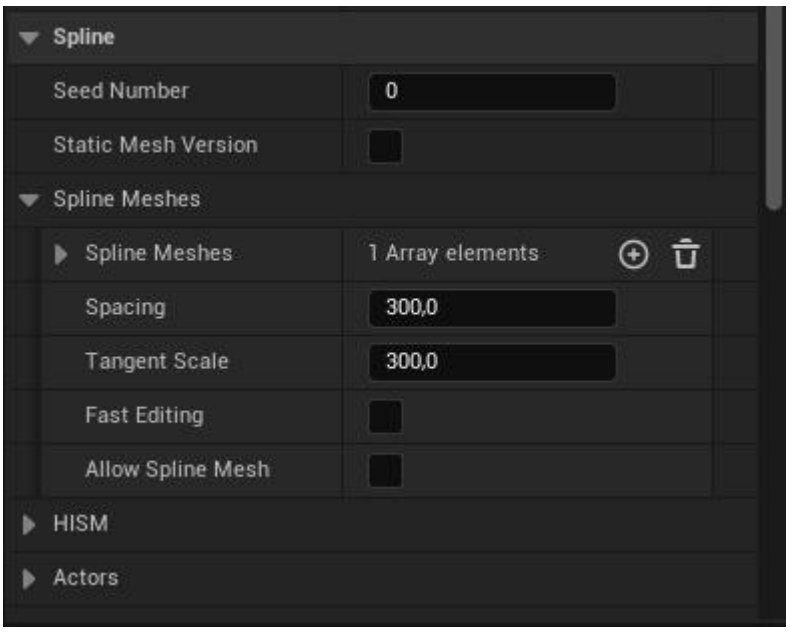
Spline Meshes

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 squeeze 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

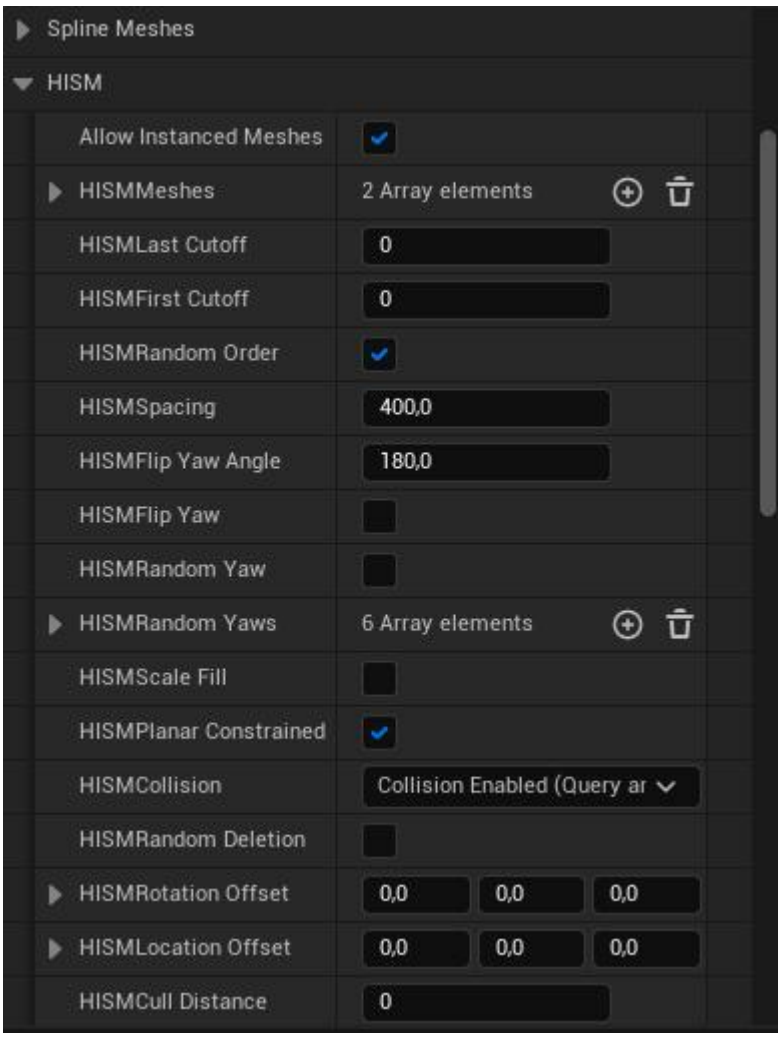
Using instanced meshes is a more optimized way because then we can batch meshes better. Unfortunately HISMs are not supporting mesh bending in the same way that spline meshes do. However there are lots of cases where we don´t really need to bend meshes so HISMs are a more optimal choice then.

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.

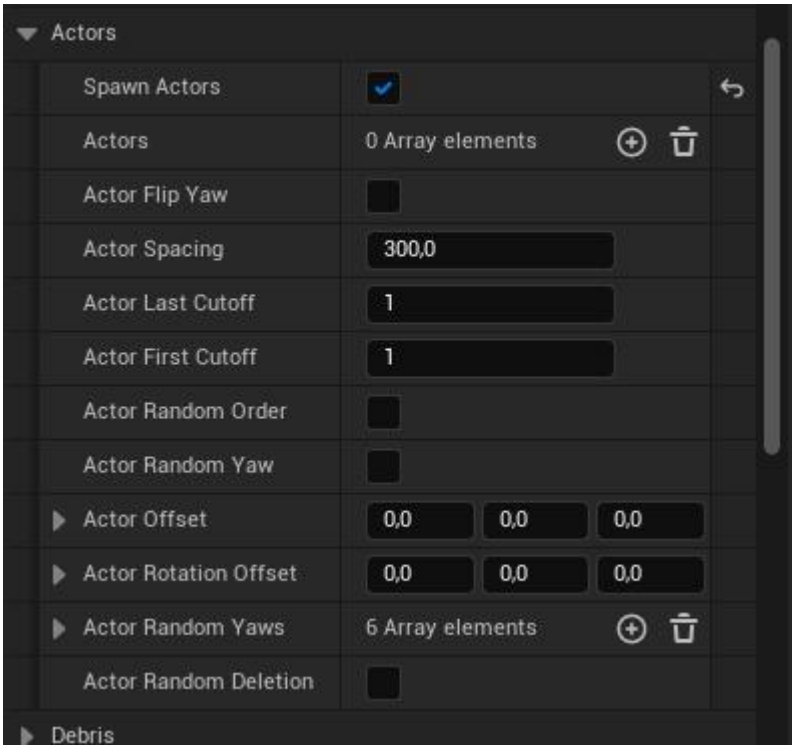
“HISMScale Fill” will help to fill gaps in cases where the mesh length will not match with the spline length. When this setting is true the system will scale the first mesh to fill this gap.

“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

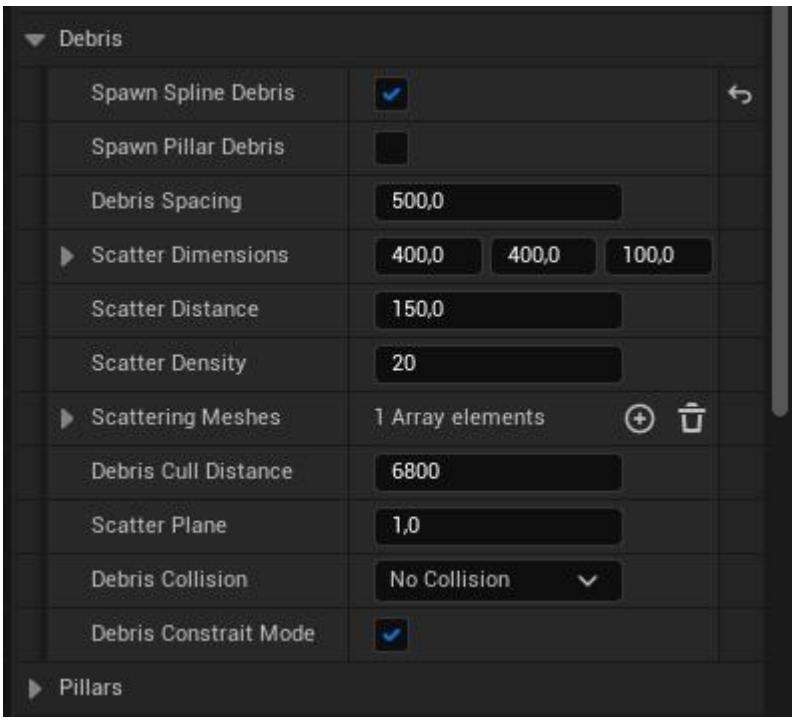
You can also spawn actors along the spline. These settings are almost identical with HISM settings. There is an array where to specify what actors to spawn and actor spacing value will control the density of how often these actors are spawned based on the spline length.



Debris Spawning

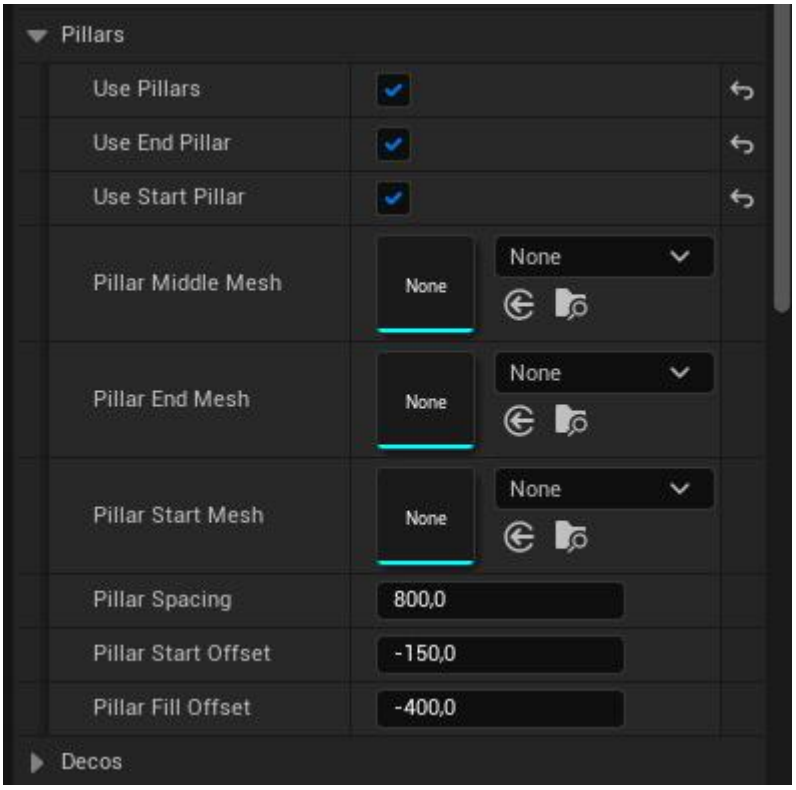
Spawning debris is also supported with this system. “Scattering Meshes” array will contain meshes that are going to be scattered. “Debris Spacing” value is controlling how often the scattering will happen along the spline and “Scatter Density” will control the amount of meshes that are spawned when that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

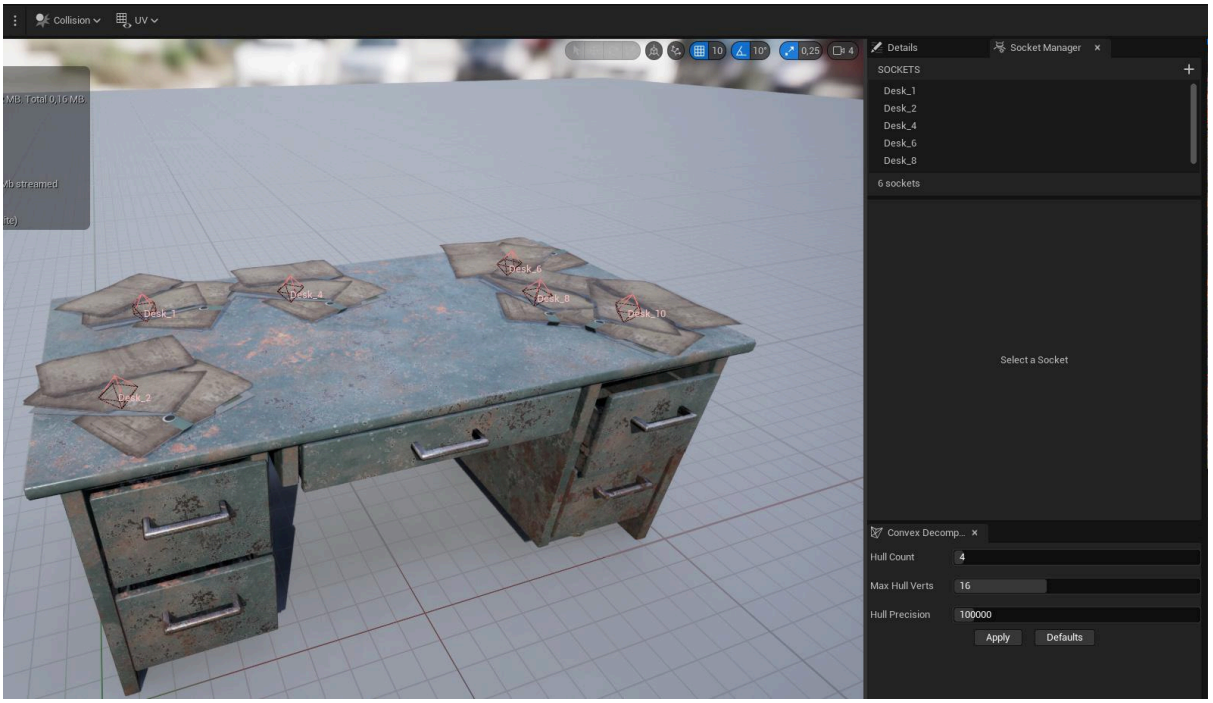
If you need to add supporting pillars you can turn the “Use Pillars” option on. This system will trace lines from locations that are taken from the spline based on the “Pillar Spacing” value. You also have extra control to specify whether or not to use end and start pillar meshes.



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. It doesn’t matter what those sockets are called. This is perfect for scattering things like ivy, hanging foliage and such.



Scattering 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.

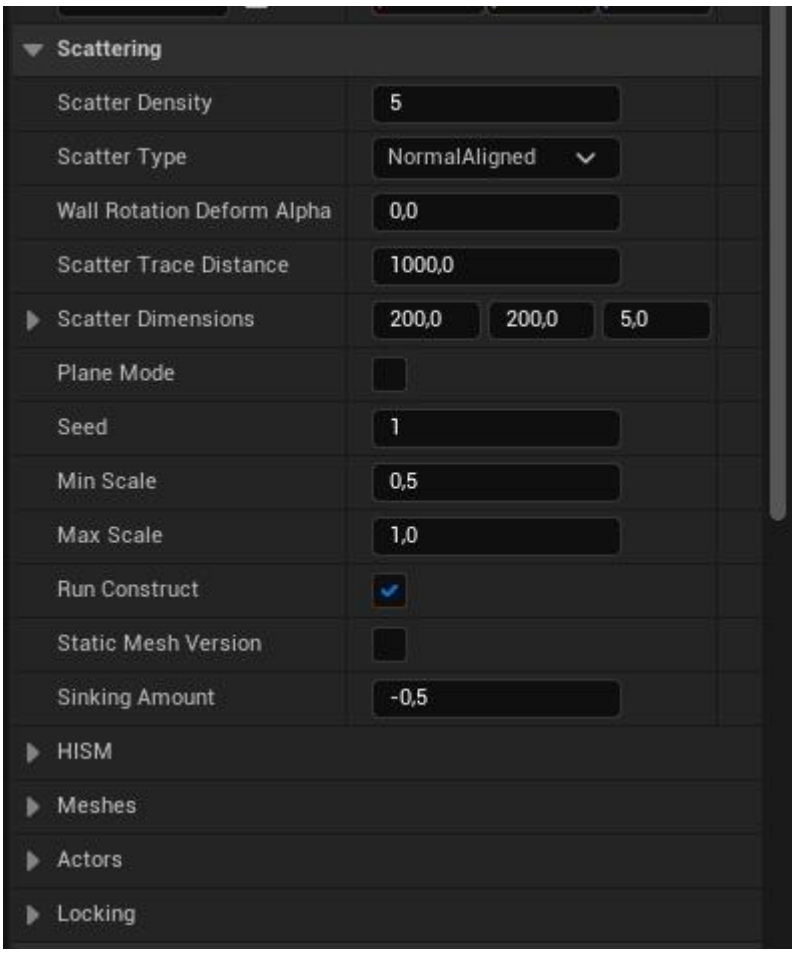
“Culling Distance” is controlling when to cull mesh instances. Smaller value means that instances will be culled faster and a value 0 means instances will not be culled unless culling volumes or systems similar to that culls it.

“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.

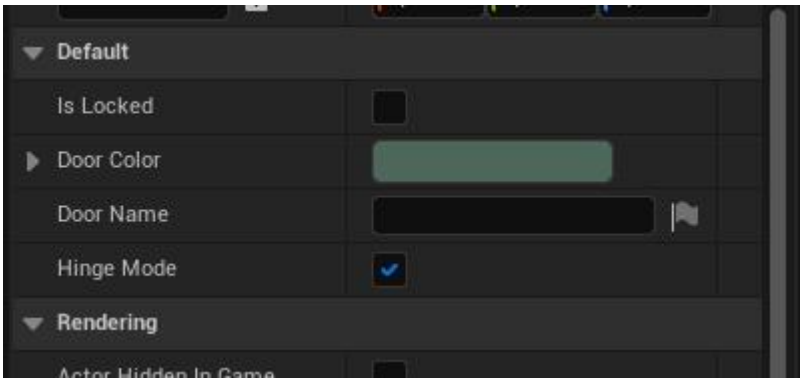


You can also use this blueprint as a child actor. This way you can for example scatter cans from vending machine blueprints etc.

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.



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.

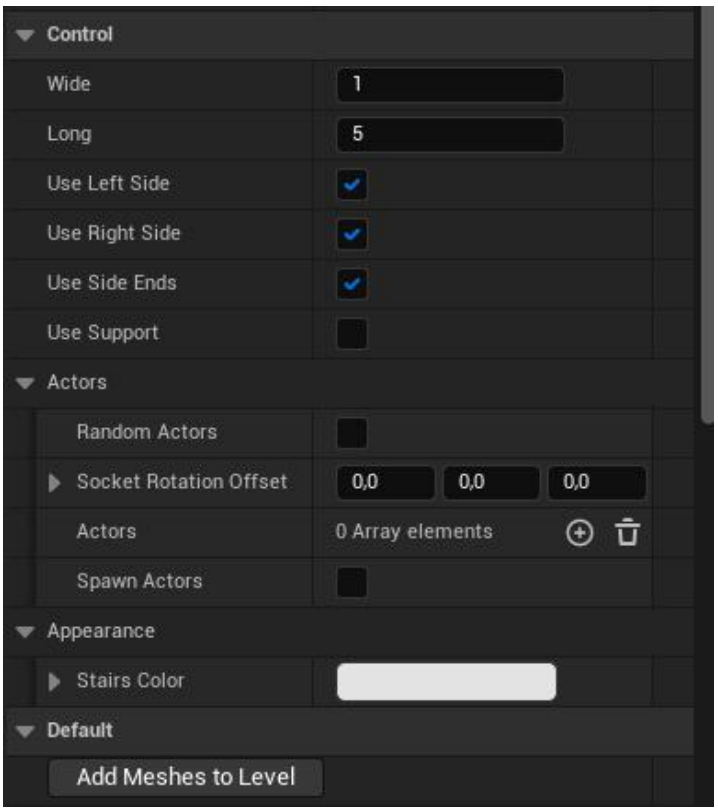
“Door Value” is controlling how much the door is open/closed by default at start. When the player is opening it, the system will continue door rotation from that point onward. Value 0 = door is fully closed, 1 = door is fully open. This system is using a timeline node to control the rotation alpha and speed.

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/Structure/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.

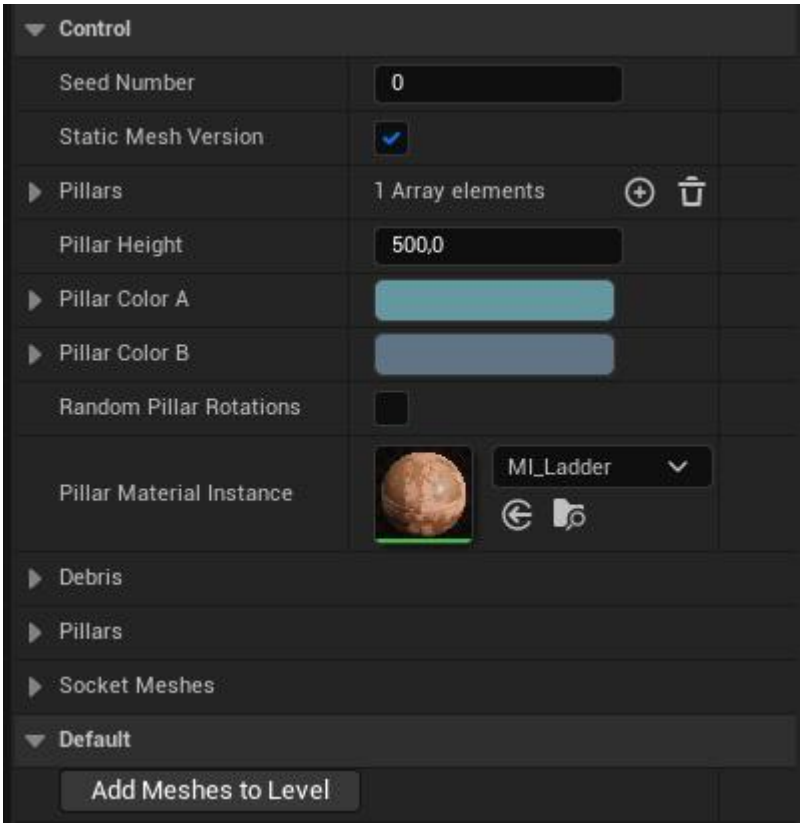
Appearance

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.

Pipes

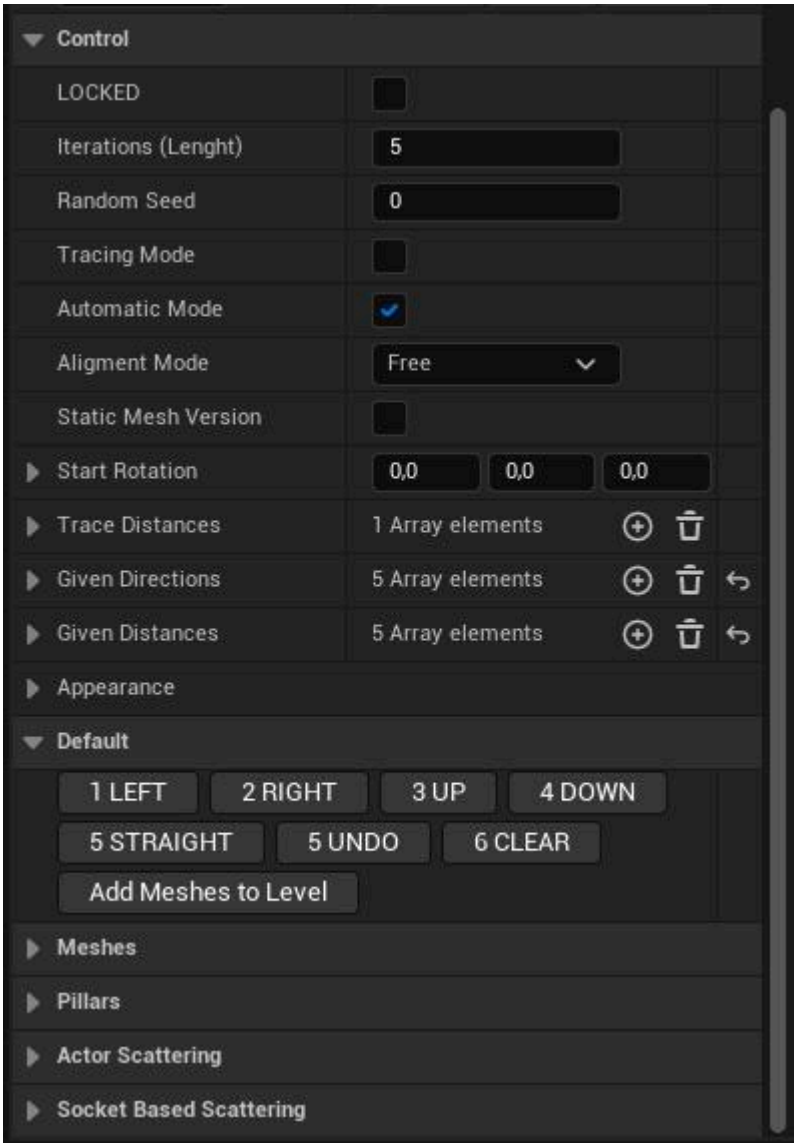
Main Logic

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*".

Foliage

This pack comes with different foliage assets that range from trees to small grass assets. Trees are modeled in a way that allows the use of PivotPainter based material effects for wind and such. You can read more about that in the documentation [Pivot Painter Tool 2.0 in Unreal Engine | Unreal Engine 5.1 Documentation]

It is advised to use a foliage painter tool for placing tree assets. This way Unreal can batch draw calls in a more efficient way. Alternatively you can also set up procedural foliage spawners to speed up this process and you can read more about that system in here Procedural Foliage Tool in Unreal Engine | Unreal Engine 5.0 Documentation

Grass assets are modeled in a way that it will reduce overdrawn as much as possible and to work better with the Nanite system.

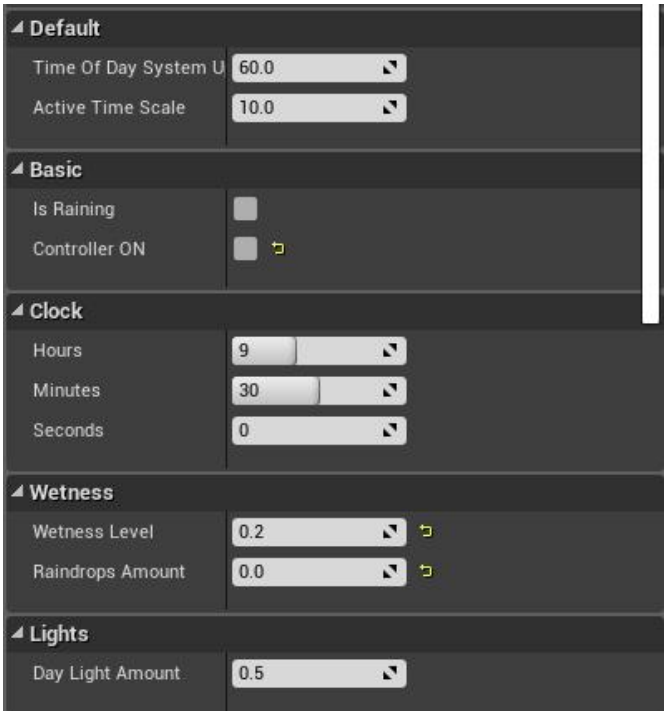
Grass and smaller foliage is using the landscape grass system. You can tweak individual LandscapeGrassTypes (*Materials -> Nature -> Landscape*) to have better control for scale, density and lighting settings. You can find more info here.

Environment Controller

Overview

This pack also comes with a basic environment controller. It supports a fully dynamic day and night cycle and an option to use rainy weather or control different wetness values (Wetness Level, Raindrops Amount).

You can specify hours, minutes and seconds and the system will update values based on that. Main logic is using curves that you can tweak for your needs and the system also works in the editor. If you wish to use it at runtime you can turn the “Runtime Time Of Day” variable to true. This way it will update the system at runtime based on the “System Update Delay”. “Time Speed” variable is then controlling how fast the time will go. Increasing that will speed up the time of day transition.



"Rainy Amount" will control the rain. This will change wetness values and will also add raining water particles to the active camera. This variable is a float and the range is from 0 to 1 where 0 = sunny/dry, 1 = rainy/wet, 0,5 = something between.

In order for this system to work, you need to give it the correct references in your level. "*Level References*" tab will contain these for fog, sun, skylight, and sky. Simply press the down arrow and choose the actor in your level.

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.

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.

You can also enable Virtual Texturing that will help to optimize landscape material performance cost. That is also located in the Project Settings, under Rendering where you can find an option called "Enable virtual texture support".

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.

It's also a good idea to tweak overall scalability settings. You can read more about that here.

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 to apply it globally.

Depending on your needs, you might be using ray tracing. This is something that will increase the performance footprint dramatically and disabling that can help a lot with the overall performance.

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

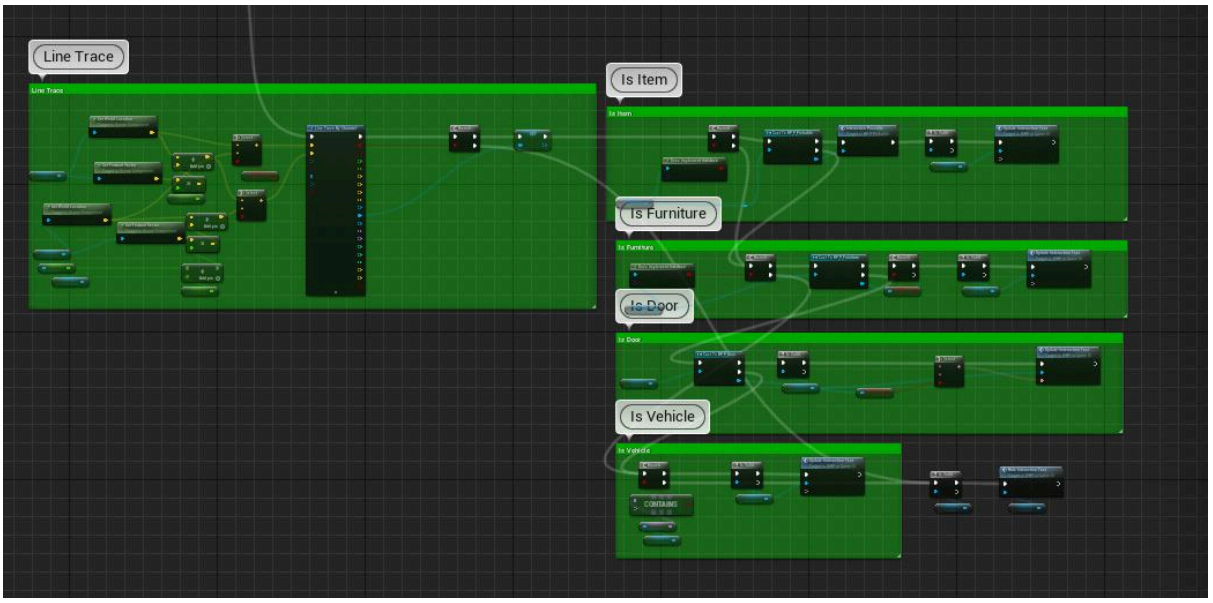
This pack also comes with an example player blueprint. It is based on the basic Unreal Engine third person pawn with some changes like an option to switch between first and third person views.

Example interactions

Player is able to interact with things like doors, furniture, pickable items and vehicles. This way it's possible to pick up actors that are inherited from the "**BP_P_Pickable**" class. It's also possible to interact with furniture to turn lights on/off for example. This is possible if the actor is inherited from the "**BP_P_Furniture**" class. If the actor is inherited from the "**BP_P_Door**" class, the player is able to open/close them. "**BP_P_VehiclePawn**" class allows you to enter vehicles and drive them.

All of the following interactions are using the same concept. Player blueprint (**BP_ExamplePlayer**) is using line traces to find actors. After that the system will start to identify what kind of actor that is. This is done by using the "*Does Implement Interface*" node.

If that returns a true value, the system will show that in player UI. When that happens, player is able to run the interaction logic by pressing the interaction button (Default "*E*").



Inputs

Input Settings

This pack is using the new Enhanced Input system so you don't no longer have to manually set up key bindings in the Project Settings. Example player and drivable cars are using their own Input Action and Input Mapping Context assets. That way you can easily edit different key bindings for different inputs.

Demo Inputs

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

Example Player

Movement:(**WASD**)
Sprinting (**Left Shift**)
Enter vehicle (**E**)
Flashlight (**F**)
Object picking (**E**)
Change camera mode (**V**)
Jumping (**Spacebar**)
Change to drone (**1**)
Hide UI: (**U**)

Vehicle

Throttle: (**W**)
Brake/Reverse: **S**)
Steering: (**AD**)
Handbrake: (**Spacebar**)
Headlights: (**F**)

Drone

Example Level

Thrust:(**Left Shift**)
Roll (**Q & E**)
Steering (**WASD**)
Change to player (**1**)

Restart level: (**R**)
Exit game: (**ESC**)
Toggle weather: (**2**)
Toggle time of day: (**3**)

Vehicles

Overview

This pack comes with vehicle assets that you can customize and use in various ways. Vehicle assets are split into different meshes so it's possible to open/close doors, rotate wheels etc.. This way it's more optimal for systems like Nanite and Lumen to handle vehicles.

Chassis mesh contains the vehicle body without wheels, doors, hood or trunk meshes. It will also use fully opaque materials so it can be used as a Nanite mesh. Vehicles are also containing a chassis transparent mesh that is usually used for windshields head and tail lights etc.. This mesh will use transparent materials so currently it can't support Nanite.

Door meshes work the same way so there will be a door mesh that will use fully opaque material so it can be turned into a Nanite mesh and then there are separate window meshes with transparent materials.

Wheel meshes are also going to be separate meshes so it's possible to specify each wheel separately. System will also handle steering wheel mesh and align wheels with that.

Drivable Vehicles

Vehicles are based on the Unreal Engine 5 Chaos Vehicle system and you can read more about that in the documentation page [Vehicles in Unreal Engine | Unreal Engine 5.1 Documentation]

In order to use this system and to prevent any errors, you should enable the Chaos vehicle plugin in the Plugin window (*Edit* > *Plugins* > *Vehicles*, enable ChaosVehiclesPlugin) and restart the editor again.

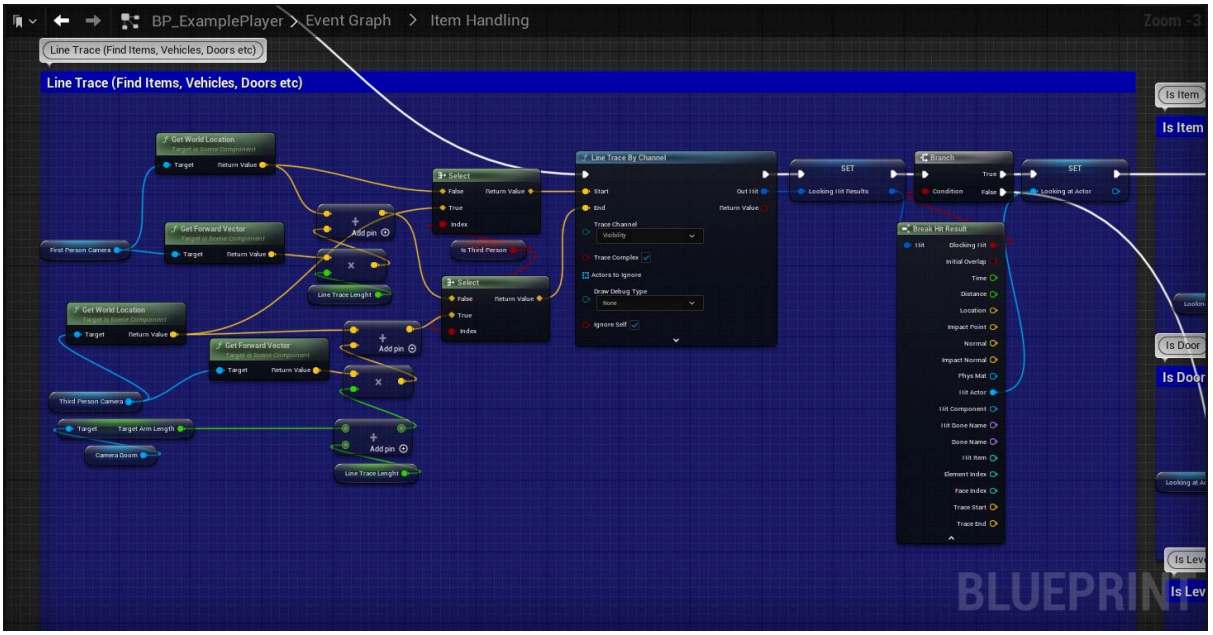
Driving System

Each drivable vehicle actor is inheriting its behavior from the parent class (**BP_P_VehiclePawn**). That actor contains logic to handle Enhanced Input system inputs, appearance details like color and age, vehicle lights etc.. If you need to make any changes to these systems, you should do them in this actor.

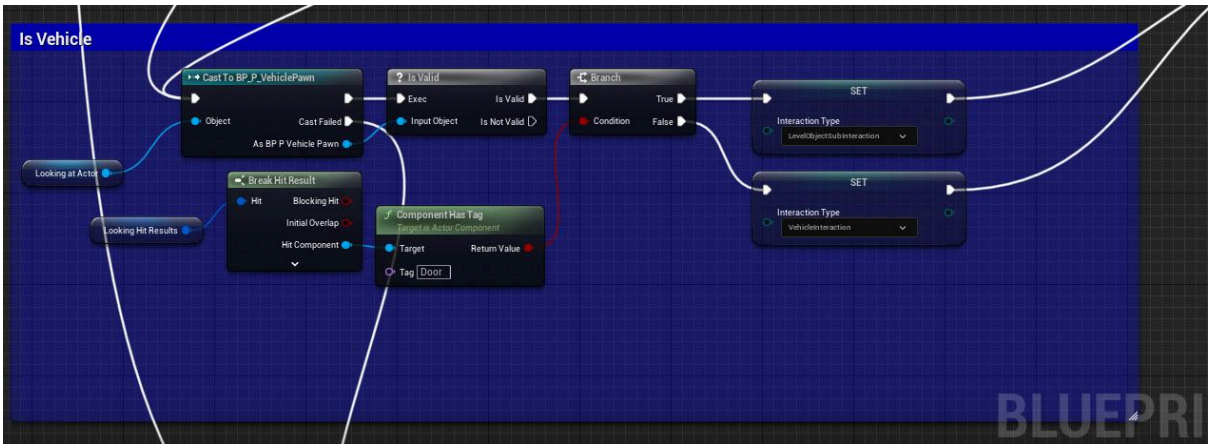
Easiest way to use these vehicles would be to use Unreal possession system and you can read more about that system in the documentation [Possessing Pawns in Unreal Engine | Unreal Engine 5.0 Documentation] I also made a [tutorial video] that shows how to do this.

Main idea is to get a correct reference to a specific vehicle actor in the level and then tell the Player Controller to pass the control to that actor. Usually this is done using a line trace.

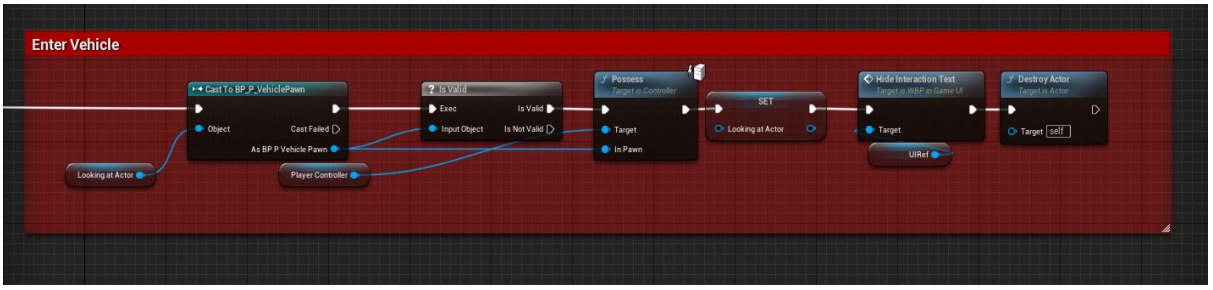
This pack comes with an example player character that contains this logic. It will use a line trace to trace against actors in a level. If the tracing hits something, it will store that in a temporary variable called LookingAtActor. After that it will start to do a few tests to figure out if that LookingAtActor is a correct vehicle or not.



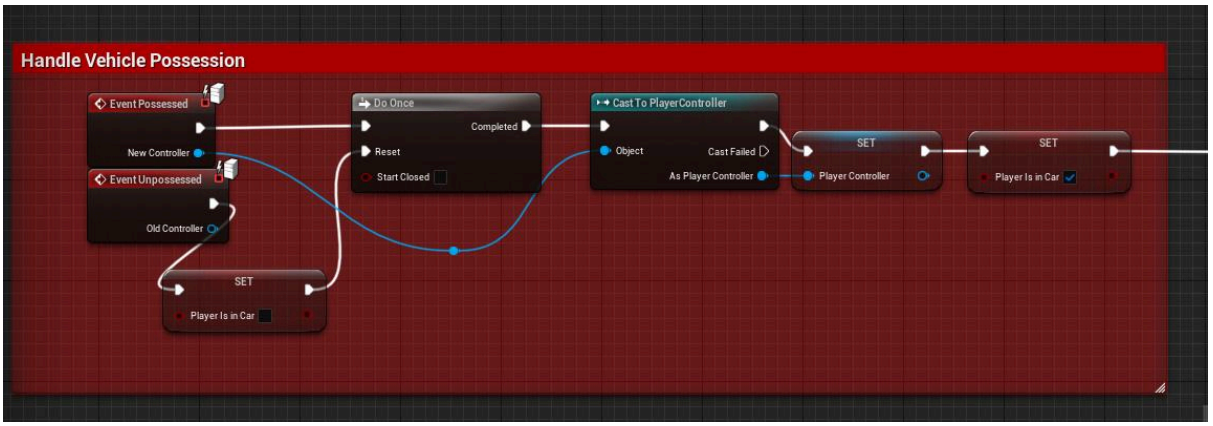
It will try to cast to “**BP_P_VehiclePawn**” and if that cast is successful, it will then proceed to the next part.



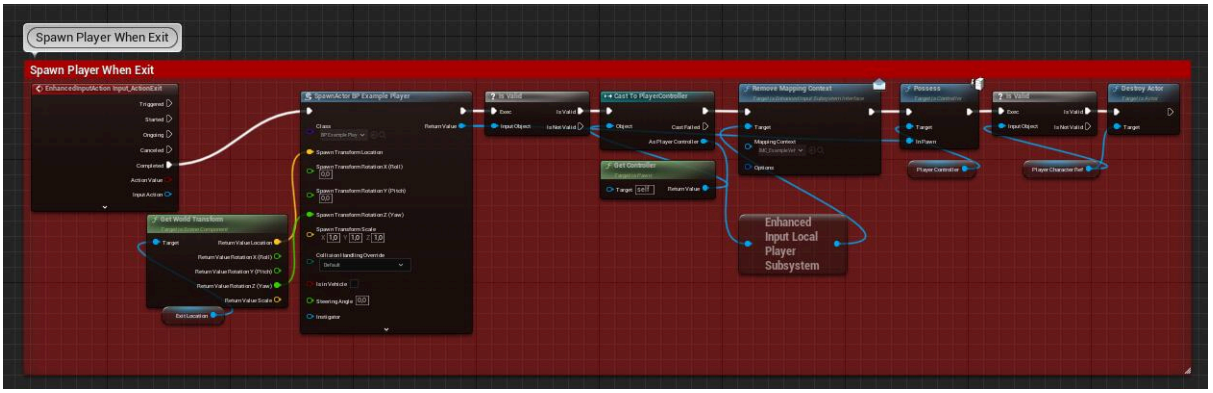
When the player presses the interaction button, it will then handle the possession to that vehicle.



In the car blueprint (**BP_P_VehiclePawn**) "Event Possessed" will now trigger because that is a pawn that we are controlling now. There we are setting a few variables that the player is in the car and setting the *PlayerController* variable.



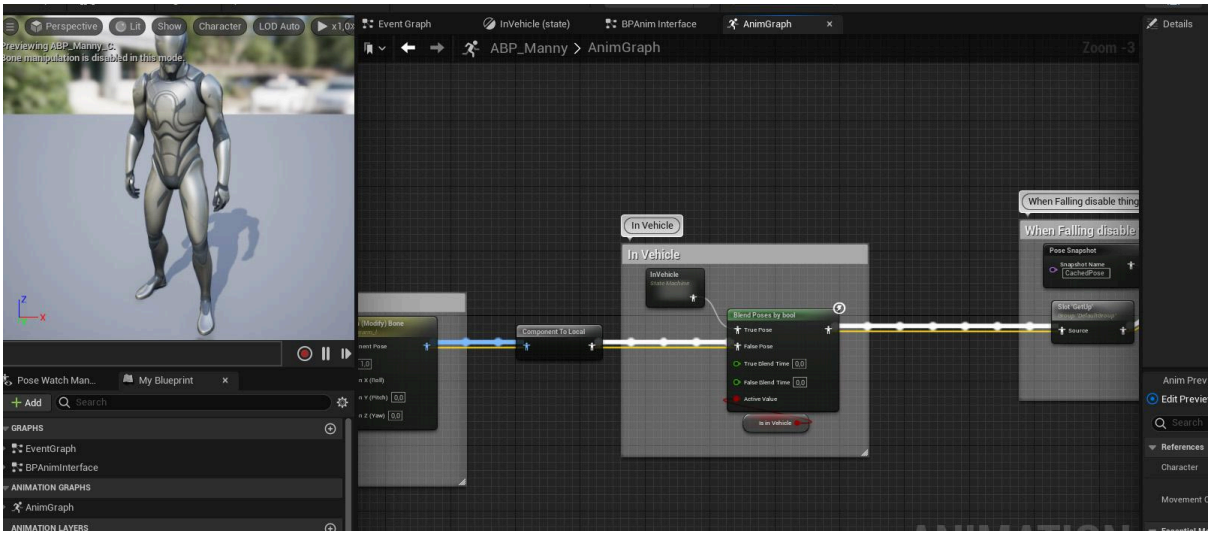
When the player wants to get out and is pressing the E key again we will just spawn a **BP_TestPlayer** or your custom character and then possess that using a possess node.



Remember that this is just one way to handle this.

Player Driving Animations

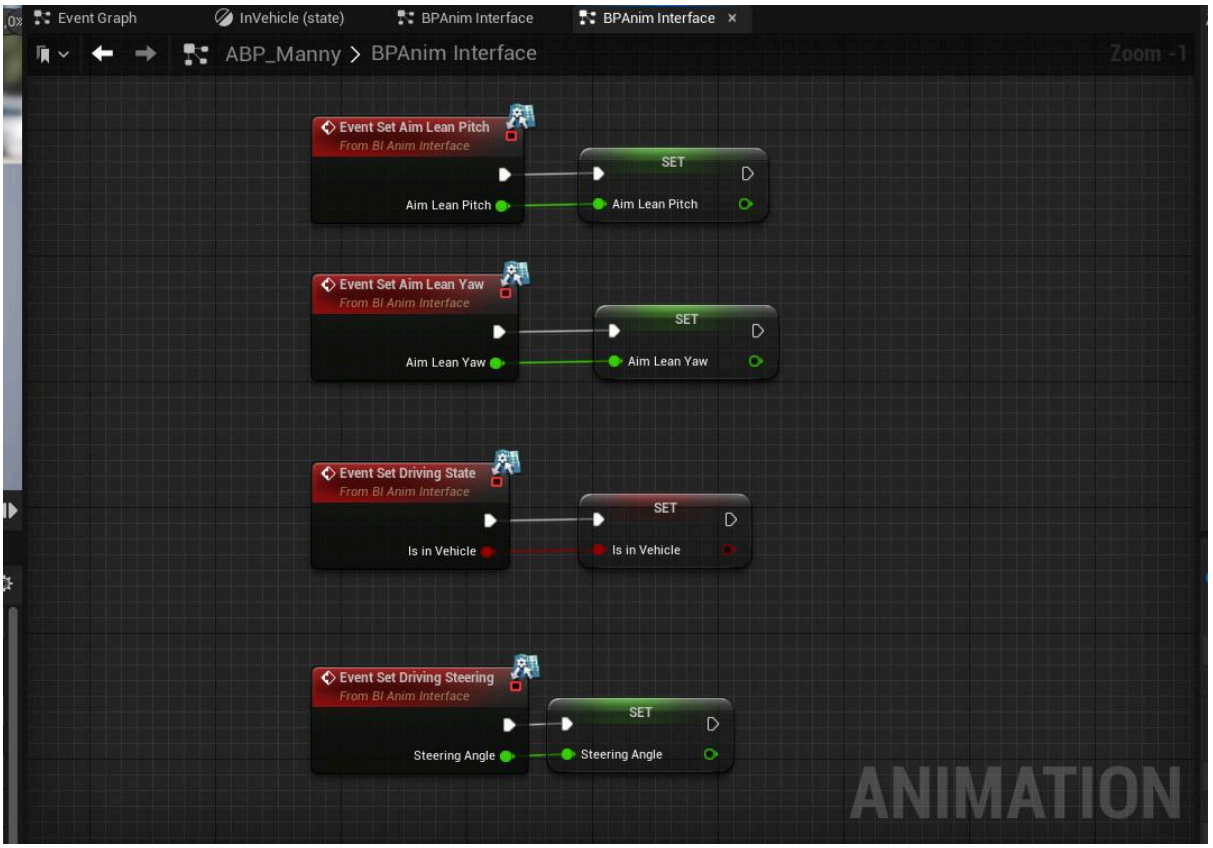
Here is how the player driving animation system works. **BP_ExamplePlayer** is using the default Unreal skeleton and the default Quinn mannequin assets. Example player blueprint is also using **ABP_Quinn** animation blueprint. That blueprint is inherited from the **ABP_Manny** blueprint that is acting as parent animation blueprint so all of the changes are made there.



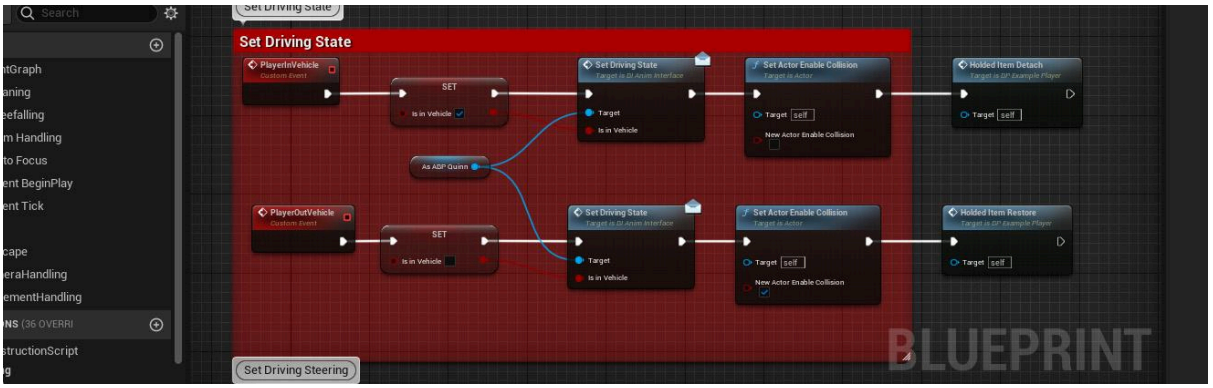
That animation blueprint contains basic character movement and in this case, also extra control for driving. *AnimGraph* is the place where all this logic is set up. When the "IsInVehicle" variable is true, it will blend the "*In Vehicle*" part in. That means it will make the player character sit and hold the steering wheel. "*In Vehicle*" part is using a simple blend space that consists of different still animations. That blend space is controlled with a single float variable called "SteeringAngle". That value goes from -65 to 65 that means -65 is fully left, 65 is fully right and 0 means the player is holding the steering wheel straight.

It's important to understand that the animation blueprint is only containing the logic for animation blending and the actual controlling is done in other actors via a blueprint interface called **BI_AnimInterface**. That interface contains events and the ones we are interested in this case are Event Set Driving State and Event Set Driving Steering. That way we can

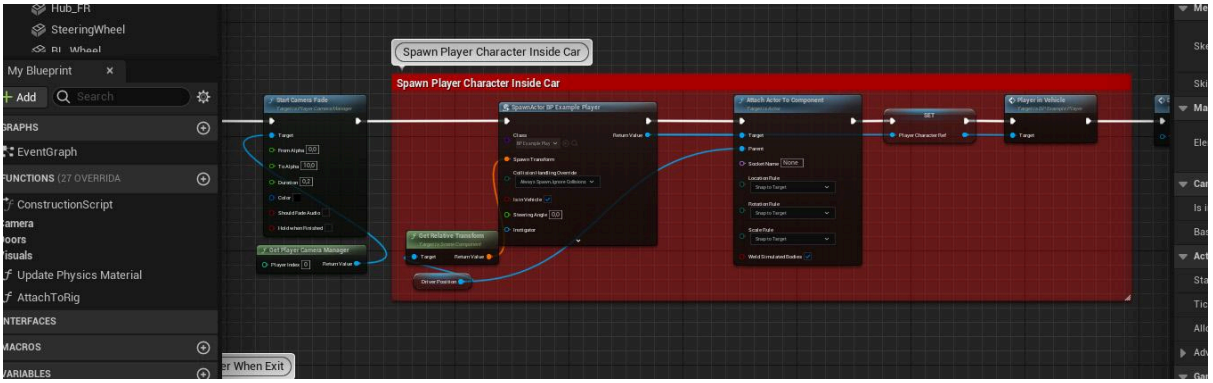
easily say to the animation blueprint when the character is in a car or is left from the car and how much that character needs to turn the steering wheel.



BP_ExamplePlayer always has a reference to this animation blueprint so it's possible to control and say when the player is in a vehicle. That blueprint also contains custom events called PlayerInVehicle and PlayerOutVehicle that will trigger the Set Driving State event that I covered previously. It will also handle some collision settings to drop the backpack etc.. Under that, you can also find an event called SetDrivingSteering that will pass steering angle to the animation blueprint so it can drive that blend space I covered previously too.



BP_P_VehiclePawn blueprint will spawn that **BP_ExamplePlayer** actor when the possession event is called. After that it will attach that player character into the *Driver Position* component that will control where the player will sit. Then it will create a new variable for that actor and trigger the PlayerInVehicle event that I covered previously. That way it will make the player character sit and hold the steering wheel.



Only thing left to do is to control the “SteeringAngle” variable so that the player character is able to turn the steering wheel. That angle is calculated in the **BP_P_VehiclePawn** too and then passed to the player. In this case we trigger that “Set Driving Steering” event that I previously covered and that will then pass that value to the animation blueprint. When all of this is working, it will make the player character turn the steering wheel when vehicle wheels are turning.

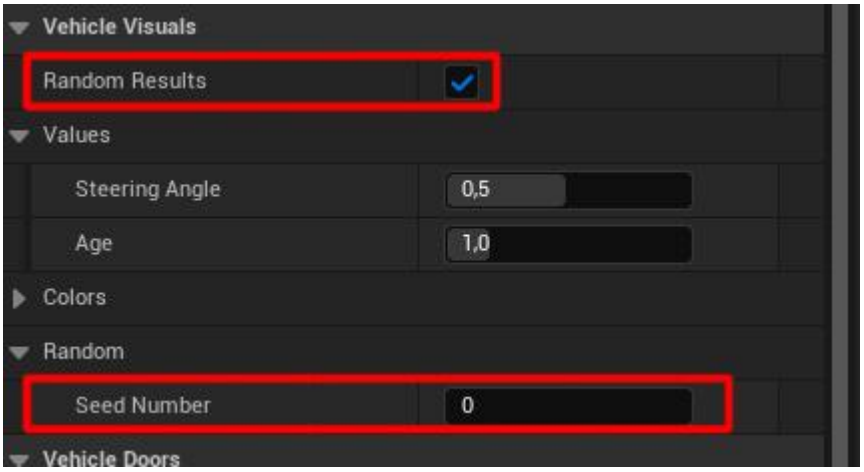
If you want to use a custom Metahuman character with this system then you need to retarget these driving animations and add the required animation logic to your animation blueprint that is covered at the beginning of this section.

Static Vehicles

Usually you don’t want to have too many drivable vehicles that all would use physics because that will cause lots of performance issues. That’s why this pack also comes with static vehicle versions. Main principle is the same as vehicle meshes are split into groups to support opening/closing doors etc.. Static vehicles also include more advanced material instances that support world aligned details that would not work well with the drivable versions.

“Steering Angle” will control wheel turning that allows for variation between vehicles. This value goes from 0 to 1 where a value 0,5 will keep wheels straight, 0 = wheels are turned fully left, 1 = wheels are turned fully right

Static vehicles are also supporting pseudo random generation. This helps to generate different versions of the cars based on their world space location and seed number. If you want to generate different results, you can change the “Seed Number” to something else. If you don’t want to use this option, you can disable the “Random Results” boolean variable



Vehicle Visuals

All of the vehicles are using advanced materials that allow them to achieve very high detail results. It will use tiling “sub” layers and vertex colors to blend between them. On top of that, there will be few input texture masks that tell where edge damage might be, where to apply rust and dirt etc.. Static vehicle materials also support top down layer blending that helps to add sand/dust/leaves on top of vehicles.

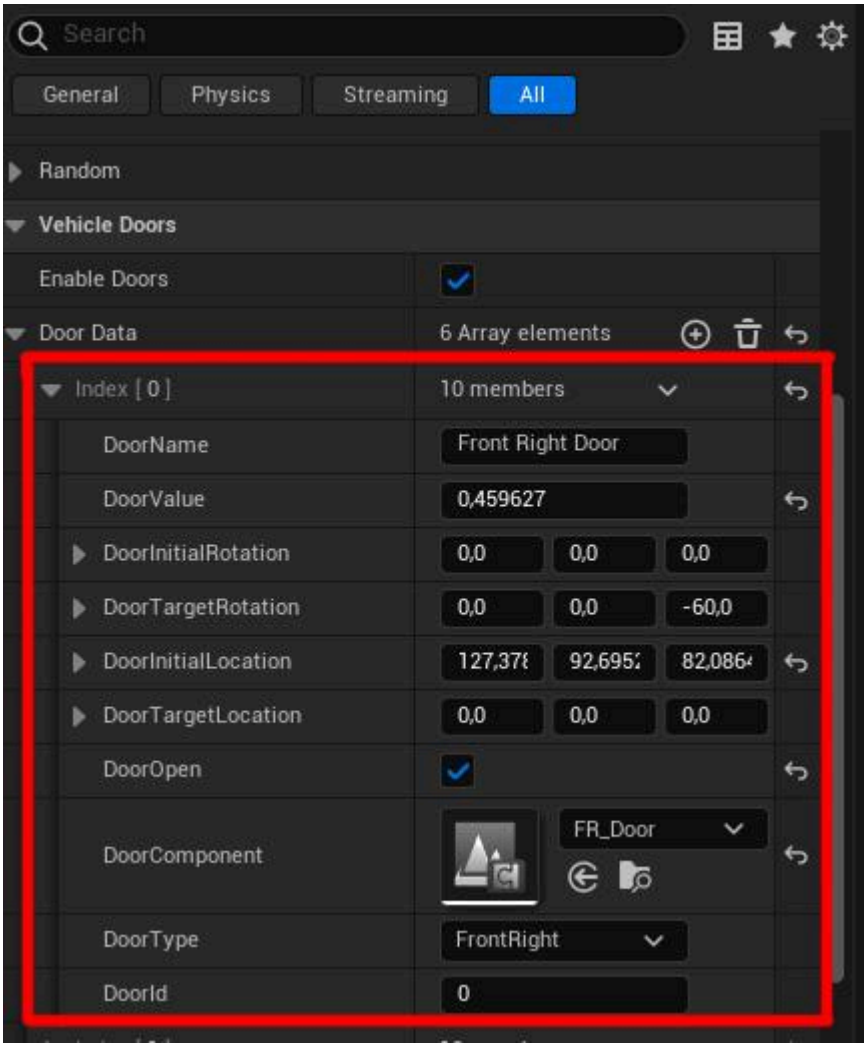
It’s possible to control all of these values separately but the actual blueprints will control only a few “master” parameters. “Age” parameter will have the biggest effect and it will basically control how damaged and old the material will look. Age value goes from 0 to 10 where 0 = new, 10 = fully damaged/old. Usually it is advised to use a value between 0 and 2.

Vehicle Doors

Like mentioned before, it's possible to open/close doors at runtime. This is done by using a generic system that allows X number of doors per vehicle. It will once again use a line trace to see if the component contains a tag called "Door". If that is true, it will use a blueprint interface called "**BI_LevelObjectInteraction**". to run functions inside a vehicle blueprint.

First function is called "*DoorInteraction*" that will try to identify what door it is in the "Door Data" array. If that function is able to find the correct door, it will then run a custom event called "*OpenDoor*. That event will use a timeline node to use a curved animation for door movement. When the timeline node is finished, it will set "DoorOpen" values that are specified in the "Door Data" array.

Most of the control is happening in that array so it's important to understand what different variables are doing. This way it's also possible to manually specify what doors are open. "Door Data" array will contain 6 items, each of them will control a specific door. "DoorName" will tell you what door it will affect. "DoorValue" will control how much the door is open or not where 0 = fully closed, 1 = fully open. "DoorTargetRotation" and "DoorTargetLocation" will specify the door transform that will be used when the door value is 1. "DoorOpen" boolean needs to be enabled in order to allow the door to function.



"DoorInitialRotation" and "DoorInitialLocation" variables will specify starting values and you don't usually need to change them (leave to default values). "DoorComponent", "DoorType" and "DoorId" variables are also something you don't have to change unless you are creating a totally new vehicle type that differs from this scheme.

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
