Creation of a modular and procedural environments based on current urban architectures.

Author: Pau Fiol Lorente

Director: Josep Serrano Recuero

Bachelor's Degree in Video Game and Design and Development

Year: 2022-23

Index

Abstract

This project deals with current modular and procedural technologies and its usage in 3D Art for Environments. The use of tools to create and distribute elements based on a series of parameters is increasingly common for the creation of digital, urban spaces, as it speeds up the creative process and can generate a multitude of unique versions.

The goal is to deepen the author's knowledge on existing city building approaches and replicate the results with a modular and procedural approach using non-standard workflows.

The result of this project is an urban space within Unreal Engine, a detailed guide on the approach followed for its creation, and the assets used to construct it.

Keywords

3D, Urban, Environments, Modelling, Modular.

Links

Presentation video: here

Blender project: here

Index of Figures

Index of Tables

Glossary

Procedural Modelling: a family of techniques that generate geometry from a set of rules.

Visual programming: A programming language that allows for the creation of programs graphically, often following the concept of “boxes and arrows”.

Modular: Construction out of prefabricated units with standardized dimensions that allows for quick, flexible assembly.

Normal: An object, typically a vector, that is perpendicular to a face. Within Blender, the vertex normal vector is the average of the faces a vertex belongs to.

Z-Fighting: Also called plane fighting. Visual artifacts created by the near-similar or identical values in the distance to the camera.

Instances: Copies of a resource such as geometry meshes that share a single space in memory.

Photogrammetry: The process of obtaining reliable information about physical objects and the environment through the process of recording, measuring and interpreting photographic images. For this specific project, material visual properties.

PBR: Physics Based Rendering. A render process that accurately replicates real world light, materials, surfaces, and their respective interactions.

1. Introduction

1.1 Motivation

The use of tools to assist in the creative process is nowadays a core part of workflows across the industry. The reason behind choosing a project that involves procedural and modular workflows is to master their use and better understand its strengths and how to address their weaknesses, chiefly to allow for more varied results.

As for the topic of urban environments, it was an opportunity to delve deeper into a topic of personal interest, as architecture and urbanism are interests of mine and I am close to many people working on the field.

Additionally, cities are a common setting for games, so this project will serve as a personal portfolio project for future potential employers for the roles of technical artist and / or environment artist.

1.2 Problem Formulation

Often during the development of a 3D videogame, the largest man-hour investment is on the art department. To address this issue and have better scalability in projects, game studios often adopt approaches that minimize the efforts of the team by using modular elements to construct environments, and further this by using procedural tools to put together these separate objects.

Because of the existing need, these tools currently exist, both proprietary and open, but unless the gameplay itself is procedural, it is uncommon for high budget games to have their cities be entirely machine generated, as the refinement and unique details artists will add is yet to be matched.

These existing solutions are most of the time based in visual programming, in order to allow a more intuitive approach for artists without extensive knowledge in programming.

To summarize, there is a wide range of existing solutions when in comes to putting together modular buildings automatically and with variations, but most do not generate buildings that reflect how styles and technology change over time.

For this reason, the most reasonable approach within the scope of this project will be to choose and adequate existing solution and adapt it in order to incorporate additional variety.

1.3 General Objectives

There are two main aspects of value here:

- Modelling:
  - Creating modular asset packs at a high standard.
  - Properly reflecting different architectural styles.
  - Master the use of photogrammetry materials.

- Tool Use and Integration:
  - Creation of node-based tools to substitute repetitive tasks.
  - Iteration to create visually interesting results.
  - Optimisation for their use in game engines.

1.4 Specific Objectives

The results of this project will be divided in two sections:

- Urban Environment:
  - Small scale urban area (a few city blocks).
  - Mostly procedurally generated from modular components, within scope limitations.
  - Custom modular asset pack with various architectonic styles.

- Workflow:
  - Define easy to use software that interacts neatly with each other.
  - Create a replicable methodology to be used.

1.5 Project Scope

The urban environment will serve for personal use in showcasing my abilities to potential employers.

The workflow will try to inform other professionals in the game environment section on good practices and solutions to achieve architectural variety in modular generation of buildings and/or urban spaces. It is not its intention to be used in-game to create random maps every time, or to be used in city builders.

Given the limitations of 4 months and a single worker, a series of compromises will need to be made. To reduce the time investment, the modular assets will be low poly, as a sculpting would take too much of the time allotment.

For similar reasons, the amount of different building styles will most likely not match the amount a real city would, instead aiming for enough variety to showcase the point and a setup that allows for posterior growth.

2. State of the art

This section will provide context on the different approaches that are in use in the digital media industry for the creation of buildings and urban spaces, as per J. Burgess, N. Purkeypile in GDC talk *"Fallout's 4" Modular level design* [2]:

Manual: This is the most straightforward method. A building is completely modelled in a single piece ready to be used in game.

This may be used for games that have few urban elements and to not warrant the initial investment that the following methods have. It is used for unique buildings (e.g., the cathedral of a city) that would not see their elements reused. However, it is also true that even within a building many elements tend to be repeated, so even then repeated items such as columns or arches will be done aside and then instanced as necessary.

Modular: A set of constructive elements with a cohesive style and standardized dimensions is created to then be used to put together buildings, either in the 3D software, or the game engine.

- **Manual creation:** This is the current standard of the game industry. This allows for very fine control of the resulting spaces, which is important to other aspects of videogames, such as narrative or gameplay, as level designers are allowed to create the levels with ready assets. It also allows for a significant time save and ensures a uniform style throughout the different spaces.
- **Procedural creation:** This is the goal of this project. The handmade elements are then assembled following a set of rules and parameters. This requires dividing the elements in groups or tags that define its uses, valid connections, and style, and creating a set of rules that define what to use, where.

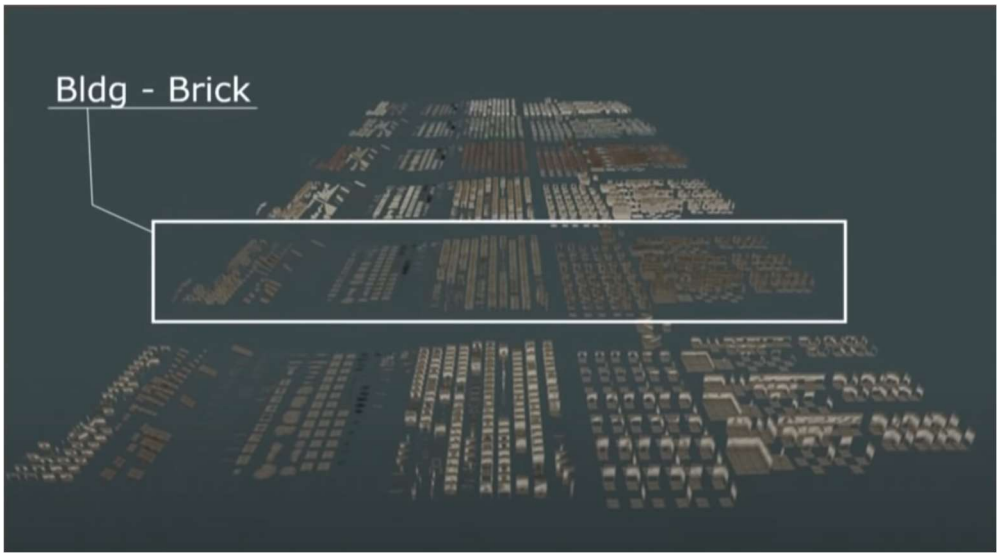

Figure 1: Fallout's 4 entire building asset kits, sorted by style.[2]

Within modular creation, a series of factors need to be decided prior to the kit being created, as addressed in "Game environment art with modular architecture" (Nataska Statham, et al.) [1]:

- **Types of modularity:**
  - Component Swapping: Having multiple elements designed to fir the same slot. For example, having multiple window borders fitting a window. It allows for easier variation within a set.
  - Fabricate to Fit: Allow for the creation of different meshes to fit a desired space. For example, one might have bases and tops for columns and the column itself would be enlarged/shrunk as needed. It is also useful when creating organic spaces that do not follow grids.
  - Bus modularity: A frame where multiple modules can be added at designed spots. Typically used in cases where the object is expected to grow, such as city builder games.

- **Granularity:** The scale at which components are made. The smaller, the finer it is the control over the outcome, but it will be more laborious to put together as there will be more elements per objects. This can be mitigated by creating prefabricated elements (also known as prefabs) combining elements into commonly used larger elements, while also maintaining those individual elements separate to use if needed.

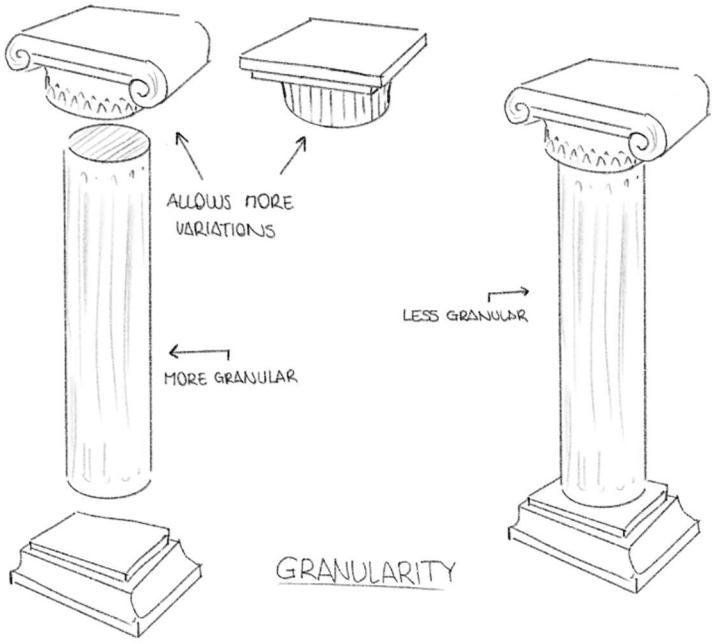

Figure 2: Granularity Example.

- **Tiling direction:** What directions is each element meant to be used, to ensure that they fit the grid and connect well with other elements.

- **Dimensions:** All elements must have proportioned dimensions to each other and its use. Most commonly it is a multiplication of human height.

- **Procedural/Parametric:** The geometry is completely generated from a set of rules. The main limitation here for its commonplace use in the industry is that the complexity of these rules can be overwhelming and requires extensive knowledge of programming to create and use. Research into creating more user-friendly approaches such as text prompts or sketches by using machine learning, but it is not ready for commercial use.

2.1 Software

There are many programs that would allow for 3D modelling and have a visual programming option potent enough for our purposes.

Houdini: A very established 3D animation software that specializes in complex simulations, particles, and general VFX. It is more than capable of handling the complexity. However, manual modelling for the individual would be less than ideal and would require to either build them completely parametrically or to use an additional modelling software.

Unreal Engine: The scripting and visual programming tools within the engine and its prefab system are powerful enough to create the city and buildings from an imported set of asset kits. It is commonly used and is the dominant option for companies where level designer put together the maps from modules when not using an in-house game engine, which we will not address here. With the release of Unreal Engine 5.2, a new framework for Procedural Content Generation has been released experimentally. This happened after the project was well underway but would have been the ideal tool for this project.

Blender: It allows for modelling, sculpting, and the use of geometry nodes all within a single workspace. Most other software listed here has a deeper set of tools in their specific field, but Blender has a broader reach and rarely one will need to swap to another program. The main reasons for Blender to have a small foothold in the professional industry are its reliability, which is slowly improving; the lack of a clear customer support system to fallback to and the lack of multi-user, automated pipelines that Autodesk can offer. However, it is entirely free and open source.

Autodesk Suite: 3DS-MAX and Maya can cover the modelling needs of the project, and Revit is entirely destined to parametric creation. However, since is focus is mostly towards architecture and engineering, the tools available are often suboptimal for our needs.

ArcGIS City Engine: Based on the Geographic Information System (GIS), this program is narrowly focused on the creation of urban environments. It fetches real geographic data and allows for a 2D sketching of the street layout, and then the generation of buildings given shape grammars or default pre-sets to be tweaked. It is used in urban planning, serious games, and movies.

This software is narrow enough that it would not make sense to use it, as the project is oriented to a more modular style of creation prevalent in the industry. However, it is a great reference to consider.

3. Project management

The tracking of work done will be done using Trello, by adding tasks under different boards divided based on the current status of the work.

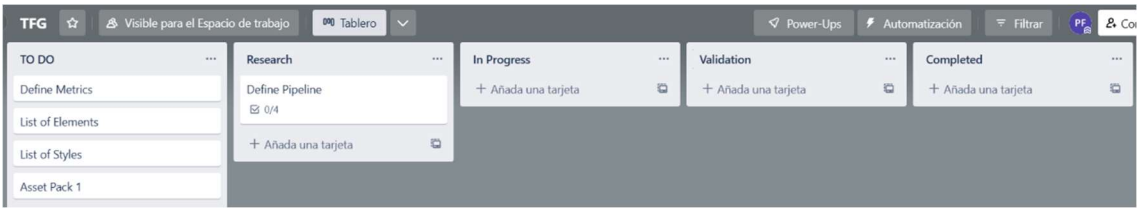

Figure 3: Trello Board

- To Do: Tasks yet to be started but is defined.
- Research: Tasks in preliminary stages, gathering the information required for it.
- In Progress: Tasks started. Checklists within the task itself will be used to provide further granularity of the progress.
- Validation: These tasks are pending review, explained below, and if the result satisfies the criteria, will go on to completed. Otherwise, it will return to “To Do”.

Validation:

For the validation process, the pipeline and the technical side will be validated by tracking time spent and comparing it with other iterations and manual modelling from previous projects. Lower work times are the goal. The technical aspect will be evaluated based on in engine performance and visual errors. Factors such as frames per second al load time will also be noted to ensure no readily apparent errors exist.

For the modelling part, validation is harder to define as appearance is a much more subjective matter, so it will be put under review of the project director after the author is satisfied.

3.1 Modifications

Ultimately, the Trello was abandoned in lieu of an analogic approach. The solution was a combination of a whiteboard, sticky notes, and a notebook. The reasoning for this was the ever presence of this elements in the workspace facilitating keeping track of the work, the positive psychological effects of a more tangible status, and the lack of need of access by multiple individuals. The functionality was effectively the same, with To Do, In Progress and Validation.

For time management, Google calendar was used instead, but was ultimately a rough guideline with worst case end dates. Because of the lack of experience in many of the elements of the project time estimations were most of the time very incorrect and made little sense to fully plan.

3.1. Risks and Contingencies

Risk	Solution
Reliance on external software	Strong version control of the files and autosave, in order to minimize issues generated outside of our control
Performance Issues within Engine	-Optimize and delete unnecessary geometry, -Additional optimizations in culling unseen building areas -Model with Level of Detail, if time permits
Interaction issues in software	Fallback onto manually export standard file formats, such as .fbx and .png
Time limitations for the additional modular styles	-Limit the number of styles. -Avoid exceedingly time-consuming styles (e.g., rococo) -Limit elements per module

[Table 3-1] Risks and Contingencies.

3.2. Cost Analysis

	Non-Commercial / Academic	Commercial Use
Software		
Blender	0€	0€
Unreal Engine	0€	0€
Alter Mesh	0€	99€
Quixel Megascans	0€	19€/month
Automatic Parallax Window Material	30€	30€
Labour		
Junior Artist		1500€/month
Hardware	0€	1500€
Total (4 MONTHS)		7705€

[Table 3-2] Cost Analysis.

This rough estimate is split by its commercialization because of the standard split in software pricing. However, because of the modelling choice being open source and free, the cost is significantly lower. The prices for commercial use assume the costs for a single developer with an expected revenue under 1.000.000€.

The computer price is a rough estimate of the cost of a PC with the capabilities to work in Unreal Engine unimpeded but is hard to give precise numbers because of the volatility of the market. Amortization of the computer is not considered.

4. Methodology

There exists a preproduction block where the goal is to test the functionality of different pieces of software, how they interact with each other, and how productive can they be considering the user's skills.

There is the task of defining the sizes of the modules, which are typically dependant on the gameplay. As there is no gameplay involved in the project, it will be based on industry standards based in the default character size of the engine.

For the choices of architectonic styles and their interaction, literature on urbanism and its growth will be analysed.

For the Production itself, an incremental approach would be best, as we are building on top of existing concepts. This approach is based on creating discrete independent goals and add them individually.

Incremental creation allows the testing of isolated elements and having more immediate results, allowing for a better time management, as any unexpected mistakes will not show up together at the end of the project. Additionally, the nature of the project lends itself well, as visual programming allows for a growth in complexity that is quite intuitive and can often be organised, in discrete blocks of nodes. It is the same for building modular packs, as a building can be defined with very few elements and then grow in complexity with the addition of further components and variations.

- Preproduction:
    - Research tools.
    - Research adequate architectural styles.
    - Visual reference gathering: Mood boards.
    - Define metrics, list of assets, variations, styles.

- Production:
    - Replicate existing solution for procedural assembly of modular buildings.
    - Create first modular style.
    - Test result and implementation of current solution.
    - Create second modular style.
    - Implement basic multiple style support in the tool.
    - Iterate implementation until results are as desired.
    - Materials selection, implementation and colour variation tools.
    - Time assessment of additional styles.
    - Create In-Engine final scene and add manual touch-ups.
    - Renders, video and build showcase.

4.1 Methodology Deviations

Because of the incremental approach taken, the tool created for the generation of buildings was often changing requirements. Since the changing requirements affected the module modelling, the development of the first architectural style was done in parallel with the tool itself, and the constant iteration was of greater time cost than expected.

Another task that was not considered was a proper analysis of the bridge plugin used: AlterMesh. The difficulties of using a small piece of software with limited assistance and finnicky set up were a time investment that pushed back on other tasks and limited the styles of buildings that could be done comfortably in time.

4.1 Software

The software that will be used for this project is the following:

- Modelling, UVs: Blender 3.3.0 LTS (Long Term Support)
- Materials: Unreal Engine 5.2 + Quixel Megascans
- Game Engine: Unreal Engine 5.2
- Visual Programming: Blender 3.3.0, Unreal Engine 5.2
- Blender to Unreal Importer: AlterMesh

The decision to use Blender over more industry standard software has multiple motivations:

- It is free and open source: This makes this method more approachable for small studios or individuals, as other modelling and parametric modelling options used in professional situations can cost thousands of euros yearly.
- It has a strong community online: It allows for easy access to documentation and similar existing projects. This platform is also a good space where one can share projects such as this one.
- It is a relatively new solution: The goal of this project is to find interesting ways to work modularly, and whilst interesting, more traditional options have already been explored in depth by expert professionals and this project would be just following their footsteps.
- With AlterMesh, instead of strictly geometry, the file opened in Unreal Engine is still parametric and can be adjusted as needed.

Quixel Megascans is a service that, in partnership with Epic Games, offers free PBR photogrammetry-based materials and assets for use within Unreal Engine, with liberty for modification and use if it is within Unreal Engine. The opportunity to work with realistic materials is something that has only become available without a great economic or expertise barrier recently and is also recently that real time applications are powerful enough to use them. Fort this reason, using it can be an invaluable advantage in the job market in the near future.

As of the delivery of this project, Unreal Engine has updated to 5.2. The engine version was updated because it released at a time when the work done in engine was still minimal, and thus migration posed no risk, as 5.2 was more stable on all relevant features for the project, such as Nanite and Lumen. It also implemented the experimental framework for Procedural Content Generation (PCG), which would have been an ideal software for this project. Because of the time of release, however, it was deemed to risky for the project deadlines to a new framework, but since its stability improvements, the project was upgraded from 5.0 to 5.2.

5. Development

5.1 Preproduction

5.1.1 Styles

5.1.1.1 Early XX Century

This was the first style created, and the one that underwent the most changes, due to its parallel development with the building generator tool as the requirements and possibilities of the tool were being discovered.

The main influence of this building are the apartment buildings of the Early twentieth century present in Spanish cities like Barcelona and Madrid. This style is a based in modernist and neoclassical architecture, but toned down because of its more utilitarian nature, as they are not geared towards opulence. It is important to note that this appearance has been and is still used today, but the internal structure has changed, meaning that nowadays with modern construction techniques there are different dimensions and elements that have changed. This is the case in the style with elements such as sliding, metal window frames, or a cheaper straight balcony railing, as opposed to the more organic and twisting cast iron railings of the past.

Figure 4: Early XX Century render in Unreal Engine

Figure 5: Early XX Century Mood board

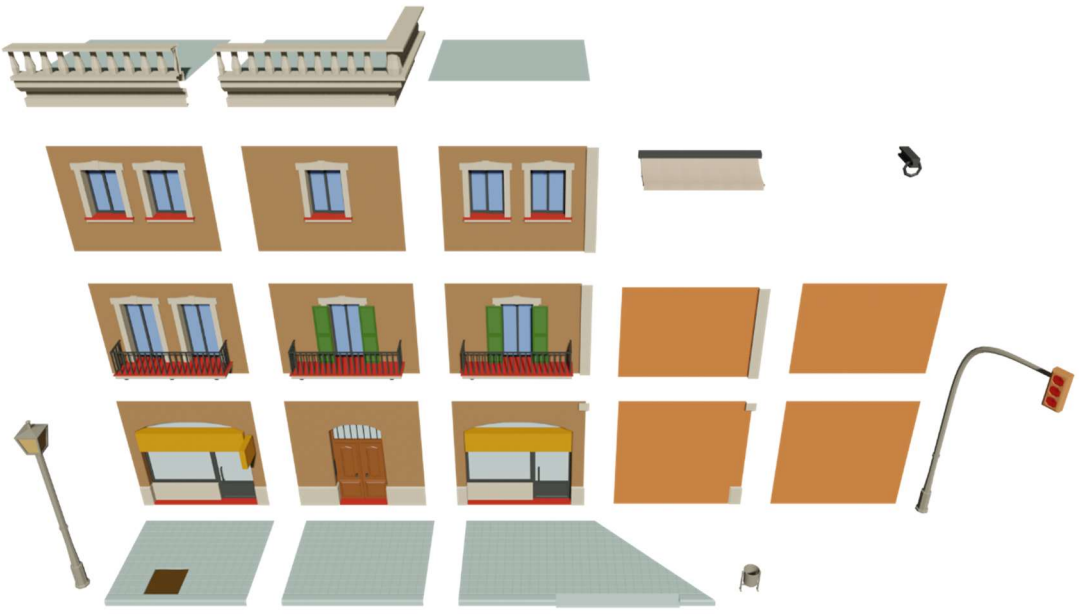

Figure 6: All Early XX Century modules in Blender

5.1.1.1 Modern

Modern here refers to the new building style that appeared with the population boom of the fifties and sixties in Spain, where urbanization skyrocketed and a high demand in housing created higher multiuse buildings to appear. This is reflected in the triple use of shop space in the ground floor, office space in the first floor, and housing above. It should also be noted that with the goal of maximizing floor space, balcony space was turned into interior space by pushing the façade outwards as much as possible, being limited by the urbanism codes of each city. These same codes also regulated how close an overhang could be to the neighbouring building, resulting in the tapered or chamfered corners of many of these buildings. It is also when attics became a premium living space, with additional balcony, now that lifts had become ubiquitous, in contrast to the first floor of the XX century style (also known as principal in Catalan).

Figure 7: Modern building render in Unreal Engine

Figure 8: Modern Mood Board

Figure 9: All Modern modules as seen in Blender.

5.1.1.1 Bauhaus

This building is very heavily inspired by Seagram Building (1958), a skyscraper in Manhattan by Ludwig Mies van der Rohe. The reasoning behind such a different building style was to create a visual point of interest via contrast and size hierarchy, to direct the viewer towards it. The initial suggestions provided by the tutor was a church, as they are across many cities' central points, have unique shapes that stand out, are and often large. The change of direction was because of a desire of avoiding religious imagery, while preserving point of interest concept. An interesting execution of this concept, with a focus on an intimidating, massive building, can be seen during the cutscenes of the game Control, as seen top middle right in the figure below.

Using a skyscraper could highlight the modular nature of this project, because skyscrapers create patterns by repletion of elements. The building was chosen as it personally engaged me, and Mies van der Rohe is one of the favourite architects of my father, who is in turn part of the motivation to do an urban environment in the first place.

The name Bauhaus for the style is the name of the architect society and school *Staatliche Bauhaus,* which Mies van der Rohe belonged to and became one of the main representatives.

Figure 10: Bauhaus building render in Unreal Engine

Figure 12: Bauhaus Mood Board

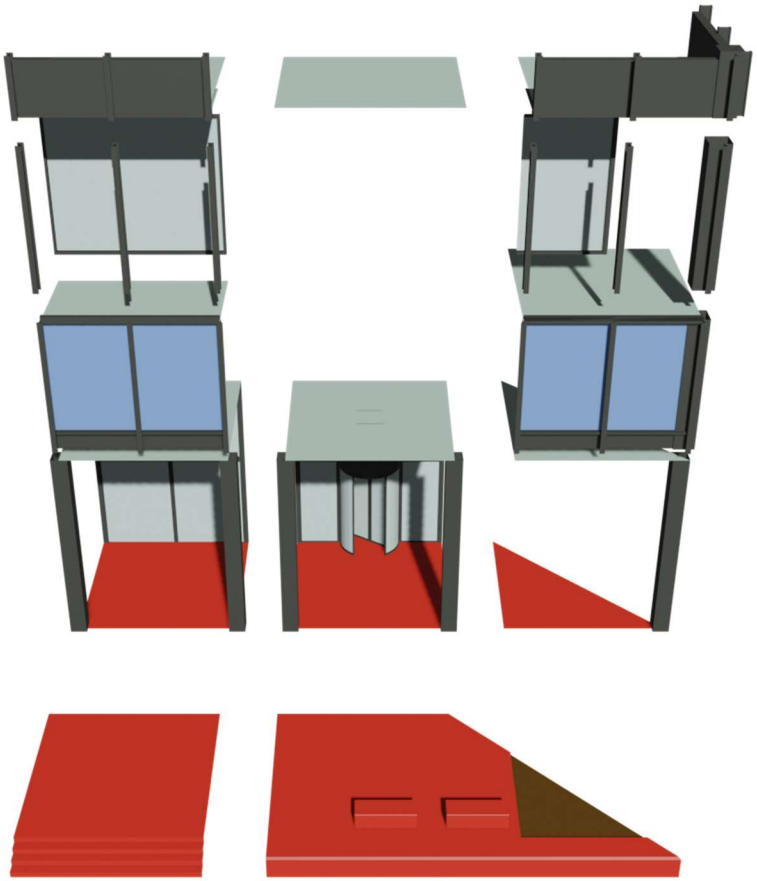

Figure 11: All Bauhaus Modules as seen in Blender.

5.2.1.1 Requirements

It is important to consider how the edges of each element will match with the other nodes. Vertices should preferably match those of other modules, as it will reduce the total vertex count when used. Additionally, modules can overlap, which can be used to create protruding elements such as overhangs or pillars in edges of the wall which can extend into the next module. Similarly, when creating corners any protrusion should be made to meet at 45º with itself and other compatible corners. Since the building generator tool is updated dynamically with each change to its modules, a straightforward solution is to regularly check how changes made in the modules are reflected in the final building.

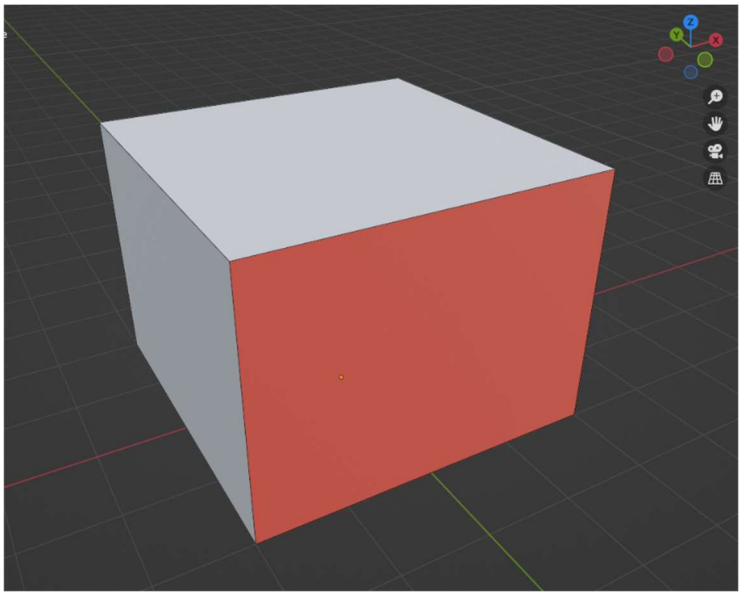

Figure 13: Relevant Face for module creation.

5.2.1.2 Metrics

The metrics of all the modules are based on a square base prism (width x width x height). These values can be defined within the building generator tool, as well as a second height value reserved for the ground floor (width x width x ground floor height). This means that when creating the modules, the goal is to fill the corresponding region of the volume, the -Y face for the walls and the +Z side for the roof. Corner modules meet with each other at the +X side (right hand). Protrusion can be any size, and indentations should be smaller than the width value.

The origin point of the modules should always be the centre of floor of said prism.

STYLE	**Module width (m)**	**Module Height (m)**	**Ground Floor Module Height (m)**	**Sidewalk Height (m)**
Early XX	4	3	4	0,2
Modern	4	3,3	8	0,2
Bauhaus	6	5	7,5	0,8

[Table 5-1] Metric list for all styles used.

5.2.1.4 List of Assets

Asset	**Category**	**Has Corner Variation**
Doorway	Ground Floor	No
Shop	Ground Floor	Yes
Balcony	Middle Floor	Yes
Windows	Upper Floor	Yes
Party Wall	Ground Floor	Yes
Party Wall	Middle Floor, Upper Floor	Yes
Roof	Roof	No
Roof Edge	Roof	Yes
Sidewalk	Sidewalk	Yes
Traffic Light	Decorations	No
Streetlight	Decorations	No
Trash bin	Decorations	No
Other Decorations	Decorations	No

[Table 5-2] Basic Module List.

This reflects the basic modules a complete building, but it should be noted that since the tool is relatively flexible, changes can be made:

- A style for Upper Floor is not required.

- Decorations can be greatly expanded, be it different decorations for each wall/roof module or a wider selection to be chosen at random.

- If a building is not intended to have party walls, the modules can also be removed. This is the case, for example, with the Bauhaus style skyscraper, as these kinds of buildings rarely touch neighbouring buildings, and it was not intended in this project.

- With minor modifications, the building generator tool can be adapted to support multiple variations of every module, as is the case currently on the Ground Floor for the Modern style, which selects between multiple modules

Because of limitations of Blender's Geometry Nodes, the order of the modules and their distribution in the different collections should be matched, or the Building Generator tool will need to be slightly adjusted. This is because the tool picks an instance based on its list number within the Collection.

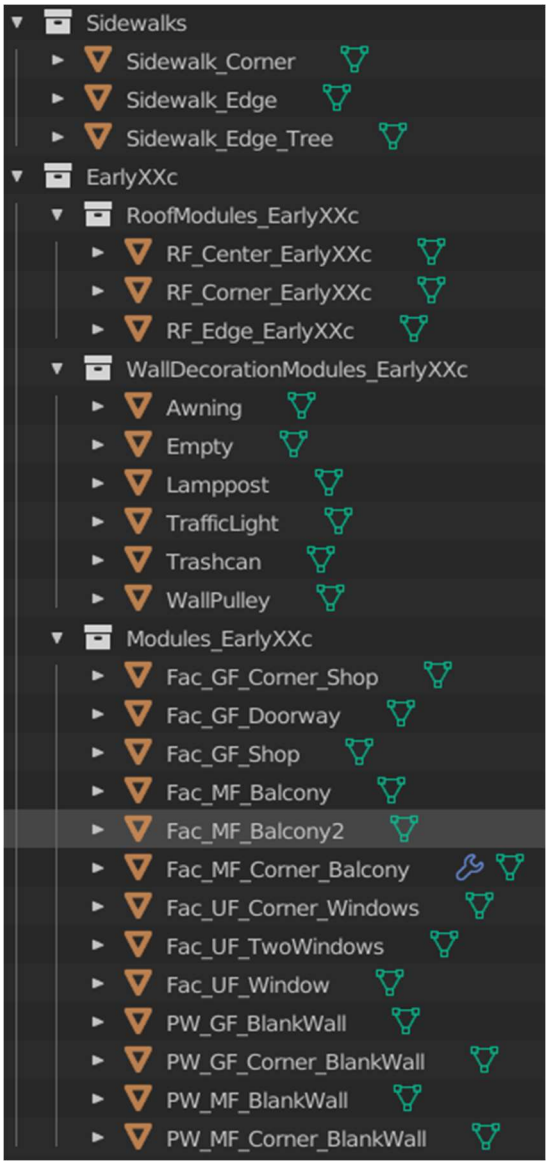

Figure 14: Elements used for the Early XX century style as seen in Blender outliner.

5.2 Production

5.2.1 Module creation

5.2.1.3 Modelling

The approach selected for this project is what is known as low polygon, hard surface. In essence, the goal is to keep the number of vertices at a minimum from the start and avoid if possible organic shapes that would difficult the overall process.

To start basic shapes (cubes, planes. cylinders…) will be placed to create the rough shapes. Since the objects in question are to interact with other modules and are architectural, it is key to ensure the metrics of all vertices are at adequate positions. To do so, the easiest way is to numerically displace the edges or vertices to the desired locations. The exact dimensions are informed by the module dimensions, various building codes, and ISO standards for elements such as steps.

Once the rough shape, or block out is done, the next step is to cut (divide in segments) the faces. The edges will be then again moved to their final position, outlining the shape of a protrusion or indentation e.g., balconies, or doorways. The polygons within this outline are then extruded inwards or outwards, again using numerical values informed by building codes. This process can be repeated multiple times from large to small extrusions until the final shape is created.

Another common tool used will be in setting faces, which will result in new polygons surrounding a smaller version of the selected one. This is useful for windows and doorways, as it allows for a quick way of creating the frames that surround them.

The third main tool will be bevelling, which allows an edge to be turned into faces, effectively rounding that edge. This allows to easily turn a prism like shape into a cylinder like shape, allowing the user to block out with prisms, which are easier to use when checking the metrics.

Once the final model is complete, it is good practice to remove any useless geometry, such as faces behind the main wall of the module. This is not to improve performance, but it will also simplify the UV and material assignment process.

When doing modules past the first one, it can be useful to check what elements of the previous modules might be useful. If so, duplicating the mesh and separating it into the various elements can speed up the process, as they can be used instead of the basic shapes during the block out process. It should be noted that since these separated meshes carry UV and material assignment information, this step can be done after these steps have been completed to further reduce workload.

5.2.1.5 UV Mapping

UV Mapping is the process of assigning the faces of a model a space within one or multiple image textures.

During this process the goal is to have the faces be flat and adequately scaled in order to avoid distortion and maximize the texel resolution. The textures used from Quixel Megascans have different scales for each file, meaning the scale of the UVs depends on the material in question. If the stucco material selected for the wall is 4mx4m the wall faces should occupy the entire width of the UV space. However, materials such as metals are provided in 2mx2m, so the scale should be scaled accordingly. If a surface is larger than the texture size, the UV assignment can either be cut using seams or be extended over the UV range (1, 1). This is only applicable on tillable textures which is the case for all Megascans.

Since we will be relaying in real time lighting in Unreal Engine, the Materials did not require baking any texture maps. This means that the UVs could overlap without causing any issues. This makes easier maximizing the area of UV space used.

For creating the UV assignments, seams are placed in the mesh to indicate the unwrapping algorithm where it can separate the faces. The ideal placement of the seams is wherever they will be least visible. This depends on each element, but a useful rule of thumb is to think the angle(s) from which it will be viewed. For example, since the balcony modules will be above the camera view from the player, the inner top edge is probably the least obvious seam. Another solid way to do so is to use as seams smaller edges reducing the potential area of mismatch.

Finally, seams are to be placed at the edges separating one material from another, as it is totally intended for a difference to be there.

The Unwrapping algorithm will generally do a perfect job if the seams are adequate, but occasionally the result will need to be realigned or squared. It is also important for directional materials e.g., wood planks, bricks to ensure the rotation is correct and consistent between modules. Additionally, the Cubic projection algorithm was used in certain elements, such as windows, because the seams that were created fitted the real-world versions of the object, and the cubic projection does not require seams to be placed.

The unwrapping was done separately by materials, in order to avoid visual clutter while working.

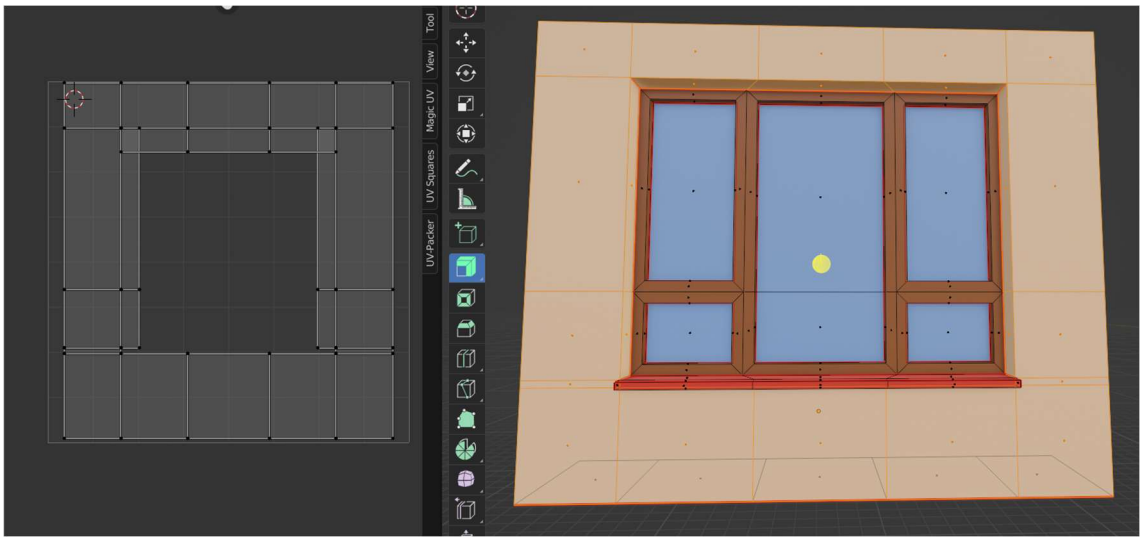

Figure 15: UV mapping of a window module's wall. Note the overlap and seam locations (red edges) at the corners.

5.2.1.6 Materials

For the material assignment, colour coded materials were created and assigned per face to each module with the goal of having the set up for its import onto Unreal Engine. Each material was assigned a colour simply for visual clarity.

In Engine, the materials used were mostly Megascans materials imported via Quixel Bridge. These materials provide a master material which was modified in two major ways:

The textures were virtualized: This makes the textures move from its usual location, RAM, to a hard drive allocation. This was done purely out of necessity, as the Megascans materials are composed by three uncompressed, high-resolution images (ranging from 1k to 8k). Since buildings range in 8 – 12 materials, the high memory costs made it impossible to smoothly use Unreal Engine with 8GB of memory. This solution is for a specific case only and is by no means a professional distributable solution without any options to address it. Improving this situation might be a future development.

The second major change was the randomisation of the base colour. This is done by multiplying a random colour by the base colour texture. This colour is created by doing a three-way linear interpolation of three different colours which can be adjusted as parameters. The use of three of them is to create a region of colours that is significantly more varied than the results of a two-way interpolation. The seed, the value that defines which colour within the range is used is determined by the division of the meshes world position and a parameter seed. This allows the material to change colour for each instance, as two buildings won't be sharing the same coordinates. It also has the benefit of being a single material instance, improving performance. Additionally, a tint parameter ranging from one to zero allows to limit the strength of the tint or turn it off altogether, depending on the needs.

With the modifications of the master material completed, it was a matter of trying multiple materials to see which ones matched best the mood boards and masked the imperfections in the seams between modules. Some of the materials have multiple texture sets in order to add further variety. In other cases, such as marble, a simple colour change does not equal a different kind of stone, and thus multiple types of marble were used and changed manually.

The only deviation of this approach was done for windows and light sources. Light sources are a basic reflective material with emissive value and emissive colour parameters. For the windows, Automatic Parallax Window Material by BB Lab was purchased from the Epic Marketplace. This material provides parallax (simulated perspective by modifying flat images) rooms that are very low on computing cost, which was a requirement. It also provides a way of randomizing the different interiors. Unfortunately, the way it works was fundamentally different of the modular approach used by this project.

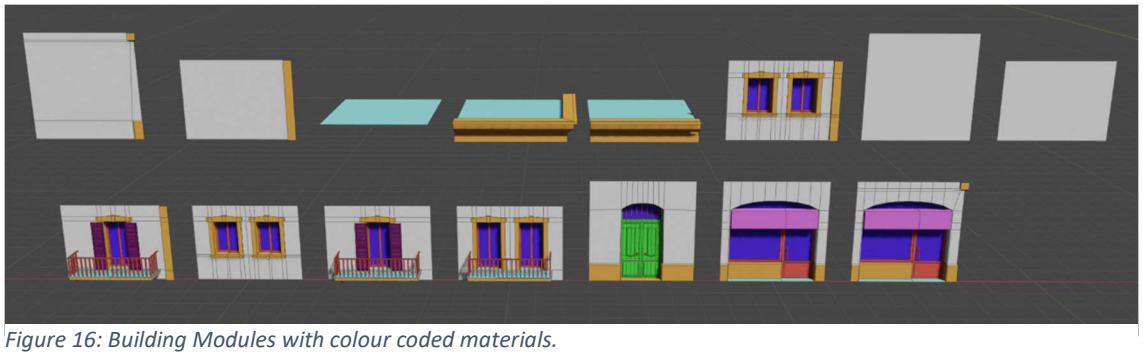

Figure 16: Building Modules with colour coded materials.

5.2.2 Procedural Tool creation

4.2.2.1 Parameters

The current parameters that one can provide to affect the result are the following:

Figure 17: Building Generator parameters as seen in Blender.

- **Building Width, Building Depth, Building Height:** the size of the building in each dimension, expressed in number of modules.

- **Module Width, Module Height, GF Module Height:** The dimensions in meters of the modules used.

- **Upper Floor Threshold:** The building floor in which the modules for middle floor will be replaced for the upper floor ones. If no upper floor is desired, the value should match the Building Height.

- **Sidewalk height:** The height of the sidewalk in meters

- **Sidewalk:** 0 = No sidewalk generated, 1 = sidewalk generated.

- **+X Façade, -X Façade, +Y Façade, -Y Façade:** Which sides should be created as façades, party walls, or non-existent. Because Blender does not support dropdown selectors, number values specified in the parameter's tooltip are used: Façade = 2, Party Wall =1, Empty= 0. This is in place to allow for buildings to be side by side without overlapping geometry and avoid creating unnecessary geometry in sides which will not be seen by the player.

- **Seed:** The value in which all random values are based. Changing this value changes all random distributions. Currently, it affects some decorative elements and the ground floor modules used.

5.2.2.1 Volume Generation

The first step is to create a basic geometry to later replace each vertex for a module. The geometry in this case is the combination of 5 grids (a plane with various cuts at regular distances), 4 for the walls and one from the roof. The dimensions and number of cuts of each grid are defined by the product of the module and building dimensions provided.

After their creation, each grid has its position adjusted to its corresponding side of the building, and its height is adjusted so the ground floor sits at Z = 0 regardless of the building height. Finally, the lowest points of the grid are adjusted to consider the height difference between the ground floor and the rest of the modules.

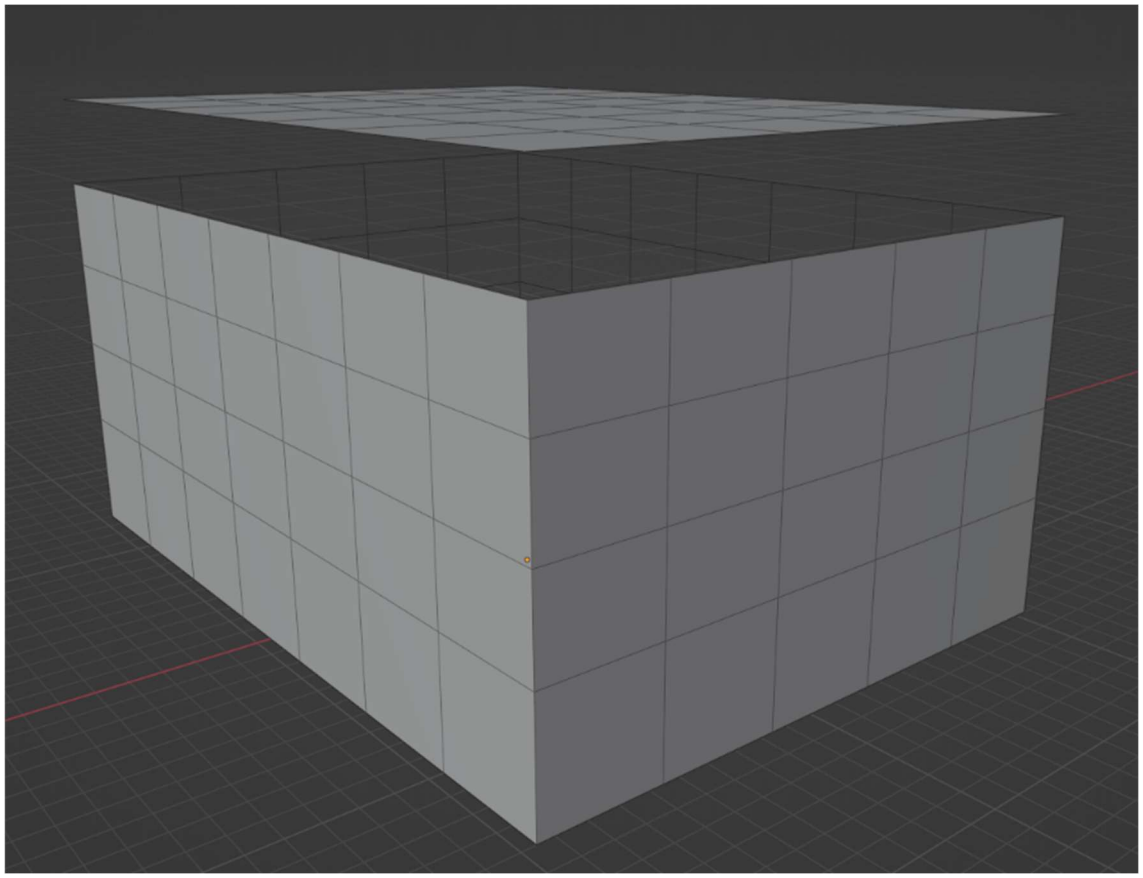

Figure 18: Basic geometry of the volume.

By using the node Instance on Points, a copy of a module can be placed in each of the vertex and evaluates all the inputs for each vertex that replaces.

To know which vertices should be affected, how should they be affected, and which instance to use, the approach of this project was to create selection masks. Selection masks are analogous to conditions in conventional programming, returning either true or false for each vertex or instance evaluated. For example, one can evaluate the normal vector of each vertex and compare it to a vector representing one of the cardinal directions to know if a vertex belongs to a specific wall.

These masks are then grouped together into single nodes, as they are reused throughout the node system and helps readability. Since groups can not access the parameters of the main node system, many of them must be provided the parameters via an external group input.

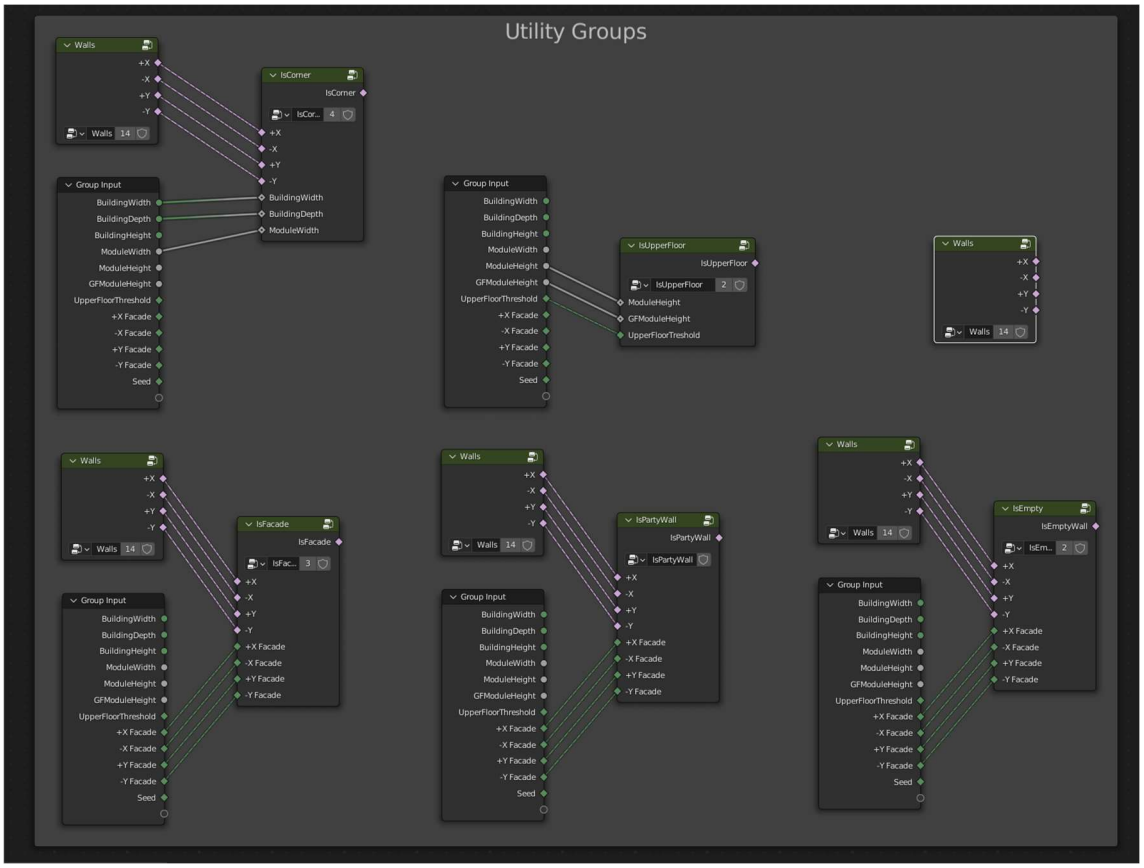

Figure 19: All commonly used Mask Node Groups.

The selection input lets the node know which vertices to replace, so this input can be used to filter out the walls marked as empty.

In Instance we can provide the mesh to be copied, or a collection of them, which one can then choose which mesh to use via the Instance Index if Pick Instance is set to true. Because of current limitations on how Blender operates with instances, the second approach has been chosen for this project.

This means that instead of working with object names numerical id values are used, which makes for a less readable system and requires the objects of the collection to be properly ordered. However, since it greatly simplifies the node system both visually and in efficiency terms, the trade-off is worthwhile. To choose the correct instance the value goes trough a series of switches connected to the various masks created. This can be conceptualized as a branching path of conditions: “Is this a façade? If so, Is this the ground floor? If so, is it a corner? If it is, use id number 2”.

In the rotation value we can provide the various rotations for each wall orientation along with a mask for the different walls, so all modules are facing out of the building.

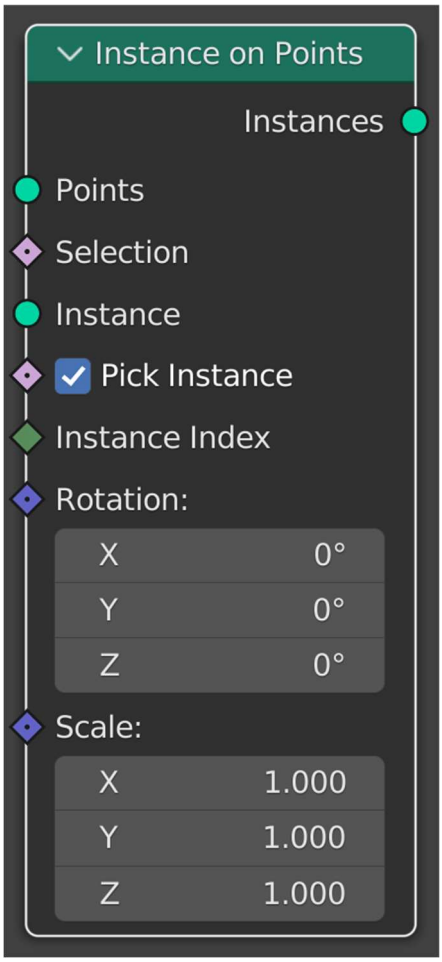

Figure 20: Instance on Points Node.

Instead of using the scale value set to X = -1.0 to mirror the necessary corners, which was the initial approach used, the mirroring will be done later in the node sequence to avoid face normal issues that would be problematic within the engine. For this reason, there is two Instance on Points For the walls, one for the mirrored modules and one for the non-mirrored, as it facilitates the selection moving forward.

There are also different Instance on Points for the Roof and the wall decorations, as the set of conditions used is based on the same concept, but the conditions aren’t. It could be merged into a single Instance on Points, but it would result in a larger set of nodes that would be harder to follow for any new users without a noticeable improvement on performance.

5.2.2.2 Instances to Geometry

When all instances are set, they are then turned again into geometry. If the project where to be used exclusively in Blender, this step would be negative, as using instances of a small object is better for performance than a single large object. However, because the final object will be a mesh in Unreal Engine, it is best to do so within Blender so that the transformation is as expected, and some clean-up can be performed.

We use Realize Instances to do so, and this transfers the values within the instances to the newly created faces, such as the UV mapping and material assignment.

It is after this step that we can flip the normal vectors of the mirrored instances to ensure that no inverted normal vectors reach Unreal Engine, in which they would be seen transparent, and can generate texturing issues.

Finally, by doing a merge by distance set to a very low value we can combine overlapping vertices, such as the corners of the modules, to remove unnecessary vertices and issues such as Z-fighting.

Figure 21: Result of the Building Generator in Blender, colour

4.2.3 Game Engine

4.2.3.1 Importing

After creating a new project and installing the AlterMesh plugin following the documentation of the plugin, the file path to the Blender executable must be provided. This is because AlterMesh essentially runs Blender in the background and imports the needed geometry dynamically each time the parameters are updated. It will be running a Blender program for each different Geometry nodes setup, so, in this case, three separate blender instances running.

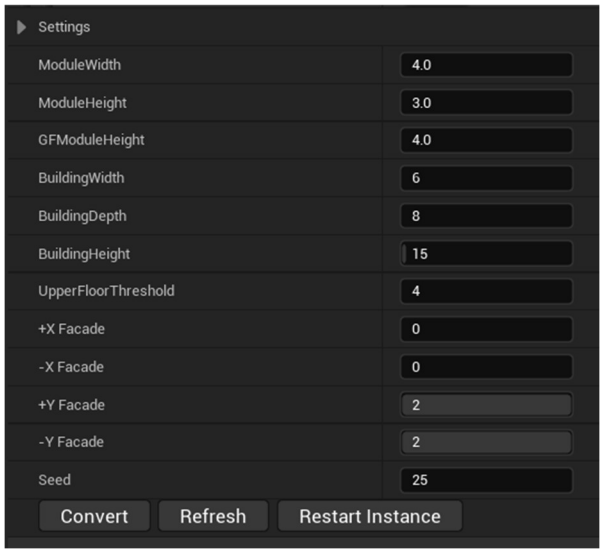

Figure 22: Building Generator parameters as seen in Unreal Engine.

To import the files, simply drag and drop into the project's content browser the desired .blend file. If multiple Geometry Nodes are contained into a single file, a dialog will allow for any or all the nodes to be imported as separate assets. In order to recognize which Geometry Nodes are to be imported, they must be present within the Blender's scene.

4.2.3.1 Material Assignment

AlterMesh objects preserve the material slots and the correct assignation, meaning that if multiple materials are used for different sections of the building in Blender, they will be the same in Unreal Engine. However, the materials are not carried along.

This is not an issue for the desired workflow, as the materials provided by Megascans are free only within the engine and Unreal Engine's Material Node system is the ideal place for doing the material customisations mentioned in 4.2.1.6.

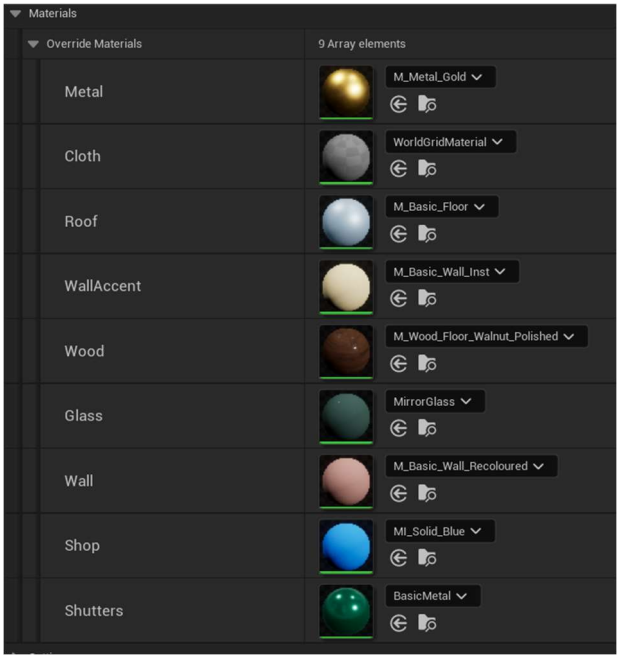

Figure 23: Assigned Materials in Unreal Engine

4.2.3.1 Tool Use

In the object inspector of the AlterMesh object the parameters of the tool can be used the same as in blender.

The Refresh option will update the mesh if any changes are made to the .blend file, which can be useful if modifications to the modules or the node setup are made and is faster than having to reimport the mesh from scratch. It can also solve certain problems that rarely arise, such as the mesh not loading when reopening the Unreal project.

The AlterMesh objects cannot be seen in play mode, meaning that they cannot be play tested. To be viewed in play mode or used in the final game build, the button Convert needs to be used. This brings up the regular Static Mesh importer, meaning the mesh can take advantage of the new Nanite system of Unreal Engine 5 for improved performance. This Static mesh is then placed at the previous position and stored as a regular mesh, meaning that additional copies can be placed.

The actor has the possibility to activate the “experimental: runtime” option, but the performance is severely affected. It is best practice to turn into static meshes, preferable Nanite, all buildings that are deemed final, in order to alleviate performance costs, even during level creation.

It should also be noted here that while dragging as a slider the geometry nodes parameters is possible, it is not advised, as it seems to request an export of each individual value change to Blender, which depending on the values modified and your computer can cause a crash. Occasionally, an AlterMesh asset will not update. This is because either the Blender file had its name modified during runtime or the Blender instance running in the background has stopped. This can be fixed by resetting the engine.

Figure 24: AlterMesh asset as seen in the Unreal Engine editor, including parameters.

4.2.3.1 Level Creation

The Level was built on top of the preset ThirdPersonGame provided by Unreal Engine. This preset, beyond the game controller and character, already includes a global illumination, a directional light that accurately mimics the sun, rotation included, a postprocessing volume for adjustments to the camera, and a skybox.

The ground plane was created with a custom material created from a Megascans road textures, set to tile at an adequate scale. To avoid an excessively apparent tiling repetition, noise textures at three different scales, all larger than the asphalt texture, were used to die and darken the surface irregularly. This makes the repetition much harder to see, and imitates stains, wear and tear and asphalt patches.

Afterwards AlterMesh assets were placed and had its materials assigned. Once ready, they were distributed forming city blocks, changing the parameters as necessary to achieve an interesting shape variety. Once a city block had the desired look, the AlterMesh assets were turned into static meshes, in order to save on performance. This is repeated for all blocks that form the main area. Since the main showcase of this environment was in the form of a render, viewers have no control over the view angles, allowing for a focus on the details only from the camera viewpoint. This, combined with the high computer demands, means that the main street seen in the presentation find is where most of the detail is place.

After all city blocks where placed, any gaps where the horizon could be seen was covered by scattering copies of buildings and city blocks at different distances, to create the illusion of a larger city.

The next step was to add vegetation, all of which was obtained via Megascans. Since the trees were a recent addition, only a few species are available. The trees chosen were European Beech for the sidewalks, and European Hazel for the Bauhaus patios. For each of the trees, there are multiple trees in order to add variety. However, since they are meant to be wild trees, they are not as manicured as trees in urban spaces. I chose the most fitting few and to avoid repetition, added slight scale changes and rotation to each tree instance. Alternatives were researched, but very few photorealistic trees were available for prices within the budget. Another advantage of the Megascans trees that can be seen is that they come with a simple wind animation, with adds to realism at minimal effort.

Figure 25: European Beech showcase, Quixel Megascans.

Decals were the final layer of elements added. Decals are a volume which projects a flat material onto a surface, at minimal performance cost, as is done within the shaders. Decals were used to add traffic lines and signs, manholes to the sidewalks, and text, in this case just my name to a few shops' signs, mostly to prove a point about the ease of use for custom diegetic publicity and brands that can add to worldbuilding. These were done in Unreal Engine because they can also be fetched from Megascans, perform better than image planes, and come with depth simulation incorporated, meaning the can look three-dimensional if needed.

Once the level was complete, a cinematic camera with 1:2,3 ratio, 1080p resolution and 35mm lens was created, the original camera animation was assigned to it and polished, lights were adjusted, and was set to render at maximum quality and antialiasing for the final video.

5.Conclusion

The use of external tools to assemble modular environments allows for a faster creation of environments with little sacrifice in quality if done properly. It also helps iterate more quickly the refinement of the level, by helping quickly adjust shapes and help with visual hierarchy, player guidance and overall aesthetics. However, the manual touch-ups to a scene really tie everything together, so making the environment totally automated could run the risk of creating bland and repetitive environments.

When it comes to software, however, justified why some industry software standards are standard. Blender, while perfectly capable as a 3D modelling software, is still in its infancy when it comes to node-based modelling. Compared to, for example, Houdini, many simple concepts require either obtuse approach or are impossible altogether. Another difficulty is clearly the debugging process, where the information travelling through nodes is hard to access, misleading, or incomplete. Blender is in constant development, and the current goal of the company is “Everything Nodes”, so it can be expected to improve in the future.

The use of a small, single developer plugin for the importing is something is full of dangers and pitfalls, with limited documentation and the lack of robustness that more established companies can provide, this tool would certainly be dangerous in a company environment with higher stakes. It is a possibility for similar tools to be built in house in order to have a dedicated programmer to the task to address any potential issues that might arise.

Finally, even with the ease of use of Unreal Engine’s lighting system and photorealistic assets, realistic environments are still hardly a tool meant for a solo developer, at least for entire products. They need extensive work in order to be properly integrated with each other and be optimised enough for real time use. The use of this approach means that either the product will be following the same aesthetic visuals of all others using this solution, or a large amount of time needs to be devoted to modifying them, create new assets and tune elements such as shading in order to stand out. This all means that a single individual, within a reasonable timeline, is not able to create environments that stand out on its own.

6. Future Lines

The release of Unreal Engine's Procedural Content Generation framework provides is an experimental release that should be capable of replicating the node tool made in Geometry Nodes, with the notable advantage of being entirely within engine, enabling things such as instancing, the use of decals, and the plethora of optimisations available more effectively. The largest of all changes, however, is that this will allow for building to be generated in game. A likely line of future is to rebuild the same building generator within PCG. This process will be easier now that the general knowledge is present, and the result using industry standard, cutting edge technology would be an ideal portfolio element for a technical artist position.

The second major line of work will be to extend the scope to entire urban spaces within a single setup, in order to be able to generate fully procedural levels. This is a large challenge, as in order to create a level or space at that same quality level and with enough variety, many edge cases and interactions between buildings, roads, street furniture and other elements such as NPCs or game elements need to be addressed.

Another great improvement that could be done would be to stray away from basic cube shapes and explore how to create module-based n-sided buildings, using for example Voronoi Cell noise as an urban grid.

Finally, there are subsystems that were planned but could not be achieved within the time given: coherent traffic signs, and cable grids.

7. Bibliography

1. Nataska Statham, João Jacob, Mikael Fridenfalk. Game environment art with modular architecture, Entertainment Computing. 2022. ISSN 1875-9521

2. J. Burgess, N. Purkeypile. “Fallout 4’s” Modular Level Design, in: Game Dev. Conf. GDC 2016, InformaTech, San Francisco, CA, US, 2016.

3. K. Lynch. ¿De qué tiempo es ese este lugar? Para una nueva definición del ambiente. Editorial Gustavo Gili. 1972. ISBN 84-252-0853-X

4. Geometry Nodes — Blender Manual. https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/index.html

5. Kammerbild. *Blender: procedural buildings with geometry nodes fields* [Video]. YouTube. 2022. https://www.youtube.com/watch?v=59PeIGmZQdY

6. AlterMesh – AlterMesh Manual. https://altermesh.com/wp-content/uploads/2022/01/AlterMesh_Manual.pdf