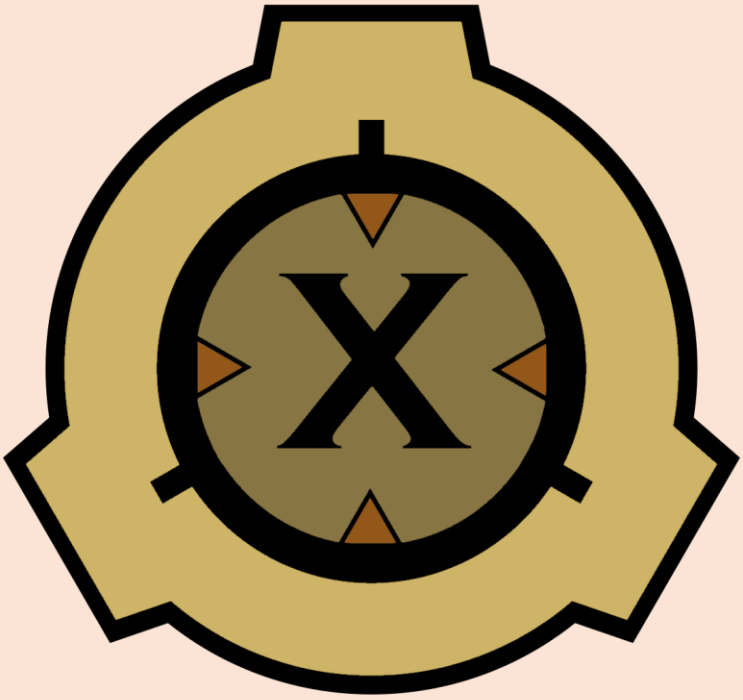
SCP
Secure. Contain. Protect

SITE-X MaD PRODUCTION PIPELINE
CLEARANCE LEVEL: ALL
DEPARTMENTS: MaD
AUTHORS: ADMINISTRATOR



OVERVIEW: This document will be your official Manufacturing Department guide and production information source for the foundation. This document outlines and describes the production pipeline of how assets in any site-related projects should be created in the foundation. This document provides a variety of information for many development aspects of the game, which can include performance and quality-of-line development process improvements, so please read **all** that **could** apply to your area.

DATE: 25th December 2022

TABLE OF CONTENTS

Section-1 | START HERE 3

Section-2 | ALL-ROUND GENERAL KNOWLEDGE 4

Section-3 | SCRIPTING PIPELINE 12

Section-4 | BUILDING/MODELING 13

- Building – Unions 13
- Building – Lighting/atmosphere of areas 14
- Building – Character Scale 15
- Building – Textures 15
- Modelling – Topology 15
- Modelling – Overlapping Vertices 16

Section-5 | MESH OPTIMIZATION 17

Section-6 | PROJECT VERSION BRANCHES 22

Section-1 | START HERE

Hello developers of the foundation.

This document is primarily applied to scripters, builders and modelers and not necessarily to areas such as graphics and sound artists.

This document is designed to help you to produce consistency and organization in your assets whilst also creating a development pipeline for each different type of developer.

This document focuses on the structuring of code and assets that are imported into the sites. By structuring code and assets in a set format, it will be much easier to go back and edit code and assets because they are all following a format.

It is key that you do follow these through your development process as it will make everyone's development process easier and more efficient.

When working as a developer on the site, it is important to keep the following notes:

- **Ask Questions.** Don't be scared to ask questions about things to your superiors, whether its design, layout, or atmosphere related, ask away!
- **Log your time.** Please for the love of God, log when you work in *#development-log*. It is inspirational to see your peers working on things actively and is a general morale booster to the team.
- **Show off your work.** We **want** to see what you are doing; communication is a value. Don't think you're a dick or a jackass, show off that thing you spent 4 hours making as we want to see it. Most importantly, be **proud** of your work.
- **Take breaks.** Getting burnt out can be bad, and you are recommended to **take breaks**. If you are feeling burnt out, tell us and make sure you log an inactivity notice it in *#inactivity-notice*. We care about you and if you ever need something please contact a member of high ranking or just anybody.
- **Don't make useless** things. Please for the love of God **don't build useless things**. I can understand if it was a specific request but if you make for say, a hyper detailed wall for absolutely no reason with 4 thousand parts in it, that's getting deleted.

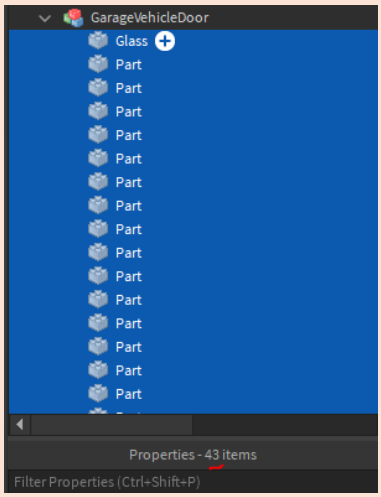
After reading the above, find your development aspect sections and read over the provided information.

>> O5 – SPOOKSTER

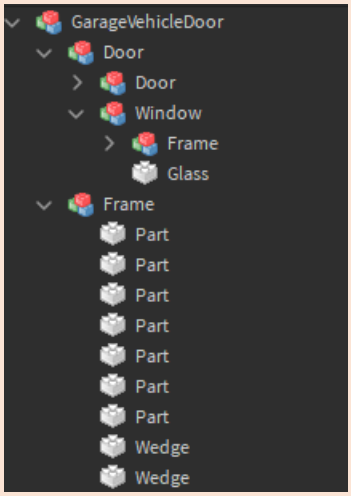
Section-2 | ALL-ROUND GENERAL KNOWLEDGE

When working with any asset that is being added into the game, in any aspect, make sure to keep it organized. When organizing, make sure to name the assets to what they are and separate the items within the models into groups so different parts of the model can be told apart from each other.

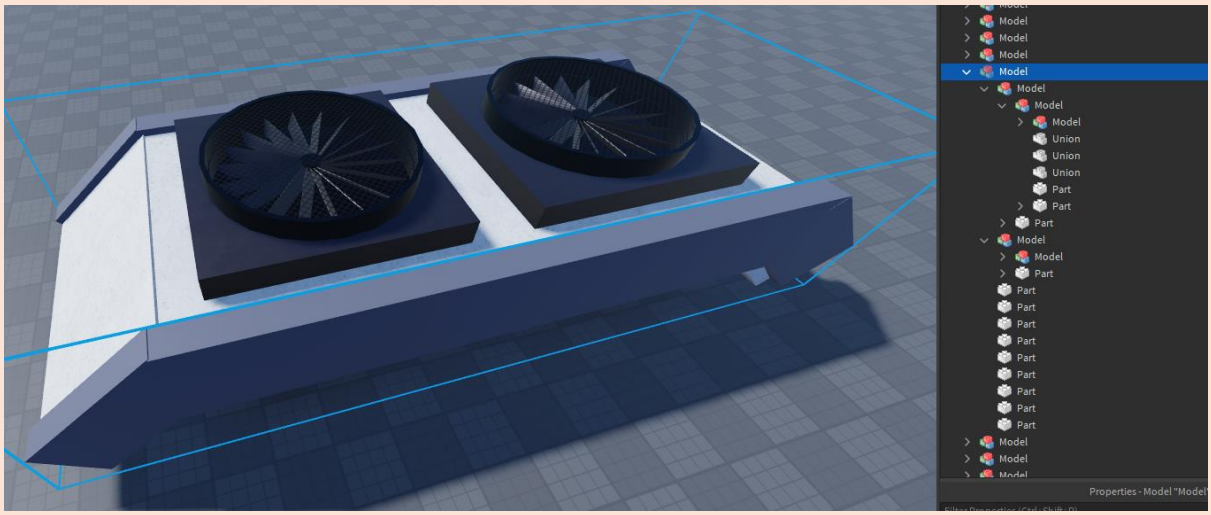
Here are some examples.



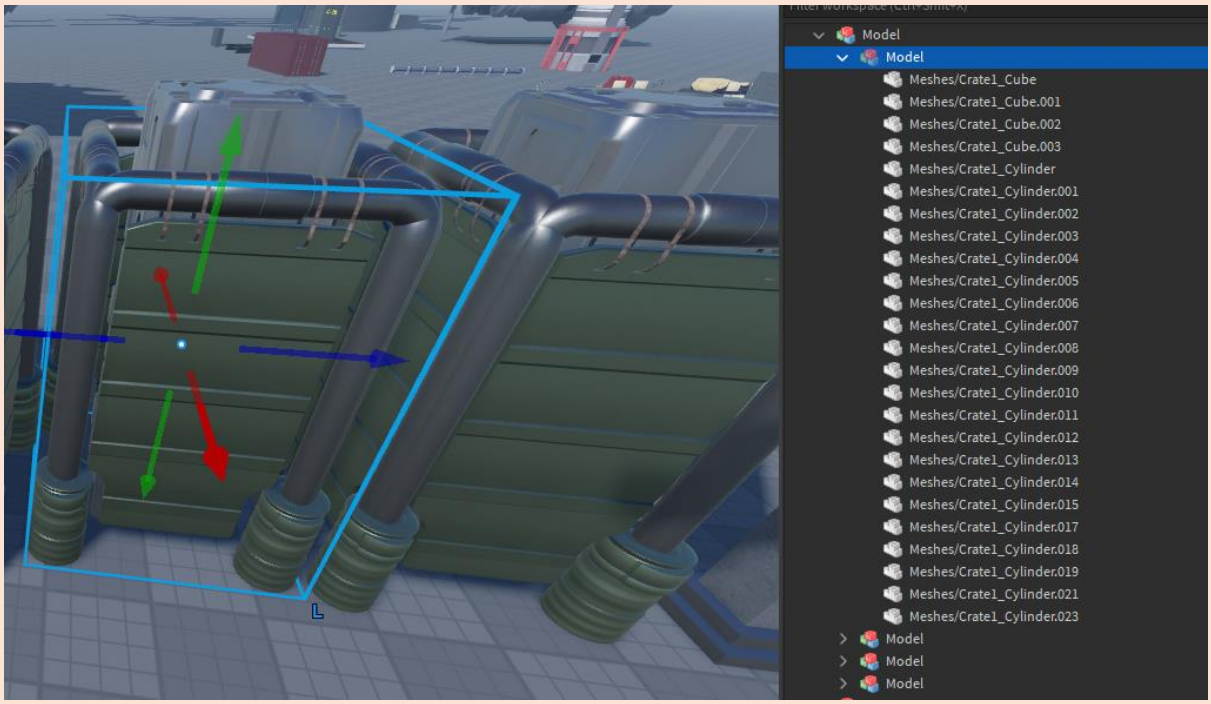
Bad Organization: Nothing is named, everything is clumped together and not separated into groups.



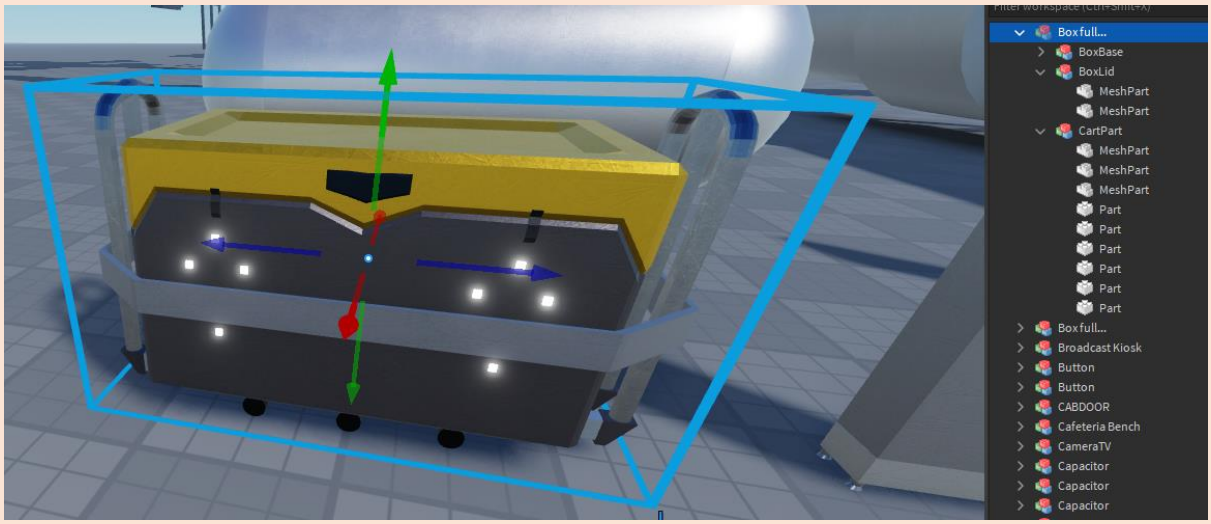
Better organization: Everything is grouped and named, easy to identify what is what.



Bad Organization: even though there are groups, nothing is named. Harder to distinguish what is what without looking through each model.



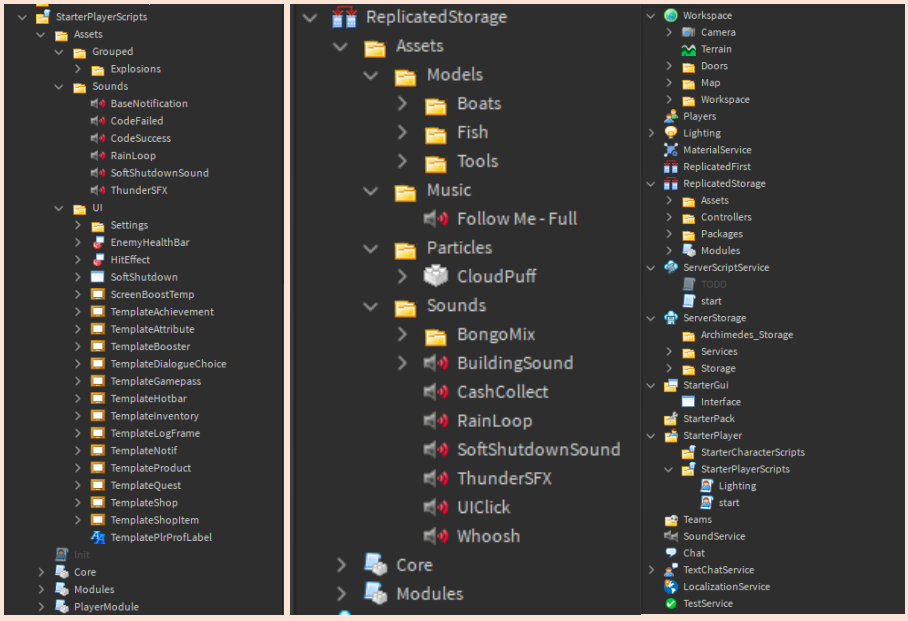
Bad Organization: The model is not named, and the inner meshes in the model should be further grouped into pieces, such as “barriers” and “centre”.



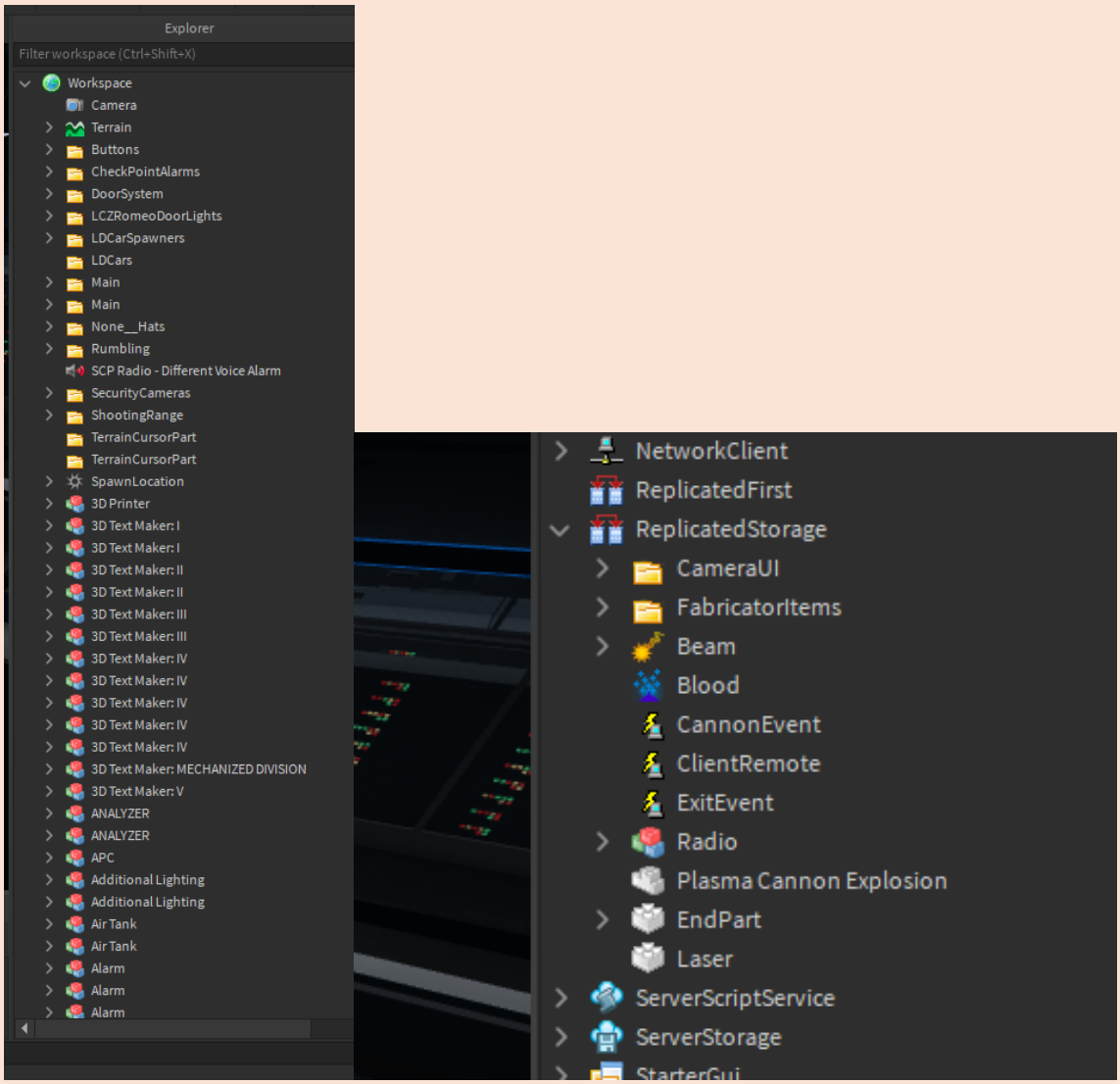
Good Organization: everything is separated and named.

Improving the organization of the models in the game can dramatically increase the speed and efficiency of working with models, not just to the builders but also to the other development aspects like scripting, which will also reduce headaches when editing the game. Being able to easily identify objects in the game with little searching and trouble will

Here are some examples from many of my own projects where I am addicted to organization in them, keeping everything separate and organized.



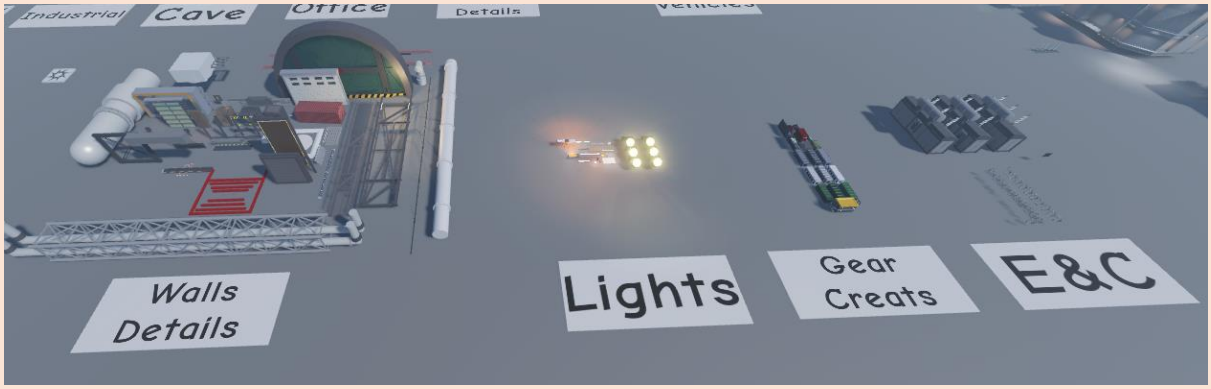
Here are some examples of bad organization from previous or current foundations that I am a part of which are horribly organized or have minimal organization.



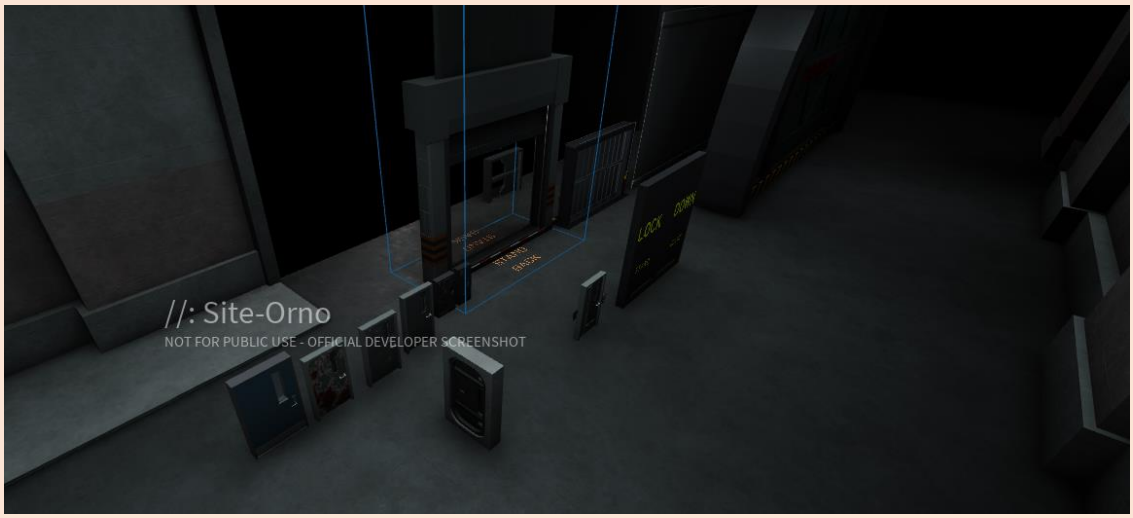
Onto another issue with general assets and storing them. **Keep assets that are not being used out of the main game**, put them into a separate (**organized**) asset place where they can be stored for later use. This dramatically improves loading time, performance and reduces memory usage in both games, both in team create and in the actual game.

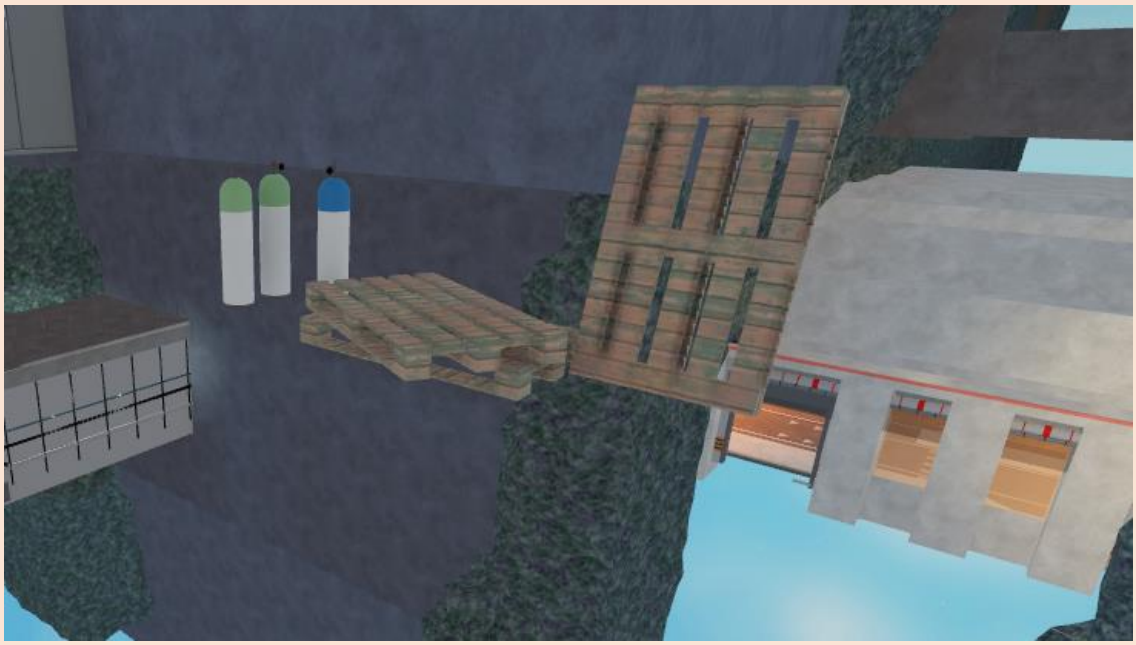
Here are some examples of what this looks like in action.

Asset storage place: heaps of assets which are organized and not in the main site game, improving load times and reduces load on any computer and also improving development efficiency by grouping them.



Example with Site Orno; random assets floating everywhere in the **main site game**.





From what I've shown, you can see there is assets that are just floating around.

This takes up (precious) performance (RAM, GPU and CPU) to which some is essential as they do not have beefy computers. So, it is important to keep things that are unnecessary out of the place.

You can still copy/paste models into the game so you can place them around the map faster but be sure to clean up the floating models around the map. Essentially, remove and unused assets that you are **not actively building with**. This also counts for anything under ServerStorage, ReplicatedStorage or any other parents.

Section-3 | SCRIPTING PIPELINE

For Site Orno's scripting team, the pipeline setup is using GitHub and Rojo (row-hoe) for project distribution and game scripting. This uses visual studio code as the code editor over Roblox Studio's built in. GitHub is used to monitor the amount of work someone does in the project and allows others to review peers' works' before and after they've been contributed to the project in either the development branch or the live branch.

When doing assignments (tasks), scripters are to create new branches from the development branch for that task and are to do pull requests to the development branch when submitting work. The dev branch disallows direct commits to the branch because of peer review. This allows us to keep track of the scripter's contributions, whilst also keeping a quality assurance on the code.

Prolonged absence of work in the repository also allows us to check with that developer if anything is happening and if they need to go on an inactivity notice, to which can be moderated if needed (e.g., no response after a week).

Rojo is a Roblox Studio plugin and Visual Studio extension which allows someone to synchronize a directory in their filesystem (on their device) to Roblox Studio through a project JSON file. The structure of the directory needs to also follow a file structure, so it is important that you learn how to structure your files, meta files, and code files so that it syncs the correct types of scripts, correct types of properties and such to the place.

Overall, the scripting development pipeline is designed to allow high insight into the scripting work being done and to make it easier to peer review and work before it is added into the development branch or the live branch. It is also designed in a way where we can limit the number of people that have studio access, as any changes to the development branch can be replicated to other developers, and therefore be synchronized in studio through GitHub and Rojo.

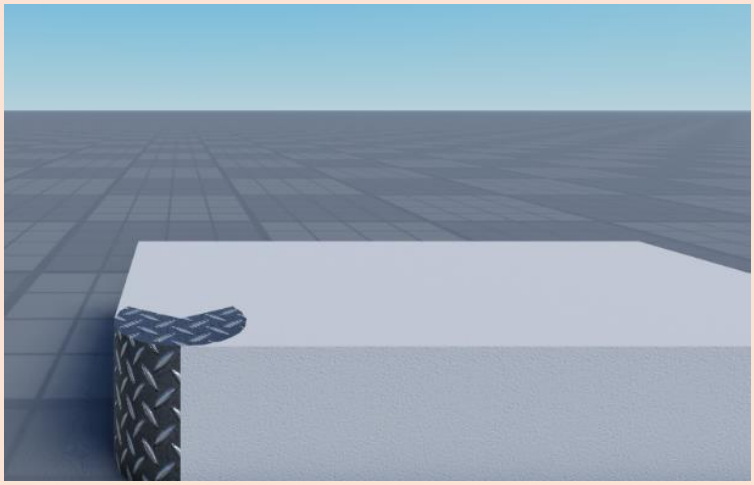
TODO: examples of using visual-studio-code, code-examples, guide on using GitHub/ROJO (video?)

Section-4 | BUILDING/MODELING

Building – Unions

When building any assets using any of the shapes Roblox provides, avoid using Unions as much as you possibly can. If you must use unions, I will also provide some tips to make sure you minimize any problems that can occur with these unions such as collision detection.

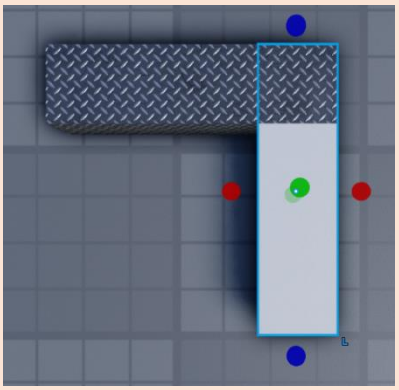
Firstly, Z-Index fighting is an issue that can be found in any game, this occurs when two different shapes share the same plane where some of their shape area overlaps with each other.



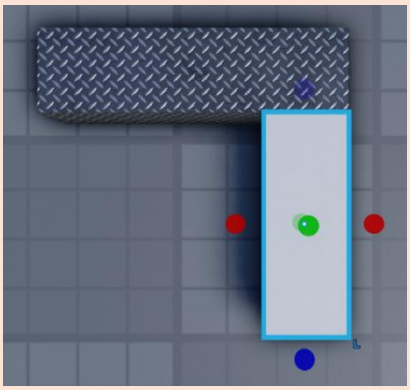
In the picture above, the diamond-plate cylinder is z-fighting with the part as the top faces share the same plane on the same y-level. To fix this, simply increment the y-position or y-size of either of the shapes by 0.01/0.001. This slight difference will mean that one shape is slightly above the other, and the camera will render the one that is above the other as the object on top.

Another way to counter measure this is to avoid collisions between parts.

Avoid collisions between parts



Collisions

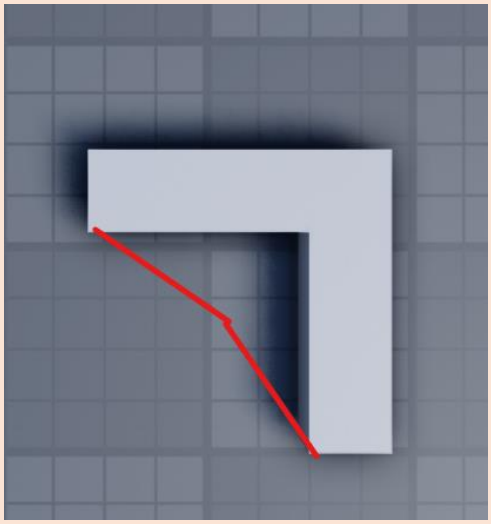


No collisions

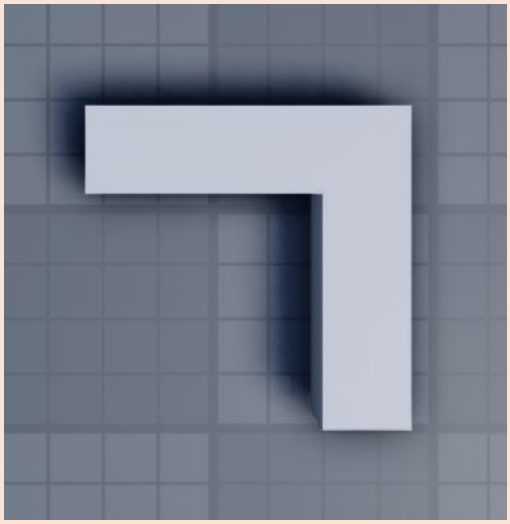
Separating them like this will prevent overlapping happening in the first place, however, it cannot be applied to all situations so the resize method would be very common.

Another issue faced with unions is the collision box. The collision boxes of unions can be outrageous at times as they make no sense, see below for an example.

Union collision box (Red)



What you see



Seen in the picture above, the hitbox should follow the "L"-like shape, however, it does a weird jump in between. Additionally, this model **should not be unioned** and should be kept as separate parts, as there is no reason to union this.

Building – Lighting/atmosphere of areas

The atmosphere, lighting and sound are important factors in the map of the game. They can dramatically change your experience playing the game. For builders, the focus of atmosphere and lighting in different areas will be at your discretion. Here are some good examples of lighting versus bad examples of lighting.

Good lighting

Bad lighting

Explanation:

Left - In this room, you can see various variations in lighting. For example, the hallways are lit up a bit more and you have lights highlighting points of interest, (garage doors, label sign, etc). The lighting matches the different sections of the room.

Right - This room is generally bright, which is okay BUT as you can see it's literally just same amount of brightness throughout, there is no variation or highlighting of certain areas.

Building – Character Scale

When you are building, please keep in mind the player. Please don't make a room with ceilings that are 4 stories tall. A very **easy** way of making sure this doesn't happen is to simply have a dummy with you while building. This makes sure that you don't make any large mistakes.

Have a look at the images below.

Example of good scaling

Example of bad scaling

The picture on the right has a basic door to which the height is 2x the dummy height, that is a ridiculous height given that the player's height would be the dummy's height.

Building – Textures

When dealing with textures here are some things to keep in mind:

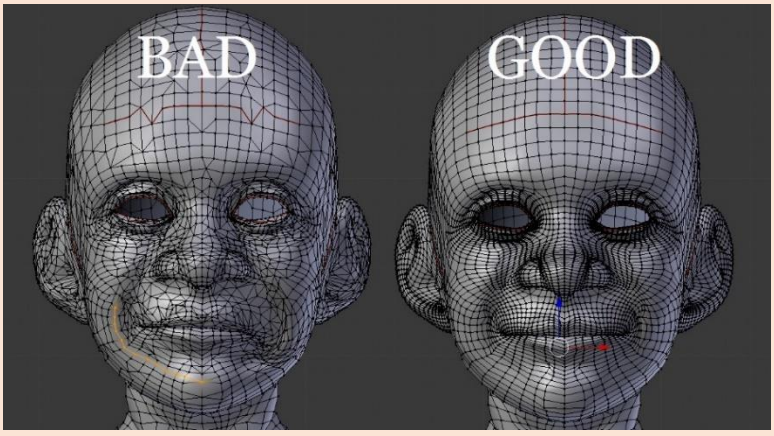
- If you can, please only apply textures to **visible areas**. If you need to make a texture more noticeable you can do so by either **making it darker or making it less transparent.**
- Multi-stacking textures can be a complete waste of performance, especially in bulk. It is ok for there to be few occasions where there are multi-stacked textures, but there should **never** be a reason to have **bulk** stacked textures everywhere.
- Reduce the number of unique textures and try to reuse and have a variety of the same textures that can be found in the game already.

Modelling – Topology

Topology is a tricky aspect of 3D modelling. Having the right topology in a model allows better efficiency and practical usage of those models, so when starting assets, try keep it in your mind as it is **very important** for any model.

Good topology basically makes an hour task of texturing into a 10-20-minute task.

Below are some good examples of what I mean. (Yes, I know it's a creepy example)

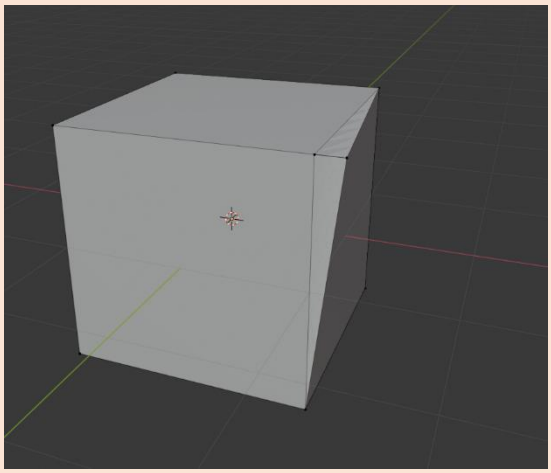


This goes without saying but **keep triangles in mind when modeling.** When you make a mesh, don't subdivide for no reason. This goes hand in hand with topology as great topology means that you don't have random triangles for no reason. Please also remember to demolish your models, when possible, to rid them of any unneeded triangles. If you do demolish your mesh's, please keep in mind that demolishing changes the topology of your model. This means that you may have to fix the topology.

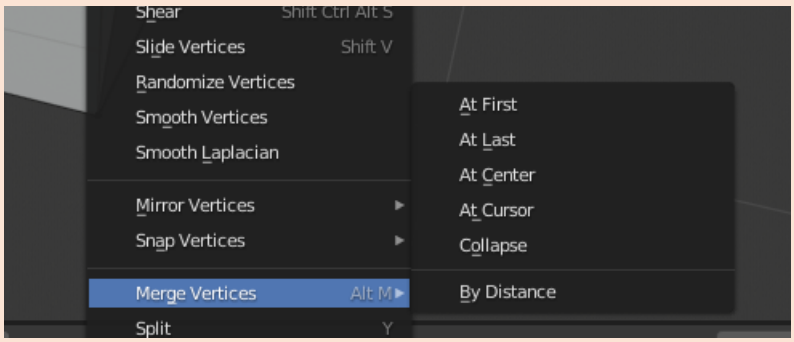
Modelling – Overlapping Vertices

This is a common mistake that happens a lot, I'm sure most of you know how to fix this easily but I will still go over it as it's very important to know especially on mirrored surfaces. When a vertex is overlapping with another, this tends to cause a large amount of shading glitches and just makes it a pain to model in general.

See below for an example.



Overlapping vertices can be easily fixed by selecting two of the overlapping vertices, **right clicking** and hovering over merge while in vertices select (press 1 on your keyboard).



After you do this, you can easily choose where you want the vertices to be merged, either in **the center** of the two vertices, **the first** of the selected vertices, or **the last** of the selected vertices.

I do not recommend using at cursor.

To speed this process up, you can simply press **Shift+R** to repeat the process **although you do need to have 2 selected vertices first.**

Section-5 | MESH OPTIMIZATION

When working with Mesh Parts and Unions, there are additional properties within the objects that when changed can potentially introduce major performance boosts when used in bulk and can also correct the collision of those meshes as well.

Examples of this can be found below, however, for this quick text section I will get slightly technical about why having specific properties can introduce more memory or performance hit; the CollisionFidelity, CanCollide, CanTouch and CanQuery properties provide the most performance boost when disabled in bulk. Here is why.

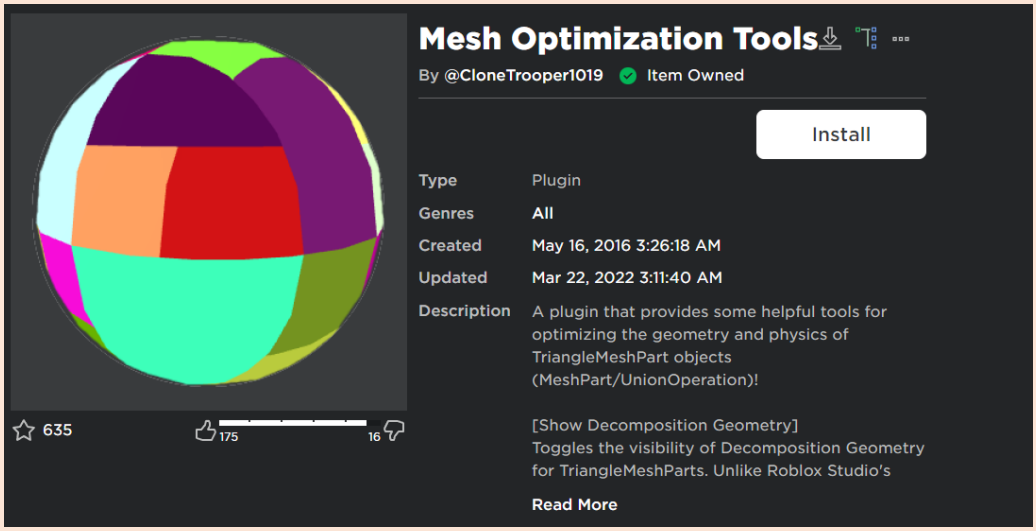
Every object within the Roblox environment has a physical bound and outline created of **Quads** for the **physics collision system.** This represents the physical boundaries of that object and is used for any collisions or raycast with the object. If the CanCollide **OR** CanTouch properties are enabled, the physics system will detect collisions within the environment given an object, and will either trigger a signal (CanTouch), or have physical interference means (CanCollide) with another object.

Additionally, the "CanQuery" property affects whether raycasts will detect that object, but that generally should be enabled unless the object is not meant to interfere with gameplay (e.g., bullet holes, unreachable objects, etc)

By optimizing the number of quads and triangles that are found in the meshes, the total number of quads and meshes which are fed into the physics system can be dramatically reduced by changing the CollisionFidelity of the mesh. By normal means, changing it from default should be your number one priority, however, some cases may be presented where you cannot change it from default

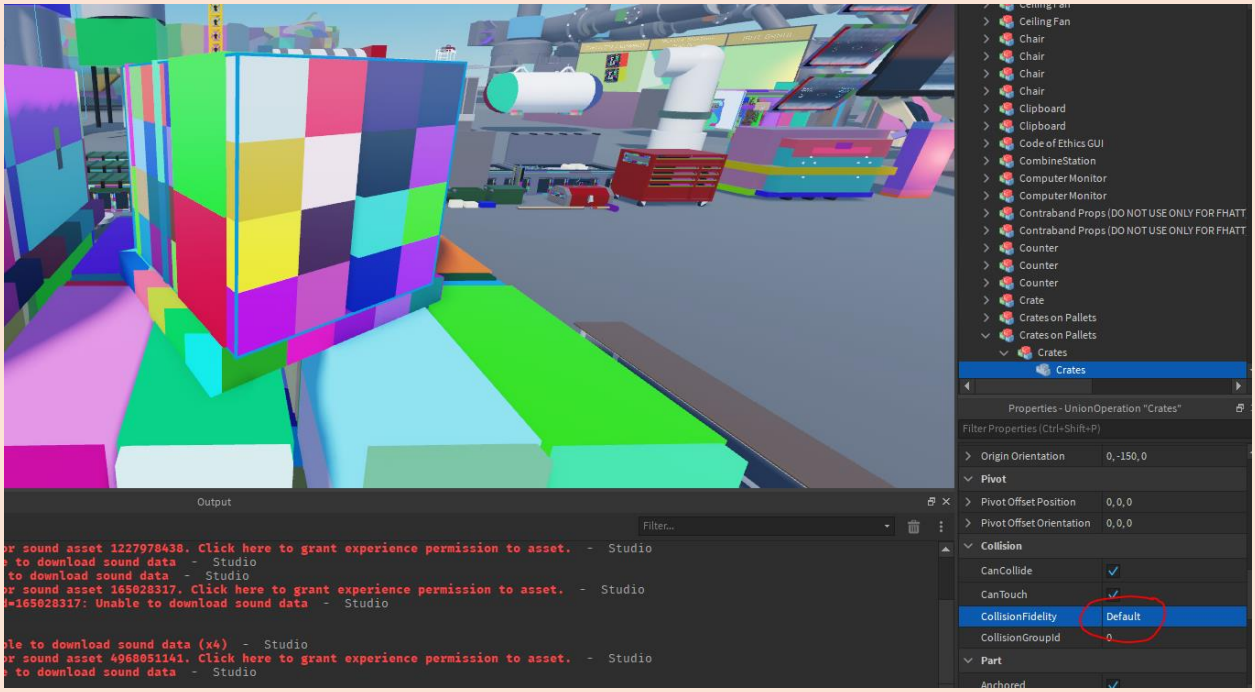
An extremely helpful plugin which shows the quads and triangles of the mesh, which was used in the examples below, is here:

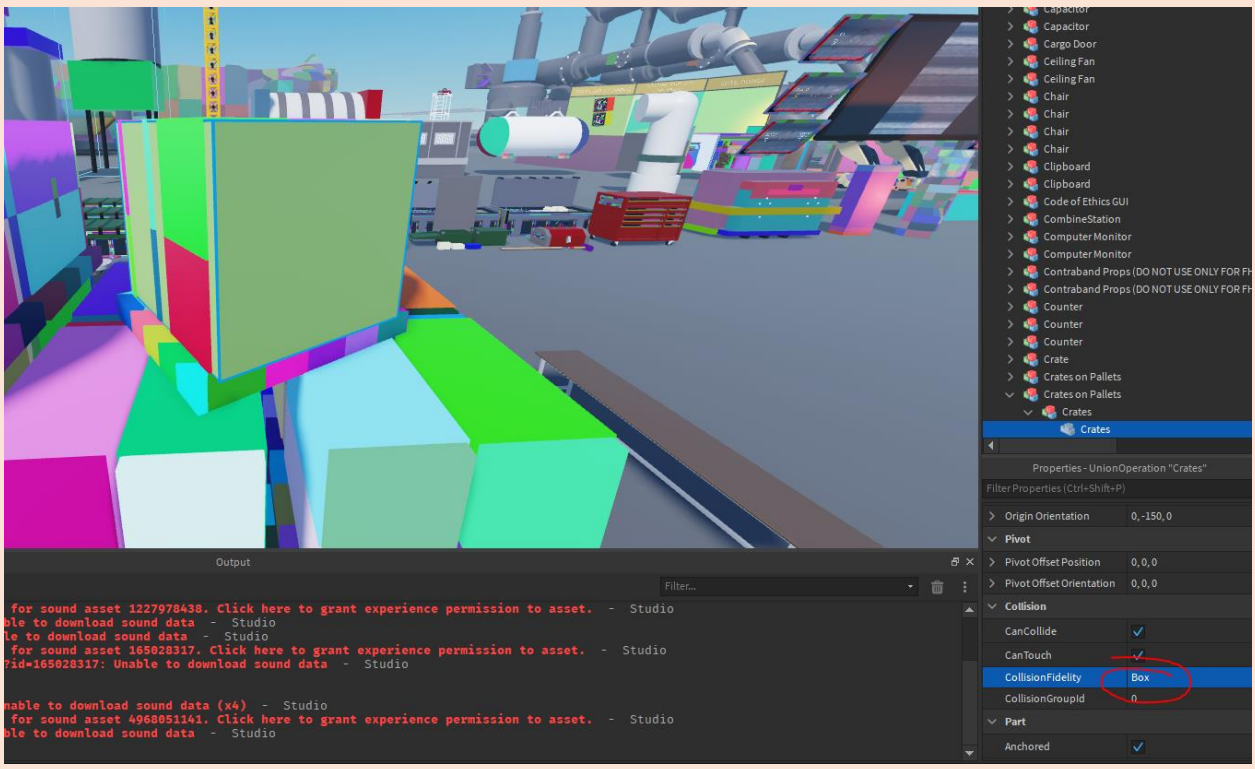
https://www.roblox.com/library/414923656/Mesh-Optimization-Tools



Now to have a look at the examples, these demonstrate the number of quads that are being reduced by using the different options.

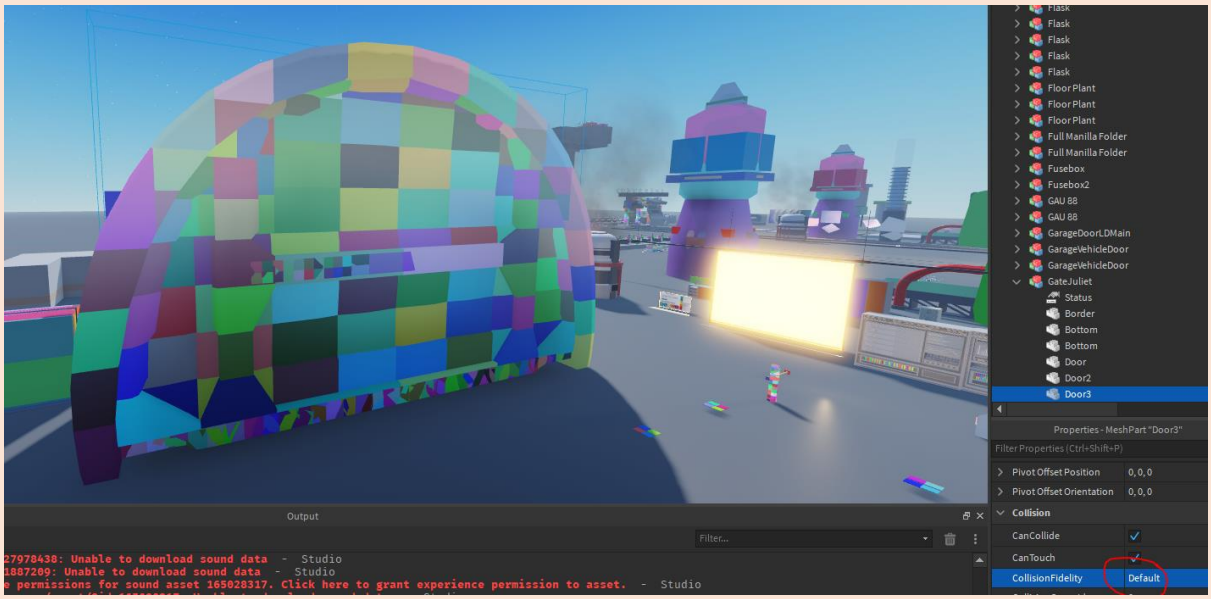
Example 1:

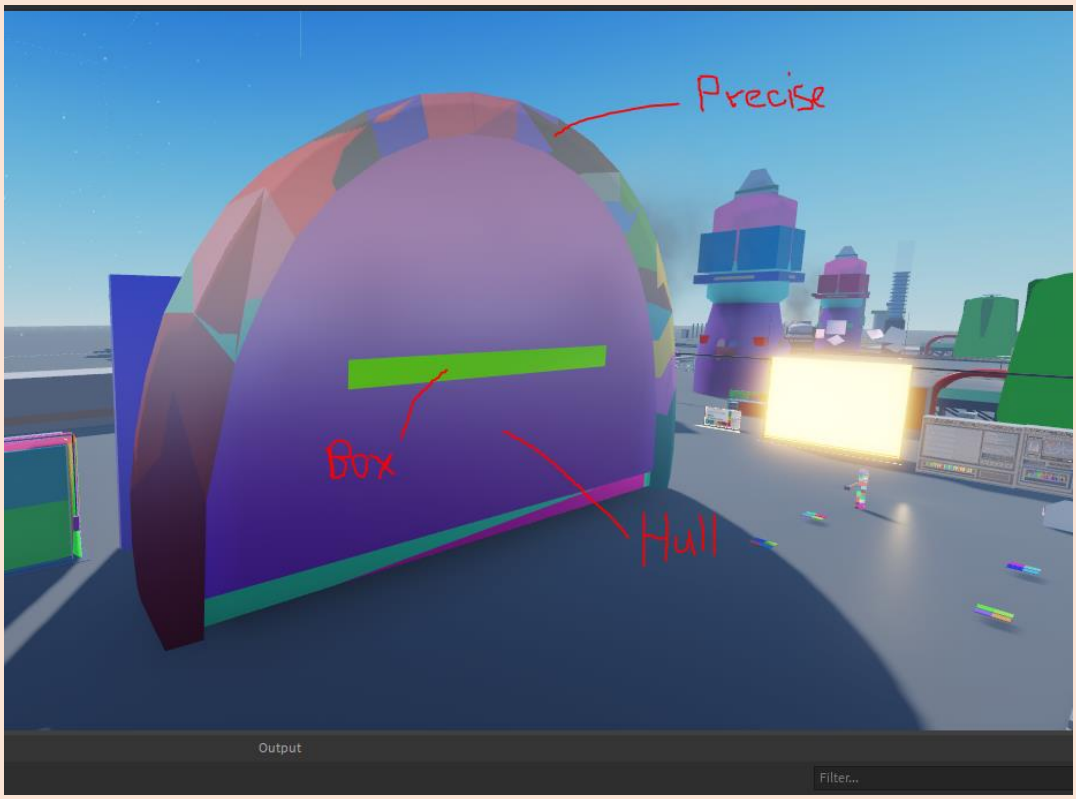




Dramatically reduced the number of quads for that one mesh, also there is no need for the precise collision of that crate mesh, as the box fidelity provides enough.

Example 2: Gate Juliet Example

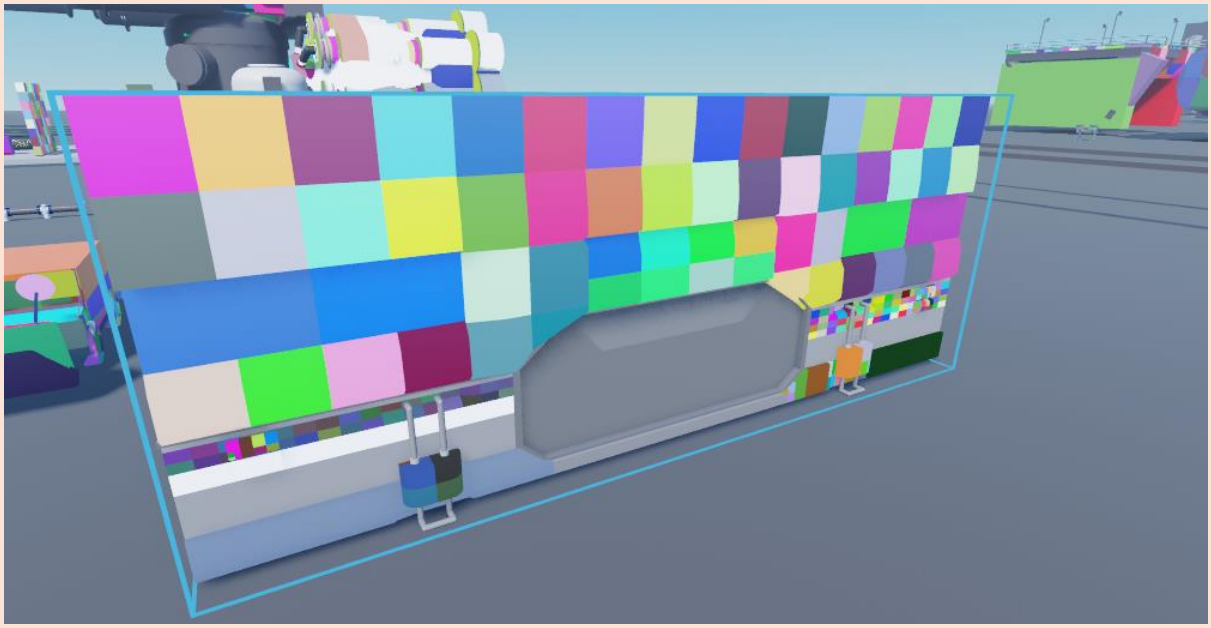


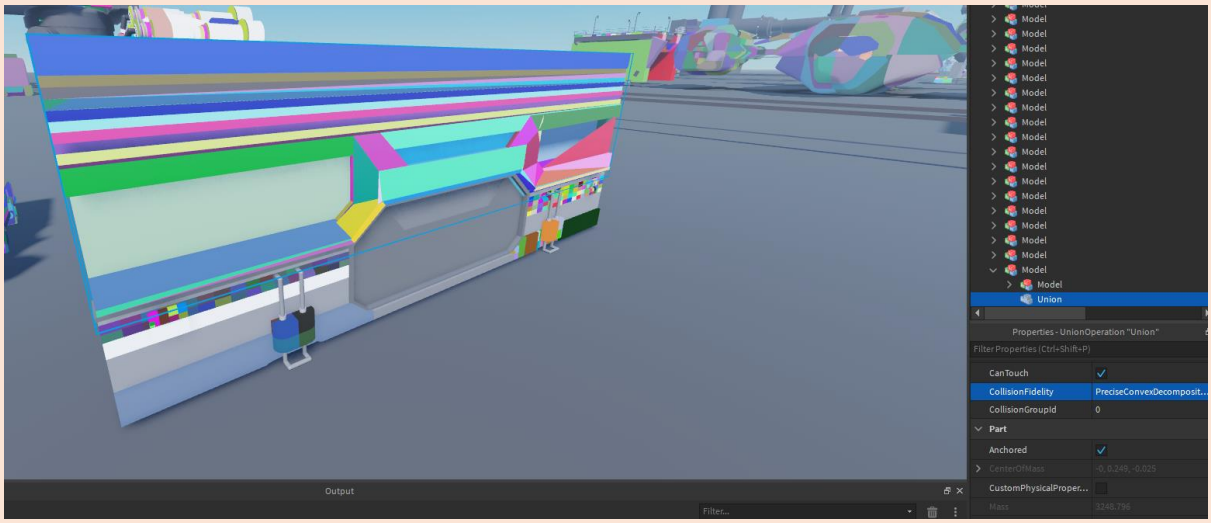


Again, no need for precise collision for some parts so we can change it.

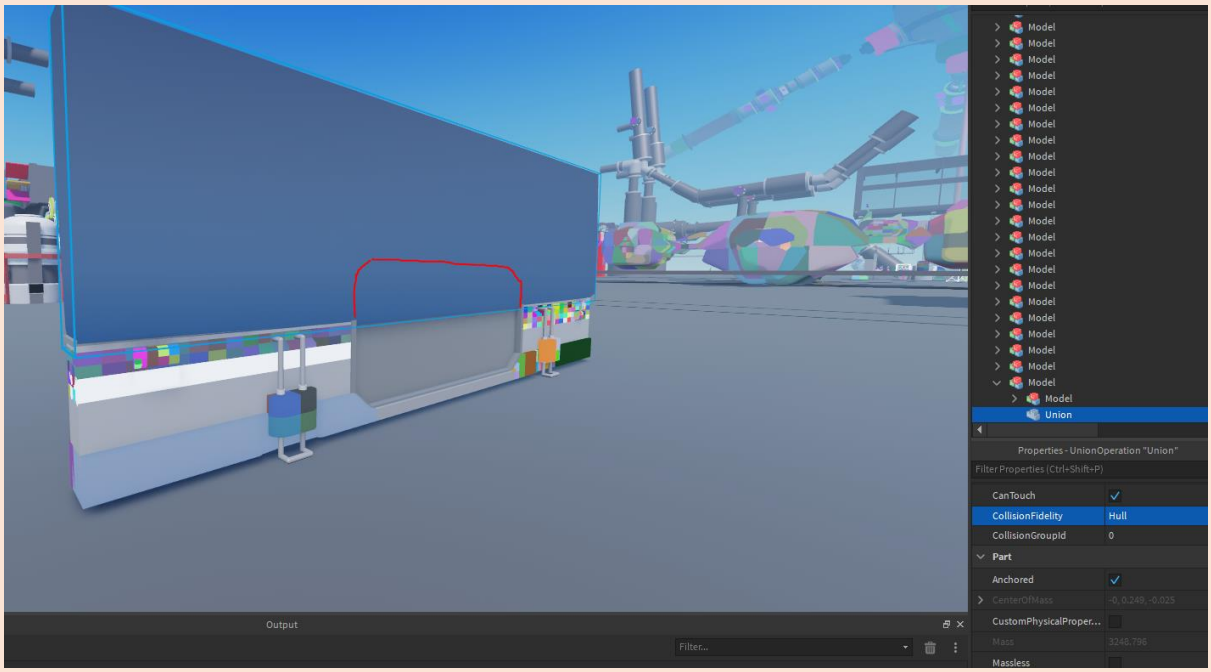
The precise composition may make the mesh look finicky at times due to the **mesh's topology**. That however cannot be fixed unless its taken back into blender and corrected by mesh editing means.

Example 3: Example scenario where default only works.





This is an example case where the **union** cannot support the different options due to the variety of shapes which Roblox poorly put together through the union system. In cases like this, you should report the object in the discord to the builders/modelers so it can be looked at and replaced with a MeshPart if not parts. Just skip over this one and leave it on default.



Using a mixture of precise decomposition, hull and box, you can optimize the mesh in a way which keeps its collision box practically the same, but overall reduces the number of physics quads within the model and each MeshPart itself.

There are some cases you cannot use anything but default like in the picture above and that is completely fine. Just make sure to note it somewhere for a builder as we may need to potentially replace the mesh if that specific meshpart / model is used in lots of places with a corrected and better one.

This is the primary lag producer for unions so keep this in mind when working with unions.

Another important note, for anything that is **non-collidable and non-interactable** (eg: bushes), set their **collision fidelity to box** as that minimizes the total number of quads/triangles that it is taking up in memory, and set their **CanCollide, CanTouch and CanQuery** to false if applicable.

That is all for this section, hopefully you grew a few additional braincells!

Section-6 | PROJECT VERSION BRANCHES

In this foundation, we are to include a version branch system for the site projects. This gives us the different state of the site depending on the three core components of the version numbers. The following format is used: **"v[X].[Y].[Z]".**

The format represents the following:

X – represents the base version of "Alpha", "Beta", or "Release". This is a given numerical integer number of 0, 1, or 2.

Y – represents major branch number (this represents larger content pushes, which is the combination of multiple minor branch pushes). This is an integer starting from 0 for the first major branch in the given base version and increments up.

Z – represents minor branch number (this represents minor content pushes onto the stack for the next major branch push).

Here are examples of the version numbers in a practical sense:

[Alpha] ** initial point of game
0.1.0 = alpha major content push 1
0.1.5 = alpha major content push 1, 5th minor branch patch
0.2.0 = alpha major content push 2
0.3.0 = alpha major content push 3

[Beta] **note: v0.4.0 can be pushed as "v1.0.0" if it's a beta release push
1.0.0 = initial beta release – 'essential beta content'
1.0.5 = '5 minor patch updates'
1.1.0 (1.1) = 'first large content update for beta'
1.2.0 (1.2) = 'second large content update for beta'

[Release] **note: "v1.3.0" can push the next update and become "v2.0.0" as part of the release.
2.0.0 – initial release – 'essential release content, revamp of beta if necessary'