GreasedScotsman's Boneworks Modding Development Series

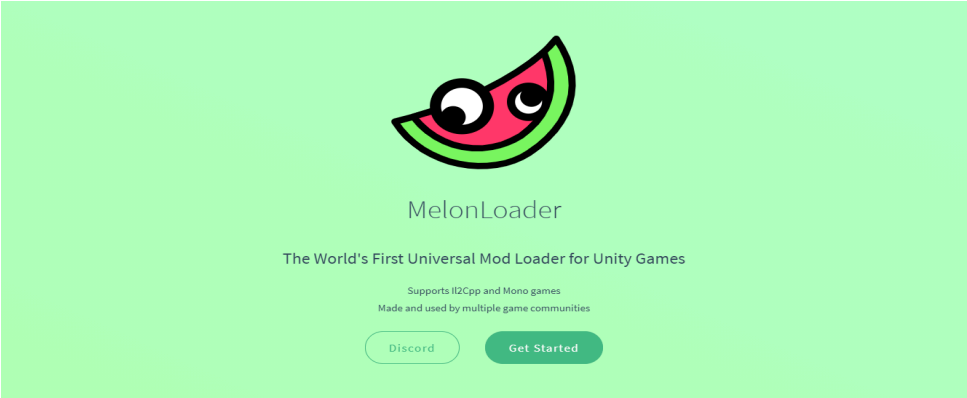
Making Melon Vault and the Modding Tools To Power a Custom Boneworks Campaign

Dedication

The "shoulders of giants" quote definitely applies when I consider my involvement with Boneworks modding. If there is any appreciation for the mods and custom content I have made for Boneworks, please understand that what I created was only possible thanks to the Herculean efforts of several community members, coders and hackers that put their talents to effective use well before I started teaching myself C# or launching Unity for the first time.

In no particular order, I humbly thank:

- **HerpDerpinstine** and the rest of the **Lava Gang**, as MelonLoader was the key that unlocked modding in the first place.
- **Maranara,** who primarily maintains and has vastly expanded the CustomMaps mod and the CustomMapTemplate. The "Hall of Egress" was the first custom map to show me a hint of what was possible, and inspired me to create my own map/campaign. Mara also agreed to lend his art talents to Melon Vault, helping to turn it from a ProBuilder boxy mess into a visual beacon of custom environmental art.
- **TabloidA,** who spent countless hours dedicating himself to my crazy ideas and whose unwavering quality and talent pushed the visual and musical bar for Melon Vault beyond anything I could have imagined.
- **SomeoneSomewhere** for his early contributions with Herp to the CustomMaps mod.
- **Trevtv,** who's ingenuity provided critical breakthroughs in the initial forays of spawning items in maps from the editor and whose patience and understanding helped me grasp the mod creation process, all while I stumbled through teaching myself a new coding language and Unity, itself.
- **Gnonme and Chromium**, for making what I consider to be the backbone of all Boneworks mods, the CustomItemsFramework and BoneworksModdingToolkit (now known as "ModThatIsNotMod").
- Every mod author, weapon designer, model maker, play tester and community member who has provided feedback or contributed to the Boneworks modding scene. *If I've not given credit where it is due, please contact me in Discord so I can correct the record.*



Origins

*This guide contains **SPOILERS** for Melon Vault: Showdown. You have been warned*. The beginning of this document will be "blog-like," as I think the context that describes the state of modding and the people who helped shape the suite of tools we use today are worth knowing as it gives perspective on the community's collective accomplishments. Later sections will become more technical, with screenshots and code snippets that will hopefully help readers understand how the CustomMapInteractions and Custom Map Unity Tools work and how they can be improved or adopted. I arrived relatively late to the Boneworks modding scene. I had paid keen attention to its progress as a long-time modder for Doom, Quake and other titles, but knew once I dove into those waters, it would be all I wanted to do, and also knew my Collectibles and Lore guide contributions to the community would suffer in quality if I shifted my focus too soon.

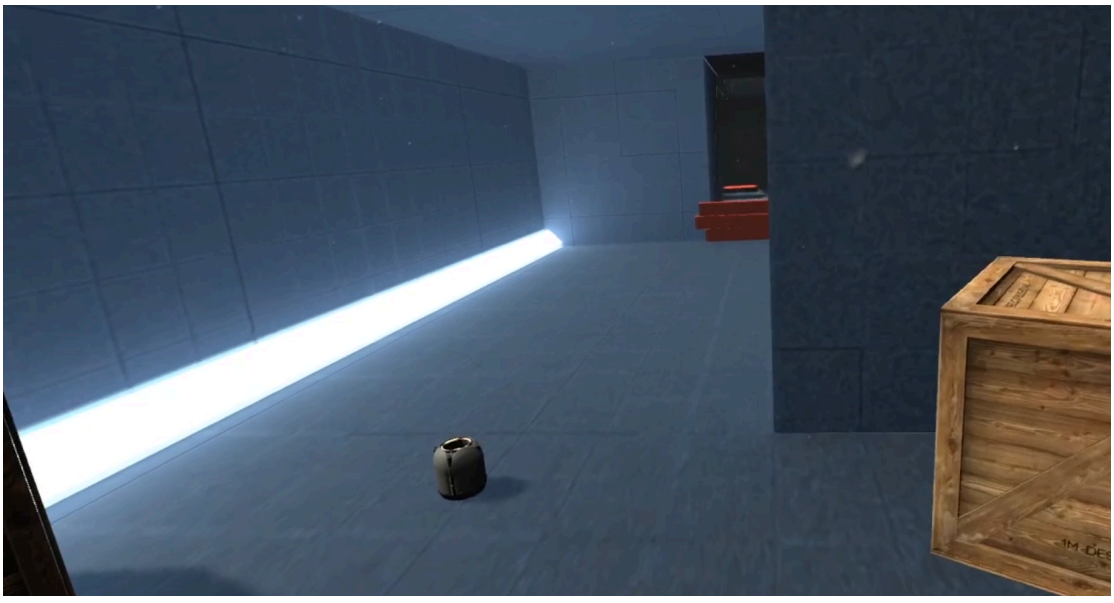
Shortly after the Gun Range/[REDACTED] Chamber update hit in April, 2020, the modding community made strides figuring out many of the essentials: MelonLoader released the same month and experiments with custom weapons and items would follow shortly after. CustomMaps would release in early May and the CustomItemsFramework entered the fray in early June[1]. At the time, items could only be spawned with the Utility Gun. Arbitrary geometry from Unity could be added into a current Boneworks scene, like Blank Box, and this geometry could have materials applied. Maps could be lit with a mixture of baked and real-time lighting, and skyboxes became possible after it was discovered that the lights and fog settings could be removed from the base map. By the time I got involved, a number of mods had already been written that could alter how the game played, like giving the player super strength or complete control over time. These mods were accomplished by building a separate DLL that MelonLoader would recognize, allowing coders to inject custom classes into Boneworks at runtime. This suite of tools opened the door for truly custom content.

When I began dipping my toe into this ocean of possibilities in July of 2020 after finishing various guides for the Zombie Warehouse update, I noticed that the steep learning curve to getting involved in the modding scene often caused artists and potential map designers to give up in frustration. Maranara had made the Custom Map Template, a Unity project meshed with the CustomItems SDK, that allowed one to compile geometry, lighting and materials into a Boneworks-compatible format. He had supplied some basic documentation for it, but there were no tools or documentation that helped designers sidestep a myriad of other technical hurdles. Instead, a very dedicated group of amazing volunteers within the modding community stepped up every single day to help newcomers to the scene get situated, but the inefficiency of this approach showed. While I was very grateful for the kindness and openness evident throughout the community, I knew this wouldn't be sustainable. Further, signs of brain-drain were also at hand just 6 months after the game's release. Some mod authors had moved on to other projects or games, requiring others in the community to try to recreate or maintain their work with each new Boneworks patch.

[1] Thank you to gnonme for clarifying some of the key events in early Boneworks modding history

Setting aside the ease of getting involved, the biggest hurdle at the time to making worthwhile custom content, in my view, was that there was no good way to get custom map information or custom functions from the Unity editor into Boneworks. Custom Monobehaviours placed on GameObjects in Unity were stripped at runtime so that only SLZ scripts and other native Unity scripts used in vanilla Boneworks remained. Another roadblock that hindered designers being able to lean into their talents was that only a few modders had access to a specialized Unity project that included intact classes and variables of SLZ's scripts. No one had access to the actual *meat* of any SLZ script functions. For example, while this special "SLZ Script"-enabled version of the Unity project would allow a modder to see the ConfigurableJoint values on the Dial that adjusts the Spotlight in Blank Box, the CustomLightMachine script functions called LIGHT_ON_OFF and LIGHT_TARGET were completely missing. Thankfully, much of the time, the classes and their variables were enough to get many components working within Unity and in-game. However, there was no way to pass a custom monobehaviour or arbitrary data directly from the Unity editor to the game.

The only way to call a SLZ script function and send it custom data that went beyond manipulating the public variables exposed in Unity was to write a whole separate mod that would inject data at runtime. Yet, custom mods only allowed you to inject custom data and functions that were part of the mod, so there was still a wall of separation between this data injection and any attempt to write scripts in the Unity editor. Figuring out the proper parameters for various API calls and functions was often trial and error, done through inspecting DLLs through dnSpy (or similar) or trudging through cpp2il output of the Assembly methods and trying to glean the requirements and outputs of various functions. The community did not yet have access to some of the DLL and UnityExplorer-type tools that we enjoy today.



An early shot of Maranara's "Hall of Egress" map that inspired me to undertake custom mapping

Still, enough collective hard work and ingenuity had been accomplished by the community that I could very clearly see the foundations of a true modding toolset emerging. My first contribution was simple: I wanted to be able to represent enemy NPC spawn points within Unity and have those enemies show up in-game upon map launch without any extra work on the part of the map designer. When I began working on this, the only way to get enemies into a custom map was to load the map, walk to the point

where you wanted an enemy to spawn, dig through custom in-game menus that drove the EnemySpawners mod, place the spawn point in the world and configure its contents and spawn frequency. Next, the spawner data could be saved into a JSON file using the SceneSaver mod. The next time the map was loaded, the player could choose to load those saved EnemySpawner settings by selecting the appropriate JSON from various menus. This was an extremely clunky process. Still, investigating how this was done gave me insight to how NPCs worked and what parameters were needed to make them function more appropriately.

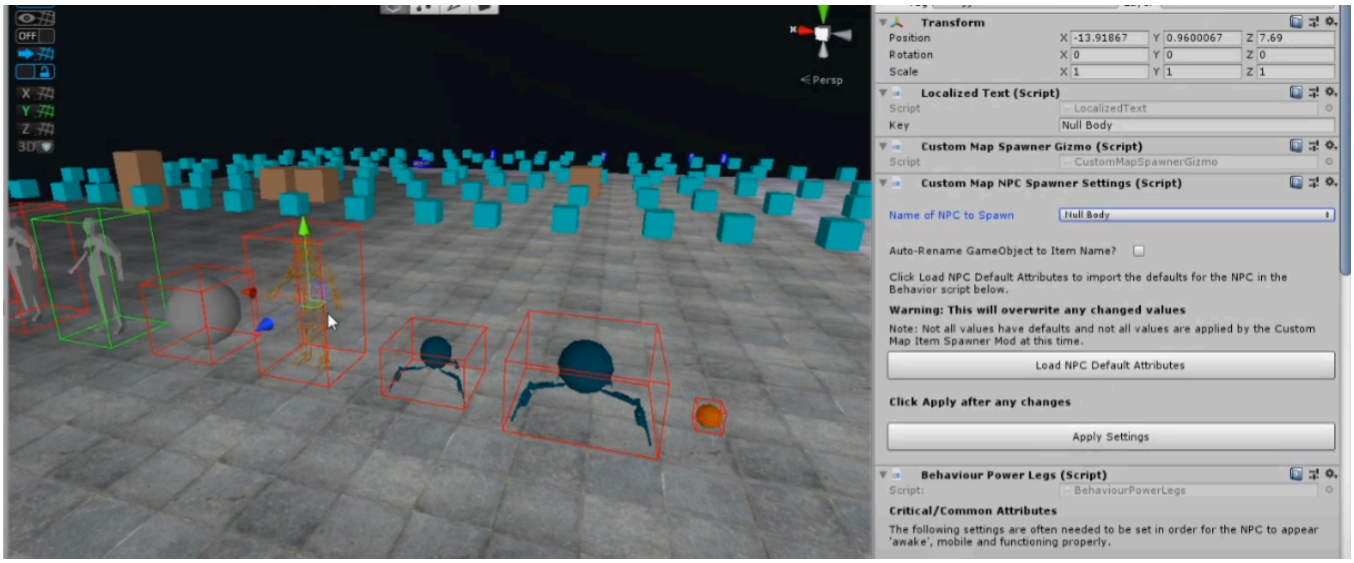
Trevtv made the CustomItemSpawner mod, which would become a game-changer generally and serve as my entrypoint to the modding scene. At runtime, the mod would search a custom map for any GameObject empties that used the name of a NPC (i.e. Crablet) and replace them with an instantiated copy of the corresponding NPC. The enemies were brain dead unless you injured them and the performance was horrendous if you spawned more than a few enemies, but it was solid progress and streamlined getting items and NPCs into maps considerably.

Getting Started

Trevtv's mod paved the way for my first offering, as I knew mappers would be very lost trying to keep track of a bunch of empty GameObject NPC spawns unless they had visual representations of them within the Unity editor. A Blender modeling neophyte (my modeling experience dated to old versions of Maya and 3D Studio Max from the early 2000s), I enlisted Maranara to help me put together a series of low-poly decimated models that would represent each NPC. I learned how to make custom inspectors and gizmos in Unity and used them to replace the empty placeholder GameObjects in the editor with mesh-based gizmos that showed low-poly representations of each NPC. I added representations for their field of view and investigation range and began some preliminary work on color coding gizmos and providing placeholders for all other spawnable game items.

However, I knew usability was key, and instead of having to carefully type in item and NPC names for each GameObject name (which often differed from their well-known names, i.e. FordEarlyExit), mappers needed simple menus so they could quickly select what they wanted to spawn from an in-editor list and didn't have to worry about typos or juggling spreadsheets that mapped common NPC names to their in-game spawn codenames, which the CustomItemSpawner required. I made prefabs for items and NPCs that had all of the custom inspectors and gizmos pre-applied so that adding those entities became a truly drag-and-drop affair.

Behind the scenes, I began writing my first mod, which was an attempt to get custom data directly from Unity into Boneworks at runtime. Trev's mod used the GameObject name as a way to specify what NPC should spawn on map load. It gave me the idea to make the first child of that NPC GameObject an empty whose name would be a long-assed JSON string containing all of the NPC's custom settings like FOV, activation range and whether a Null Body had the thow attack ability. This prototype worked, but it was extremely clunky, ham-fisted and full of map design assumptions.



Early version of GreasedScotsman's Custom Map Unity Tools and Custom Inspectors, August, 2020

Thankfully, Trevtv improved his CustomItemSpawner mod and began using the LocalizedText component as a way to pass in a text string of data. LocalizedText is a simple Unity component that has a public text string variable that saw very little to no use in vanilla Boneworks, but, most importantly, was not stripped out at runtime. Furthermore, Trev and I started playing with all of the parameters on the BehaviourPowerLegs script. As we mapped out each item's use, I tried to document our findings in-editor in the custom Unity Inspectors and gizmos that I packaged as the Custom Map Unity Tools. We found that we could set the values in Unity using a BehaviourPowerLegs component attached to the typical CustomItemSpawner empty. This and the LocalizedText key would serve as placeholders that could be accessed at runtime, allowing us to apply the settings to the instantiated NPC spawned by trev's mod. I was so new to scripting in Unity at this point that I piggy-backed off of the CustomItemSpawner mod to help provide this NPC customization.

This was a great start, but I wanted to go much farther in exploiting the LocalizedText key and started tinkering with other ways I could use it as a conduit to get data directly from the Unity editor into the game at runtime. My first pass at this was to encode and parse settings as JSON strings, but modders at the time thought the requirement to include a JSON parser with my mod would be too cumbersome, so I dropped the idea and wrote a simple homegrown string parsing function that would allow me to pass parameters from Unity into Boneworks consistently.

My NPCSettings project was growing in scope and I was worried I was straying beyond Trev's intent with the CustomItemSpawner mod. I chose to split off my work into its own project with still only a nascent understanding of C#, Unity and modding in general. Beyond customizing NPCs, my next challenge was to provide mappers with the tools to easily drag-and-drop entities that would trigger events and cause the player's actions to impact the level, as custom maps at the time were just spaces to walk in and occasionally shoot enemies that stood still until you damaged them. I turned to learning about Unity Triggers and Events.

Past modding experience had ingrained an appreciation for triggers as ways to help bring life into a game world. I also knew they would solve one of my biggest frustrations with Boneworks combat: Ford could approach almost every combat situation from a safe distance, well before any enemies could

become a threat. Moreover, enemies rarely seemed to react to what you were doing (unless you damaged them directly) and were fairly stagnant until shot or hit.

I did not yet have access to the “SLZ Script”-enabled version of the Boneworks Unity project, so I began developing my own trigger event solution called CustomMapInteractions. I would later facepalm my head through my desk once I did get access to SLZ’s scripts, as a chunk of what I spent hours creating could be achieved by the simple, straight-forward use of the TriggerPlayer or TriggerLasers SLZ scripts. However, by the time I finally gained access to these crucial scripts, I was well into the development of Melon Vault and, more importantly, the system I had put in place actually gave me some crucial advantages and access to details of the triggered events that weren’t readily accessible using the SLZ’s trigger scripts.

One of the largest issues (at the time) was that injected code couldn’t examine any details about the OnTriggerEnter/Exit listings of the Unity or SLZ trigger components and also could not benefit from the native collision Filter settings. This meant that the ways one could use a SLZ trigger would be limited to changing common, global parameters about a GameObject, like whether it was active, if an AudioSource’s clip should play, and so on. In short, SLZ trigger components could be used, but a mod would still be required to attach a custom class to hold and execute any custom OnTriggerEnter/Exit code. However, I worried that using a trigger script commonly used by SLZ might run into compatibility issues or somehow collide with their normal use in-game. To avoid any headaches, I settled on using Unity’s Generic Trigger and the LocalizedText components. CustomMapInteractions would use these as markers to affix my custom CMITrigger class. From there, the first parameter in delimited LocalizedText keys specified which trigger action should be taken and the remaining parameters listed the relevant values that the trigger needed to function, like the magnitude, direction and type of force to apply when launching the player with a Jump Pad trigger.

Super early Melon Vault boss area playtesting, 2nd puzzle

Triggering Violence - [Trigger-Target]

The very first Trigger I imagined any mapper would need would be the full control of how and when enemies engaged with the player. Part of the immersion of any video game arises when enemies react to things that the player is doing. Target triggers would also allow level designers to determine when enemies might engage with the player using distances that make sense for the environment and the area's lines of sight.

At the time, one of the problems with the native "range-only" activation of Boneworks NPCs was that enemies could agro you through walls and other obstacles. This was largely because I hadn't yet figured out Scene Zones, which served as the backbone for enemy AI sensors. For example, in Melon Vault's opening encounter with a Null Body, the player climbed out of a vent and watched as the Null Body killed themselves in red lasers, letting the player know "Lasers = Bad." However, if the second Null enemy relied on range-only activation, they might become active well before the player approached them, since the level designer would have to set the range of the NPC large enough to accommodate it activating at various hallway widths.

```
/// <summary>
/// NPCTarget is fed a NPC GameObject (checks for AIBrain happen beforehand).
///
/// This trigger wlll change the MentalState of the NPC to Agroed and provides the player's TriggerRefProxy for target tracking.
/// </summary>
/// <param name="npcTargeter">The npcTargeter is a NPC that is being triggered.</param>
2 references
private void NPCTarget(GameObject npcTargeter)
{
    if (npcTargeter != null)
    {
        BehaviourBaseNav npcBehaviour = npcTargeter.GetComponent<AIBrain>().behaviour;

        npcBehaviour.SwitchMentalState(BehaviourBaseNav.MentalState.Agroed);
        npcBehaviour.SetAgro(Player.rightHand.triggerRefProxy);
        //MelonLogger.Log("Targeting Player: " + npcTargeter.name);

    }
}
```

CustomMapInteractions: Trigger-Target - NPCTarget() Method

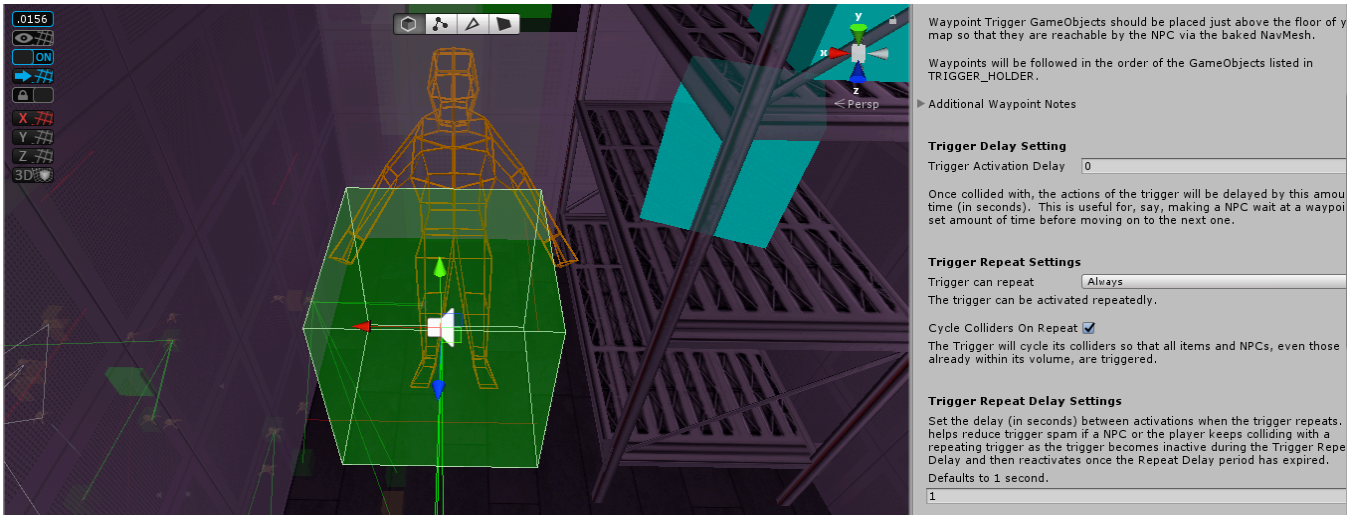
The CustomMapInteractions *Trigger-Target* allowed map designers to place a trigger volume in the map that could be activated by other enemies, the player or any GameObject with a Rigidbody, like a cardboard box. This meant map makers could now structure combat scenarios so that enemies agro'ed to the player in sensible and more immersive ways. They might respond as the player disabled a protective barrier or ambush the player after they passed a certain point. Most importantly, it meant that level designers could lead the player to areas where the geometry was fashioned around combat, with options for cover, sight line height differences, varied angles of attack, and so on.

Timing is Everything

Unity's OnTriggerEnter() event occured pretty much every frame. If a map dev wanted an enemy to attack the player through code, their agro'ed MentalState only needed to be set once. Yet, since our NPCs were really just empty GameObjects with some gizmos, the SLZ trigger scripts could not be used to apply a MentalState to that NPC without modding.
I set out to create a suite of generic trigger events and Inspectors that map designers could arrange and combine to generate chains of events that brought dynamism to their maps. Further, I knew I

wanted much greater control over how, when and what could activate a triggered event. Finer trigger permissions were needed so I could control whether a trigger could be activated by only the player, just an enemy, both, or perhaps just the player's hands but not their feet. Since mods relied on injecting code into the game at runtime and because Boneworks is a IL2CPP game, certain aspects of Unity or Boneworks would remain inaccessible. One of these areas centered on the Filtering system that came with Unity's Generic Trigger interface. Another was that anything listed in the OnTriggerEnterEvent() section was completely hidden at runtime from Boneworks mods.

In the meantime, I turned to controlling how often a trigger could be activated. CustomMapInteractions's *Trigger Repeat Settings* property had three options: *Always*, which would constantly trigger, *Never* ensured the trigger fired only once, and a *Custom Repeat* option that took an integer value that decreased with each trigger activation until reaching zero, causing the trigger to become inert. Along with this repetition functionality, I implemented a repeat delay so that level designers could control the frequency of each trigger's repetition. So, if one wanted a trigger to always repeat, but only fire every 5 seconds, that would be possible.



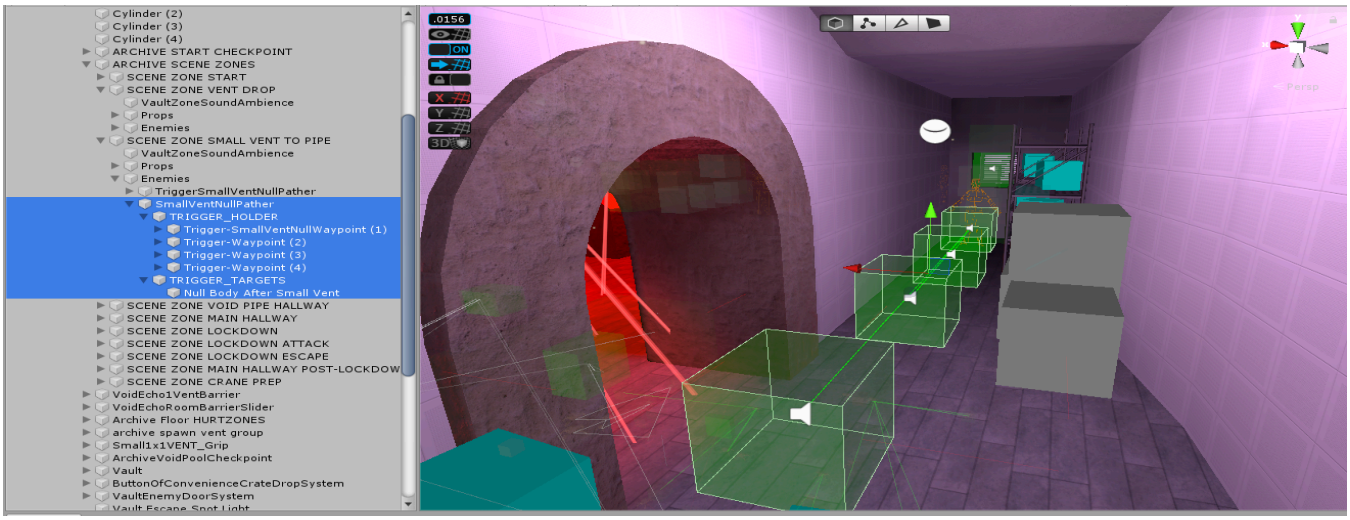
CustomMapInteractions: Trigger-Waypoint Custom Inspector, Custom Map Unity Tools

Whose Child Is This?

Through some experimentation, I think I found that all NPCs, weapons and items, whether Instantiated through Trev's CustomItemSpawner or (eventually) through ZoneSpawners, were children of the root hierarchy in Unity. In other words, any relationship they had with the GameObject hierarchy in Unity or the trigger that they were interacting with was lost as they spawned. This meant that if, say, I wanted an entire group of NPCs to target the player if the player stepped into a Trigger-Target volume, there would be no way at runtime to distinguish if spawned *Null Body [5]* was part of the group of Null Bodies that should be set to Agro by the trigger or if *Null Body [5]* was just standing around a few rooms over and was just waiting for the player's arrival.

In short, I needed a way to structure where spawned enemies appeared in the hierarchy so that I could ensure trigger(s) relevant to them acted up on them. Much later in the modding development process, I would figure out how SceneZones worked and then gain access to the ZoneSpawner's "spawns" List, providing another, much cleaner way to solve this NPC tracking problem.

My solution at the time was to structure each Trigger with TRIGGER_HOLDER and TRIGGER_TARGETS empties. Trigger volumes that were relevant to a NPC would be placed into TRIGGER_HOLDER. TRIGGER_TARGETS would include the list of NPCs or items that would be affected by the activation of any of the trigger volumes in TRIGGER_HOLDER. This allowed me to check if any NPC colliding with a trigger was a child of that Trigger's TRIGGER_TARGETS hierarchy. If not, that NPC's parentage was modified. This meant individual NPCs or groups of NPCs would only be affected by the triggers with which they collided or intended to be activated.



CustomMapInteractions: Trigger-Waypoint Hierarchy, Custom Map Unity Tools

She Told Me to Walk This Way - [Trigger-Waypoint]

Another notable concern I had about Boneworks NPCs was the lack of enemy movement or interaction prior to them noticing the player or taking damage. Enemies that walked, patrolled, conversed or were seen in the distance performing some sort of task before they became aware of the player convey the idea that they were in a functioning game world. Boneworks NPCs did have the ability to Roam, but their movement would be random and only occurred from time to time. Figuring out how to get NPCs patrolling on set paths was my next endeavor.

Thankfully, Boneworks NPCs followed typical Unity Nav Agent navigation around the NavMesh, so creating the CustomMapInteractions *Trigger-Waypoint* was fairly straightforward. This interaction had a few stages: A series of Trigger-Waypoint volumes would be placed down by the level designer and the NPC would be positioned in the first Waypoint so that they would touch it upon spawning into the map. Alternatively, their movement could be triggered by the player, an object or even another enemy at an appropriate time. As the NPC collided with a Waypoint trigger, CustomMapInteractions would feed it the next destination in the chain. Waypoints went through a few iterations, but the release implementation had the Waypoint GameObjects as children of a TRIGGER_HOLDER object. As a NPC hit one of the Trigger-Waypoints, the next sibling Waypoint GameObject would be computed and set as the enemy's destination.

```
/// <summary>
/// Waypoint GameObjects are siglings of each other and have a TRIGGER_HOLDER empty GameObject parent.
/// As a NPC (AIBrain) hits a Trigger-Waypoint, the next sibling is computed and fed as the next destination,
/// calling MoveToTarget for the triggered NPC.
/// </summary>
1 reference
private void GetWaypoint()
{
    if (triggerColliderOnly == true && _collidedWithTriggerObj.GetComponentInParent<AIBrain>())...
    else
    {
        // NPCs are children of TRIGGER_TARGETS.  Triggers are children in TRIGGER_HOLDER, which is a sibling game object to TRIGGER_TARGETS.
        foreach (var triggeredNPC in transform.parent.parent.Find("TRIGGER_TARGETS").GetComponentsInChildren<AIBrain>())
        {
            int nextSibling = transform.GetSiblingIndex() + 1;

            if (nextSibling < transform.parent.childCount)
            {
                npcWaypoint = transform.parent.GetChild(nextSibling);
            }
            else
            {
                npcWaypoint = gameObject.transform.parent.GetChild(0);
            }

            if (npcWaypoint)
            {
                //MelonLogger.Log("NPC "+ triggeredNPC.name+ " was triggered by: " + triggeredNPC.transform.parent.parent.name+ " with MoveToTarget: "+ npcWaypoint.name);
                MoveToTarget(triggeredNPC.gameObject);
            }
            else
            {
                MelonLogger.Log("WARNING: Could not find destination after hitting Trigger: " + triggeredNPC.transform.parent.name);
            }
        }
    }
}
```

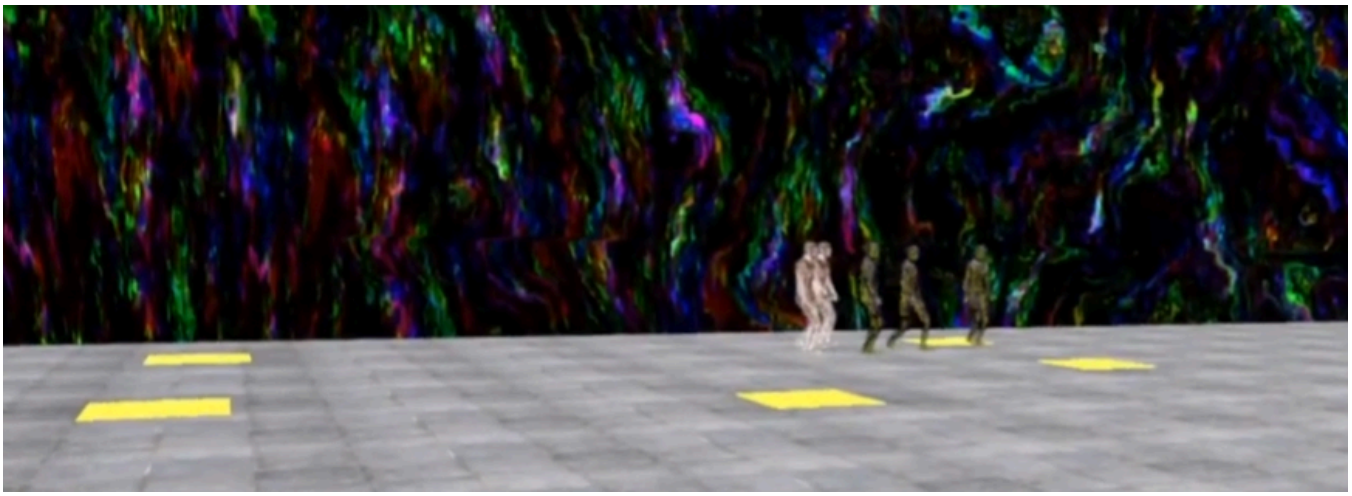
I doubt anyone reads these caption things. I shall reclaim time and space.

With the Waypoint route handled, the last step was to actually get the NPC moving along its path. Thankfully, an unfinished AI state called *Investigate* that apparently wasn't used in vanilla Boneworks helps sidestep an interesting bug. Without being set to Investigate, NPCs would path to each waypoint just fine. However, if they spotted the player or took damage while in the Roam or Rest MentalStates, they would hilariously play their Agroed sound, but then continue along their path as if they had never seen the player. This wasn't the intended behavior, and thankfully, the Investigate state correctly caused NPCs to leave their Waypoint path and attack the player if they took damage or if the player entered their Investigation Range. CMI's MoveToTarget() function simply forced a NPC's Investigate MentalState and fed the native BW navigation SetPath() method the Vector3 position determined by CMI's GetWaypoint() method.

```
/// <summary>
/// Using an existing npcWaypoint determined by GetWaypoint(), MoveToTarget sets the NPC's MentalState to Investigation
/// This MentalState allows NPCs to leave their waypoint pathing if they take damage or if the player or other hostile entity
/// gets within their Investigation Range.
///
/// SetPath sends the NPC to the specified Vector3 position, in this case the next sibling of the Waypoint chain that was
/// determined in GetWaypoint().
/// </summary>
/// <param name="npcTargeter"></param>
2 references
private void MoveToTarget(GameObject npcTargeter)
{

    if (npcTargeter.GetComponent<AIBrain>())
    {
        npcTargeter.GetComponent<AIBrain>().behaviour.SwitchMentalState(BehaviourBaseNav.MentalState.Investigate);
        npcTargeter.GetComponent<AIBrain>().behaviour.SetPath(npcWaypoint.position);
        //MelonLogger.Log("NPC " + npcTargeter.name + " is moving to: " + npcWaypoint.gameObject.name);
    }
    else
    {
        MelonLogger.Log("Brain not found for " + npcTargeter.name);
    }
}
```

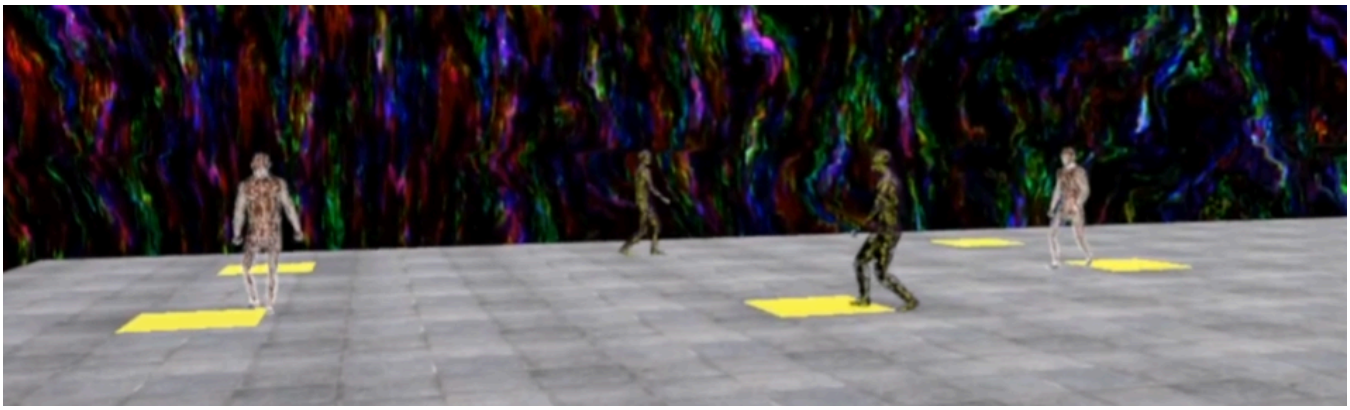
Not All At Once

While it was useful to finally have NPCs that could path a patrol and engage with the player in a more interesting manner, a problem was introduced: If more than one NPC was to follow along the same set of waypoints and were all members of TRIGGER_TARGETS, each time any NPC hit any of the Waypoint triggers, all NPCs got that same signal and moved to the same position. This meant that eventually, all NPCs would be crowded together and moving in unison to the same Waypoint.



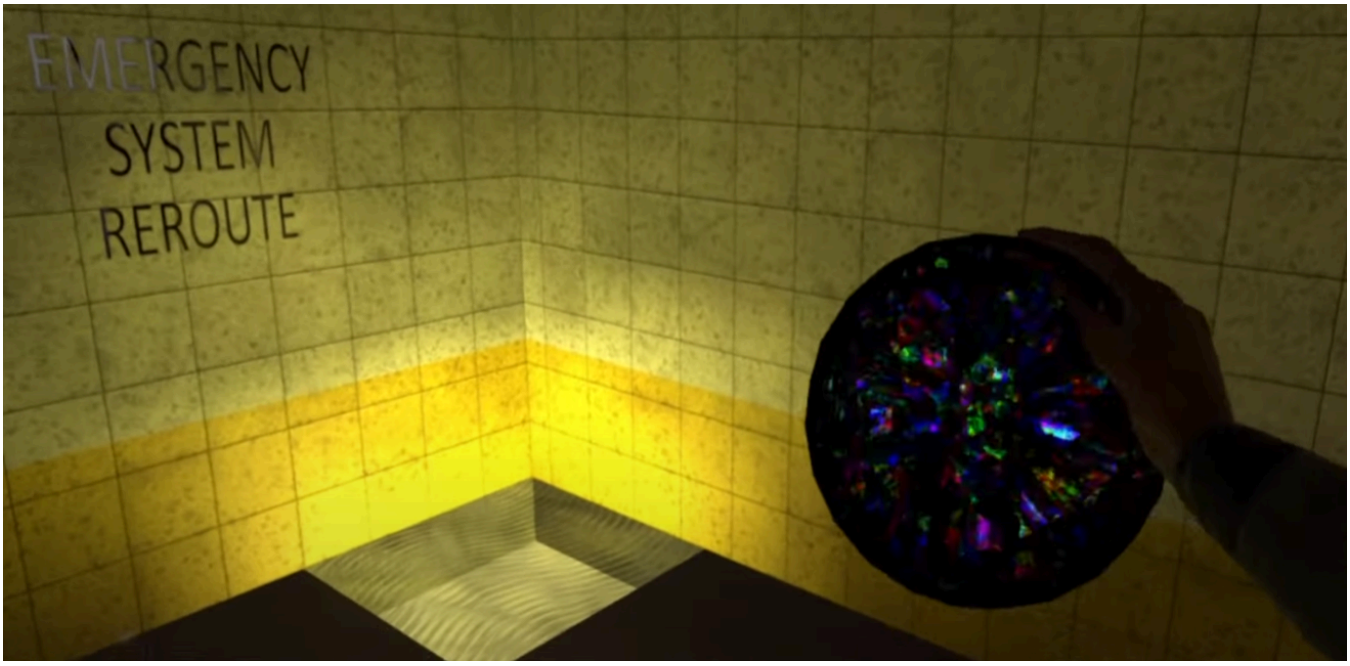
To solve this and still keep the improvements that ensured NPCs and objects could be tracked and were only impacted by their relevant trigger volumes, I created a “colliderOnly” property for all CMI triggers. If this option was checked in Unity, only the specific NPC that touched the trigger volume would get activated. This meant that you could have a group of Null Bodies using the same Waypoint loop, but they could all be pathing on their own segment of the waypoint chain and their SetPath() calls would be their own.

I eventually added a *Trigger Delay* property, which would simply Invoke the Trigger’s action some configurable time later rather than instantly. This worked nicely with Waypoints, as it gave the impression that Nulls might stop and survey the area before continuing along their patrols. One of the last additions to basic trigger functionality was the Percent Chance to Trigger parameter that could be used to provide some randomness and variety to any trigger activation.



Scotsman's Log, Vault Date 20200921

After Waypoints and Target triggers, I turned to making grabbable, holster-able, pausable Audio Logs that I named Void Echoes. Melon Vault would tell most of its story with the visuals, events and action that the player experienced throughout, but the details that tried to explain *why* the player was infiltrating this huge building or *how* they came to be chased down by giant electrified balls of death as they hunted for power cells in the dark… *that* story would be conveyed by these Void Echoes. Without access to any SLZ scripts yet, I was unaware of the GripEvents component, so I fashioned my own trigger-based system that would respond to the player and check to see if they were touching an Echo. If so, then audio playback would begin. If the player grabbed the Echo and held down the PrimaryInteractionButton, a timer would start and, if its time limit was reached, the audio playback would pause. It was a simple system and served its purpose at the time, though later (unreleased) versions of CustomMapInteractions revamp this feature.



"I plan to kill the player… repeatedly." --GreasedScotsman [Trigger-Checkpoint]

Melon Vault would be huge compared to the myriad of custom maps that were available on BoneTome.com at the time. Checkpoints were needed. Thankfully, Maranara had already done all of the hard work for this trigger, which simply moves the spawn point to a new position and lets the CustomMaps player respawn code do the heavy lifting. These spawn point changes do not survive a close of the game session, but for Melon Vault, I could now construct deadly scenarios without worrying that the player would be frustrated by being forced all the way back to the start of the map upon death.

```
if (spawnCheckpoint)
{
    spawnCheckpoint.transform.position = gameObject.transform.position;
}
```

Beam Me Up, Mara [Trigger-TeleportObjectToTarget and Trigger-TeleportPlayerToTarget]

While not shown in the first “Melon Vault Reveal” video, I had the map’s opening “Swing from the rooftops” encounter mapped out in a test area. However, because the player could fail to let go at the right time and fall to their deaths, I needed a way to reset the moving pieces so the leap could be attempted again. Teleporting objects with triggers seemed like a simple solution to this problem.

```
/// <summary>
/// Assuming the teleportTarget (often an empty destination GameObject) and teleportSource
/// (the GameObject to be teleported) exist, simply set the position of the teleportSource
/// object to the teleportTarget destination.  Use the cached references to any rigidbodies
/// of the teleportSource object, set them to zero velocity and angular velocity to prevent
/// the teleported object from flying about, knocking into other items or, if the speed is
/// high enough at the time of teleportation, passing through floor colliders and the like.
///
/// Optionally, set the parent of the teleport source to the teleportTarget if desired.
/// </summary>
1 reference
private void TeleportObjectToTarget()
{
    if (teleportTarget && teleportSource)
    {
        Rigidbody[] srcRBs = teleportSource.GetComponentsInChildren<Rigidbody>();

        teleportSource.transform.position = teleportTarget.gameObject.transform.position;
        teleportSource.transform.rotation = teleportTarget.gameObject.transform.rotation;

        foreach (Rigidbody srcRB in srcRBs)
        {
            srcRB.velocity = Vector3.zero;
            srcRB.angularVelocity = Vector3.zero;
        }

        if (parentTeleportedObjToTarg == true)
        {
            teleportSource.transform.parent = teleportTarget.transform;
        }
    }
    else
    {
        MapLogger.Log("Couldn't find teleportSource or teleportTarget for " + gameObject.name);
    }
}
```

Rifling through the respawn code in CustomMaps mod, I learned how Mara teleported the player from Blank Box to the custom map’s spawn point. I yoinked the key parts of the process into a CMI trigger.

```
3 references
private void TeleportSourceToDestination()
{

    Rigidbody[] srcRBs = sourceObject.GetComponentsInChildren<Rigidbody>();

    if (sourceObject.name == "PLAYER_HOLDER")
    {
        //playerChain = GameObject.Find("[RigManager (Default Brett)]");
        MelonLogger.Log("TELEPORT: Player to Destination: " + targetObject.name);
        Vector3 vector = playerChain.transform.position - playerBody.transform.position;
        playerChain.transform.position = targetObject.transform.position + vector;
    }
    else
    {
        if (sourceObject.name == "WEAPON_HOLDER")
        {
            if (sourceObject.GetComponentInChildren<WeaponSlot>())
            {
                sourceObject.transform.position = targetObject.gameObject.transform.position;
                sourceObject.transform.rotation = targetObject.gameObject.transform.rotation;
            }
        }
        else if (sourceObject.gameObject.name == "AMMODISPENSER_HOLDER")
        {
```

This crude but effective method of yanking the entire player chain through the map has since been replaced by the far more elegant and native BW Teleport() function that ModThatIsNotMod's easy access to the RigManager exposes:

```
/// <summary>
/// Using the ModThatIsNotMod methods for confirming player existence, resetting
/// any wayward hand motion and performing teleportation, send the player to the
/// teleportTarget.
/// </summary>
1 reference
private void TeleportPlayerToTarget()
{
    if (teleportTarget)
    {
        MapLogger.Log("TELEPORT: Player to Destination: " + teleportTarget.name);
        rigManager.physicsRig.ResetHands(Handedness.BOTH);
        rigManager.Teleport(teleportTarget.position, true);
    }
    else
    {
        MapLogger.Log("Couldn't find teleportTarget for " + gameObject.name);
    }
}
```

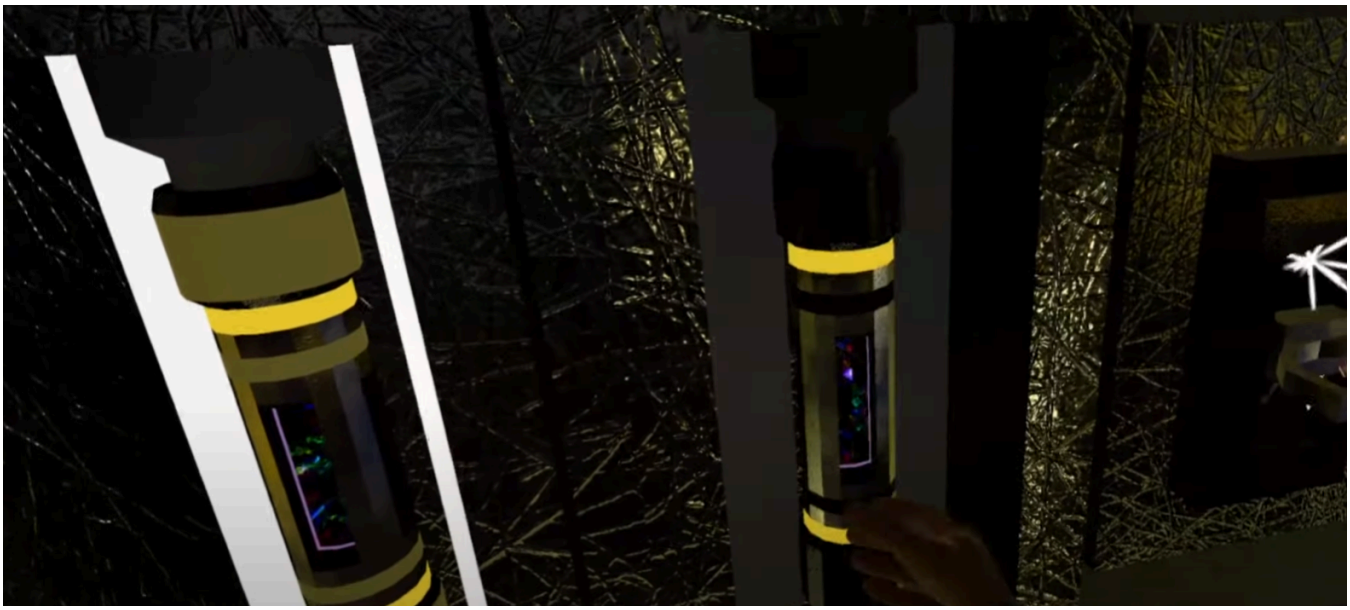
However, trigger-based teleportation meant that I could also provide a way for map designers to reference and move items like the Ammo Dispenser, Health Machine and JukeBox anywhere within their custom maps. I handled this by instructing mappers to make GO empty placeholders with specific names (i.e. AMMODISPENSER_PLACEHOLDER) and swapped those out with the actual Ammo Dispenser from Blank Box upon trigger activation. Mara would eventually cache and provide references to these objects in later versions of Custom Maps, streamlining this process further.

Show Me Some Identification [Trigger-RigidbodyTarget and Trigger-RigidbodySocket]

Melon Vault needed to have doors and barriers for which the player needed an item to open. However, I did not want to just borrow the Boneworks key system. Instead, I liked the physicality of an ID keycard swipe, and set out to make a trigger that required a unique object for activation.

My first pass at this, Trigger-RigidbodyTarget, thankfully has since hit the cutting room floor as it was overly convoluted. It involved a sequence of overlapping triggers, and, while functional, I realized that I could simply check the uniqueness of an object before handling the triggered action and bail early if the incorrect object was supplied.

Power Cells in Melon Vault used the Trigger-RigidbodySocket, which was my own take on the Boneworks battery system. I wanted players to have to hunt for unique power cells during the stealth section of the map, but Boneworks batteries were both interchangeable and removable. I needed items that would only work in their specific stealth section and only activate their intended receptacle.



The CMI scripts included in the upcoming CMaps Template release have unique object activation for almost all triggers and don't require any of these RigidbodyTarget/Socket shenanigans. Hilariously, I would later learn that SLZ's scripts included a component called "TriggerLasers" that already had a unique object parameter built into it. Cue the facepalm through the desk. Reinvent the wheel, indeed.

Some Dead Ends Lead to Amazing Places

I am a huge fan of stealth action games. Dig deep enough into the history of my YouTube channel, and you will find my "Zero Takedown Ghost" full walkthrough of Deus Ex Human Revolution, meaning the game was completed without knocking out or killing anyone (except for when the game forced you to do so >:( ) and with no trace left of my passing. Lost to the memory hole are my even older ghost-style playthroughs of the Splinter Cell series.

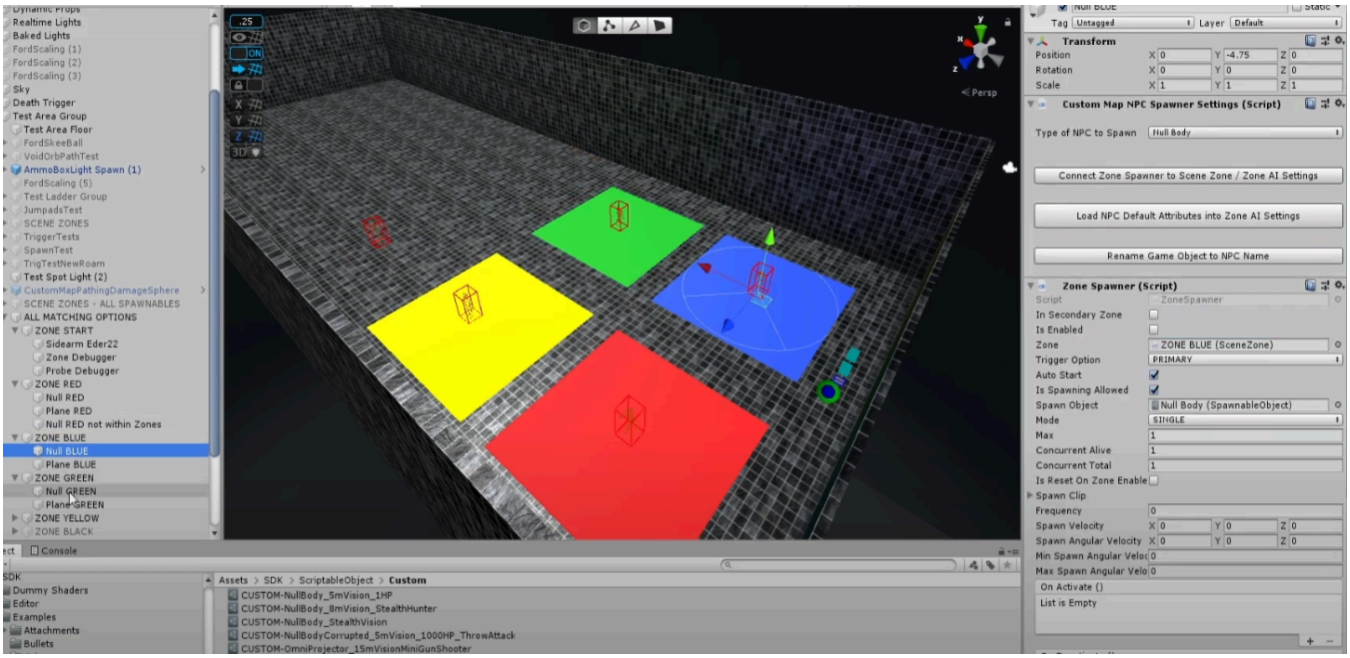
I was curious (and hellbent) on seeing if I could introduce serviceable stealth gameplay to Boneworks. To that end, I spent about 3 weeks trying to ham-fist modifications to Boneworks NPCs to adjust their detection behaviour, field of vision, movement speed and agro routines. While I had some success, the solution was not truly viable and the endeavor ended with very little to show for the time spent. Still, I learned in some detail how Boneworks NPCs worked and what many of their limitations were.

Two additional extremely critical things happened along the way. First, Maranara had, in passing, talked to me about something called SceneZones but he hadn't quite figured them out. Modifications had been made to a beta version of CustomMaps so that they were mostly functional and he knew they were the way SLZ spawned their enemies. Most importantly, they provided huge performance gains compared to the current CustomItemSpawner method of just Instantiating the enemy in a brain-dead, deaf state. The goal was to figure out how to get them working to the point that map developers could just drop Scene Zone and enemy prefabs into the map as desired. I had toyed with directly accessing the collection of Pooled enemies through script, but knew following the "SLZ way" would be ideal.

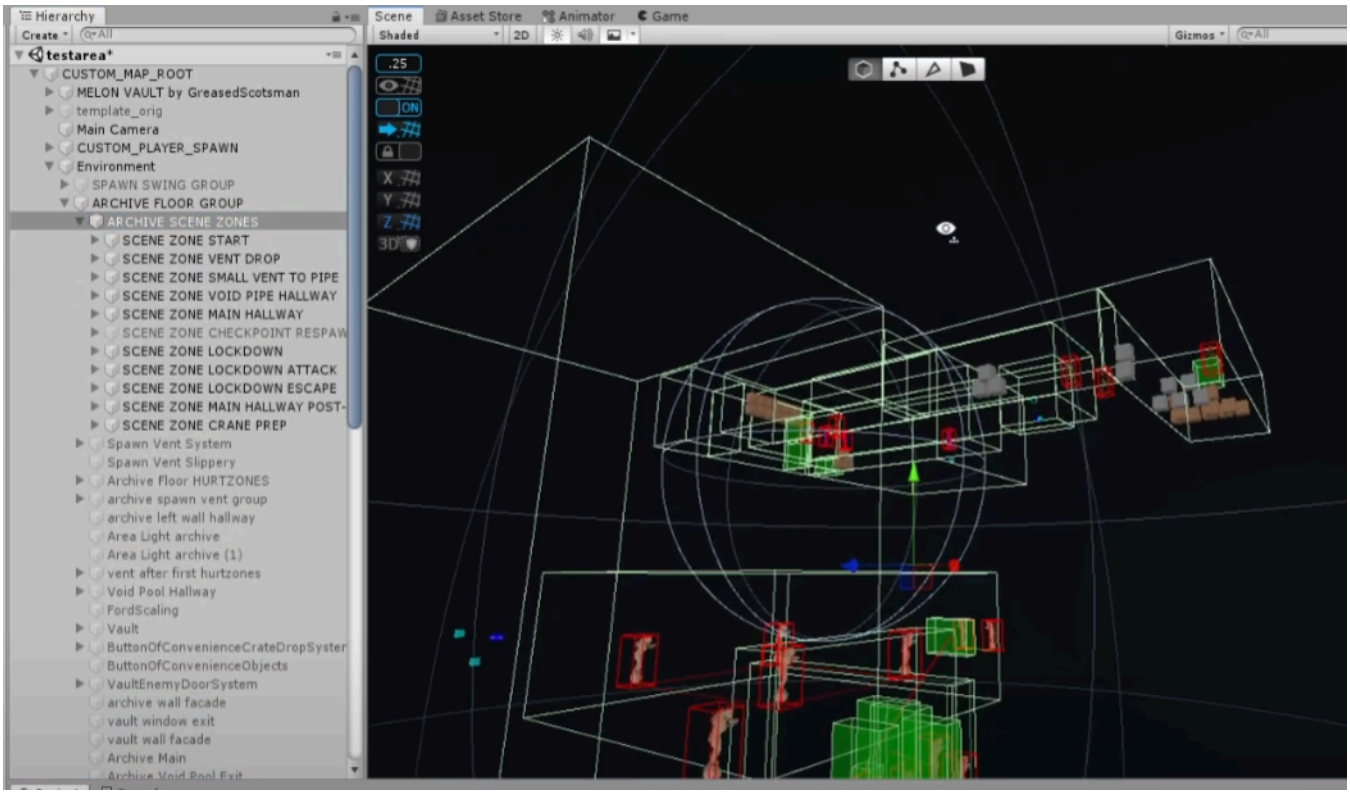
Second, a SLZ Script-enabled version of the Unity project became available to me for the first time. In retrospect, I view most of my modding endeavors up to that point as just blindly stabbing in the dark, potentially reinventing wheels and just trying to find solutions to problems as I encountered them without any real understanding of how the underlying systems of the game worked. Gaining access to the scripts changed much, and most importantly, gave me insight into how SceneZones worked.

Setting the Scene(Zone) [Trigger-EnableZoneSpawner, Trigger-SetZoneSpawner, Trigger-ZoneSpawnerPerformanceSettings, Trigger-ZoneSpawnerCombatSettings, Trigger-HideSceneZoneItemsInChildren]

As my understanding solidified, I realized how important these changes were, and I decided to completely scrap or refactor nearly all of the custom Unity inspectors I had made, despite them being a huge chunk of my CustomMapUnityTools project. One of the biggest realizations was that the contents of the prefabs within the ScriptableObjects for NPCs and items in the Unity editor were irrelevant. I could use fake versions of prefabs that had no dependency ties, no artwork, scripts, sounds, etc. It seemed (guesswork) the game only cared about the ScriptableObject's unique ID at runtime and would use that to reference the actual NPC/weapon prefab and spawn the relevant enemy or item. With this, SceneZones not only functioned, but they worked in a way that we could release to the public without worry of infringing on SLZ's intellectual property.



Frantically, I worked to revamp the suite of custom inspectors in CustomMapUnityTools to accommodate Scene Zones, made prefabs for items, NPCs and SceneZones themselves so getting them functional with all of the proper components would be a drag-and-drop affair. I also made some helper scripts that would update the ZoneAISettings fields depending on which NPC was chosen by the CMUnityTools menus and an Auto-Connection script that would ensure any children ZoneItems or ZoneSpawners under a SceneZone GameObject were correctly tied to the Zone. In testing, I had found that any gaps in the array of ZoneItems would cause the entire SceneZone (and sometimes all subsequent Zones in the map) to malfunction. I also tinkered with SceneZone linking and, at the end of September 2020, mentioned a portion of what I learned in my “Boneworks Modding - Major Update, Melon Vault Sneak Peek" video. I chuckle in hindsight considering the line in that video that Melon Vault would contain “at least 30-40 minutes of gameplay,” since playtesters often clocked in the final version of the map at 2 to 3 hours. In the following weeks, I completed my rework of the CustomMapUnityTools and released two tutorial videos, one that focused on Getting Started with custom map-making and the other that detailed everything I had learned about SceneZones. Maranara and I worked tirelessly to get all these changes into CustomMapTemplate 2.0 and pushed it and a CustomMapInteractions release out to the public.



Over time, I made several more tools that gave map designers control over ZoneSpawners. The functionality of many of these triggers could also be achieved using SLZ’s PlayerTrigger or TriggerLasers scripts, as ZoneSpawners often simply required one-time activation and didn’t need extra custom code to operate. This meant the Unity editor’s native OnTriggerEnter events interface would suffice. However, I developed my own set of tools in order to have fuller control over trigger repetitions and sequences of trigger activations for Melon Vault. I also wanted to provide options for players to dynamically control the frequency or activation of certain SceneZones based on their in-game actions. Trigger-ZoneSpawnerCombatSettings would allow one to adjust the difficulty of combat and

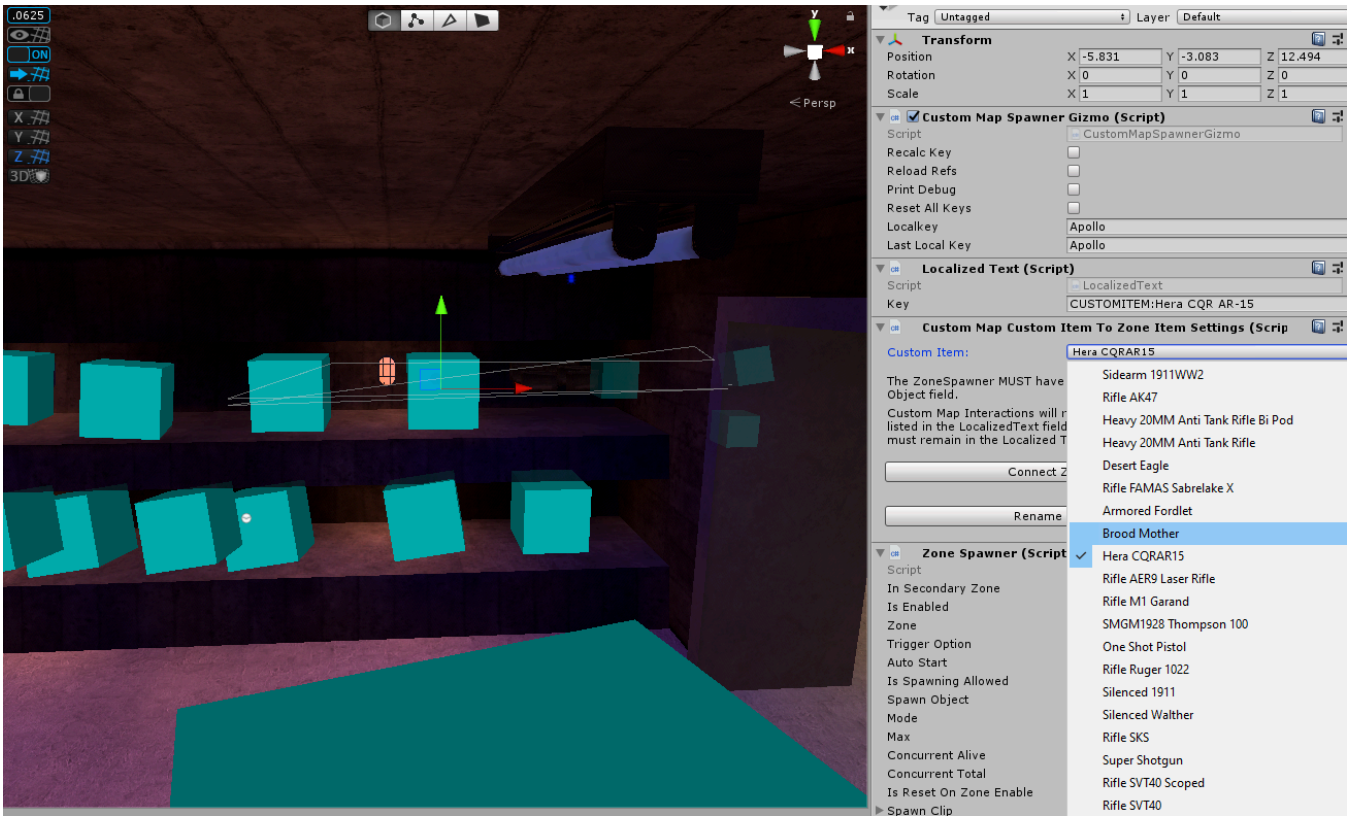
Trigger-ZoneSpawnerPerformanceSettings could provide some performance improvements for lower-end systems by adjusting Zones that spawned several NPCs in non-combat scenarios.

As I fleshed out Melon Vault's Armory room, I wanted to feature several custom weapons made by community members and therefore needed to be able to spawn them using SceneZones. However, this posed a problem, because the only method of spawning custom items involved the CustomItemsFramework dynamically generating a custom item's ScriptableObject on launch of the game and using the Utility Gun to place the object in the world.

To get around this, I used faithful Apollo as a stand-in ZoneItem so that the ZoneSpawner component would function properly (Unity or BW would complain about ZS's that had missing info), then wrote a special SpawnPoolItem function that specifically looked for a CUSTOMITEM:*customItemName* parameter in the LocalizedText key. Using this, I could then search the pooled objects at runtime during MapLoad for the customItemName item and, if a match was found, replace the Apollo SpawnableObject with the custom item's dynamically-generated SpawnableObject.

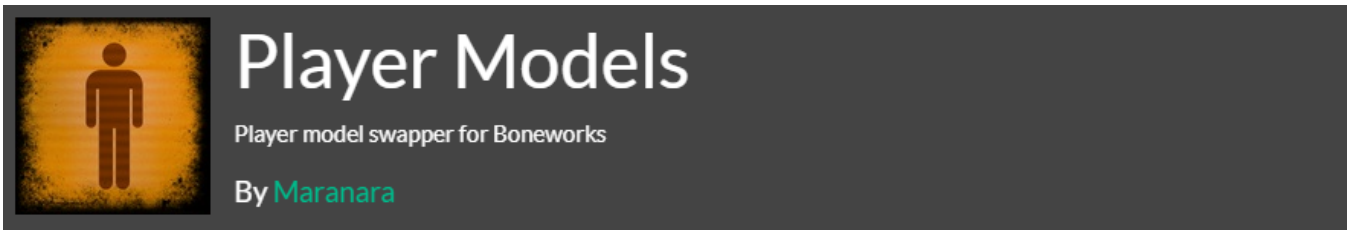
```
1 reference
public static void SpawnPoolItem(LocalizedText localizedText)
{
    string customItemName = localizedText.key;
    bool itemAllowed = true;


    if (customItemName != null && customItemName != "")
    {

        if (customItemName.Contains("CUSTOMITEM:"))
        {
            customItemName = customItemName.Replace("CUSTOMITEM:", "");
        }

        foreach (var blacklistItem in Lists.blacklistedItems)
        {
            if (customItemName.ToLower() == blacklistItem.ToLower())
                itemAllowed = false;
        }

        if (itemAllowed)
        {
            //MelonLogger.Log("Pooled Item Allowed: " + customItemName);

            try
            {
                bool customItemSpawned = false;

                foreach (string titleText in PoolManager._registeredSpawnableObjects.Keys)
                {
                    if (PoolManager._registeredSpawnableObjects[titleText].title.ToString() == customItemName)
                    {
                        SpawnableObject spawnableItem = PoolManager._registeredSpawnableObjects[titleText];

                        ZoneSpawner itemZoneSpawner = localizedText.GetComponent<ZoneSpawner>();
                        if (spawnableItem != null && spawnableItem.title == customItemName)
                        {
                            if (itemZoneSpawner.spawns.Count > 0)
                            {
                                foreach (GameObject spawnedItem in itemZoneSpawner.spawns)
                                {
                                    if (spawnedItem.name.Contains(customItemName))
                                    {
                                        customItemSpawned = true;
                                    }
                                    else
```

I then added Unity editor tools that, given the proper path to the map designer's CustomItems *.melon* files, would populate a drop-down menu for use in the CustomMapCustomItemToZoneItem prefab I created for ZoneSpawning custom items. This meant mappers could select custom items easily from a list rather than try to type (or sometimes outright guess) the name of a custom item, as the internal name of an item and its filename were often not the same.



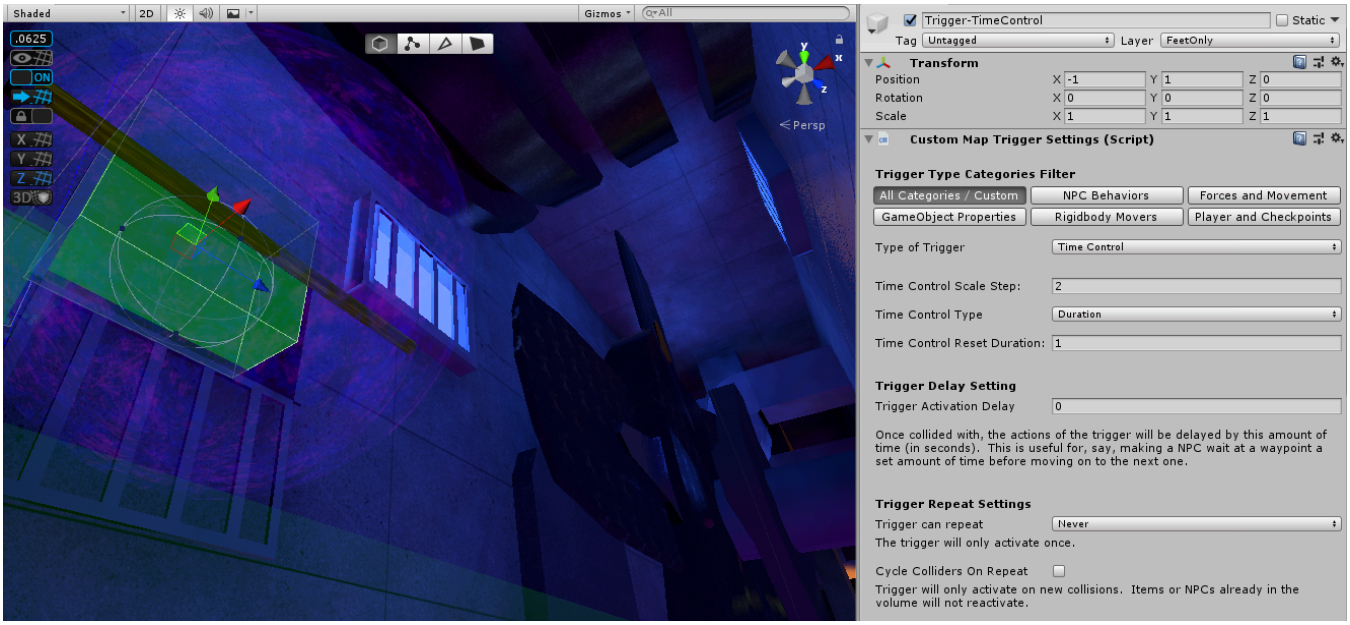
It Puts the Lotion On Its (Custom) Skin - [Trigger-ChangePlayerModel]

I followed suit with Mara's PlayerModels mod, reading in the *.body* files and providing a menu drop-down so mappers could easily select which model they wanted to apply to the player when hitting a Trigger-ChangePlayerModel volume. This trigger simply reflects into Mara's PlayerModels mod and executes the swap. I'd say where and when in Melon Vault this is used, but it's pretty obvious for those who have played through the campaign and will remain a nice surprise for those who have not yet had the chance.



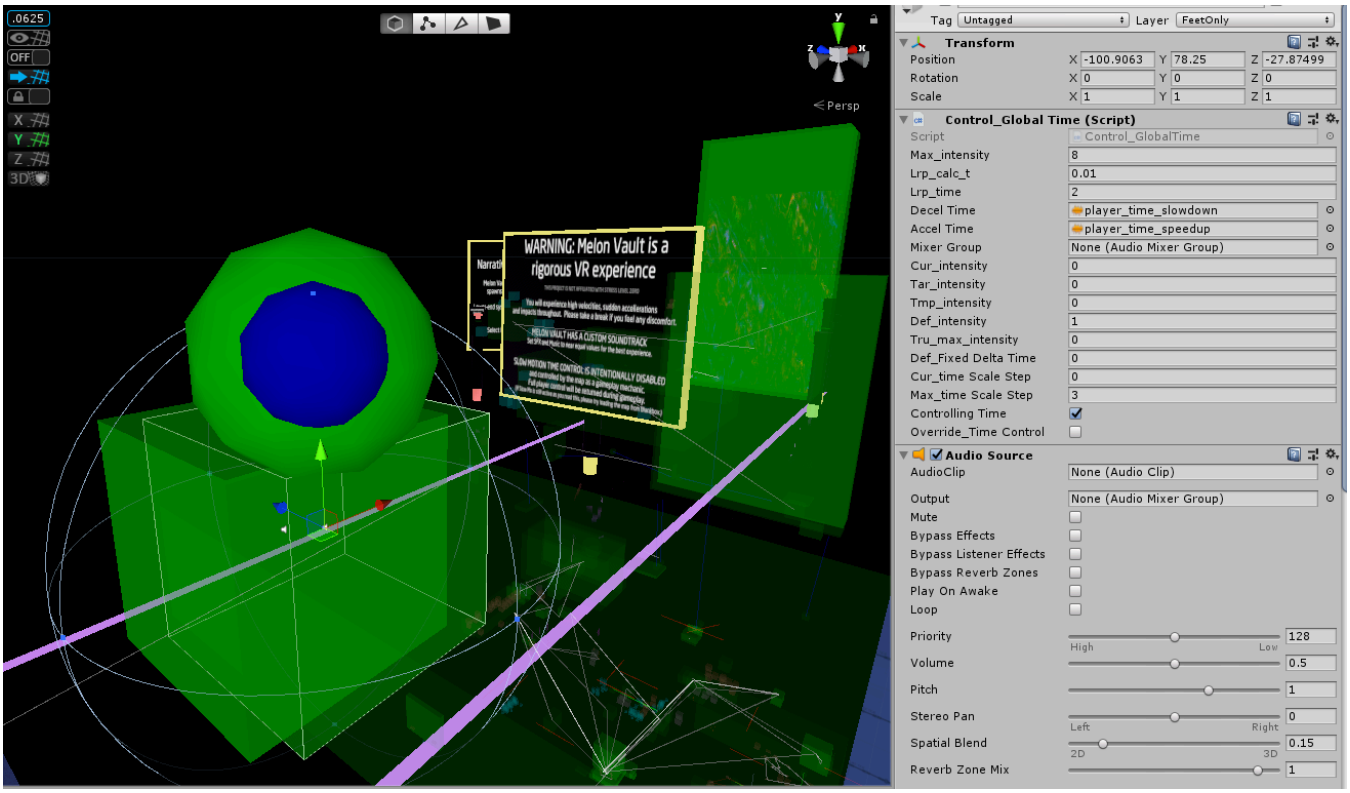
It’s About Time - [OverridePlayerTimeControl and Trigger-TimeControl]

With all of the excitement and work that came with the CustomMapTemplate release behind me, I turned back to Melon Vault’s development. One of the coolest features in Boneworks was the ability to slow time. However, without mods or rebinding inputs, the player usually lost the ability to maneuver for the duration since the left thumb couldn’t control movement while also holding down the Slow-Mo button. I also never liked how easy the game became since the player could slow time at will and for as long as they wanted. My next goal was to harness the power of time control for the map designer so it could be used as a game mechanic in hopefully interesting ways, rather than an at-will easy-mode crutch (insert old man speech about video games being difficult “back in my day…”).

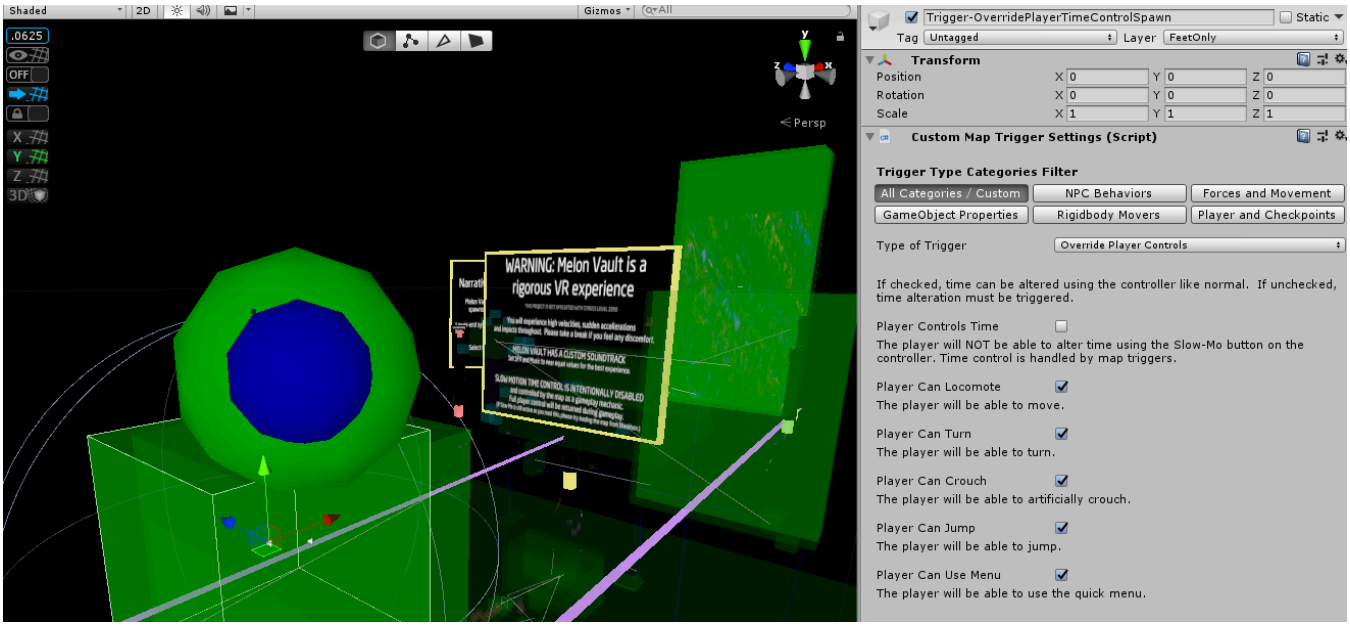


The ConfigureOverridePlayerTimeControl trigger was a “configuration” trigger that only needed to fire once and was typically used at the map’s spawn location. This configuration was applied through triggers as a way to circumvent not having access to the Player Rig within the Unity editor--modifications to the player could only happen at runtime and after the player existed in the map.

This trigger grabbed references to the Control_GlobalTime component and its AccelTime and DecelTime audio clips from the Player’s DataManager, and the HeadSFX AudioSource from the RigManager. It then copied these values into a separate version of the Control_GlobalTime component that resided on the ConfigureOverridePlayerTimeControl trigger itself. This trigger object also contained its own AudioSource with Spatial Separation set to 2D so that audio played from it could be heard regardless of the player’s position in the map. Copying these values to some non-player object like the ConfigureOverridePlayerTimeControl object was necessary because access to the components that resided on the Player Rig became null or inaccessible once the ability to slow time was disabled.



Overlapping the ConfigureOverridePlayerTimeControl trigger was the OverridePlayerControls trigger, which allowed the map designer to configure everything from a player's ability to locomote, jump or use the radial menu. Most importantly, the BodyVitals component included a way to disable the Slow Time button, and with the player's time control powers removed, the only remaining way to slow time was through triggers laid down by the map designer. Note that when applying any BodyVitals changes via code, one had to perform a soft propagation of any modified values before they took effect.



Creating Custom NPCs - Stealth Hunting “Security Seekers”

After spending weeks toying with Boneworks NPCs and being unable to get the behaviours I needed from them, I decided to create my own stealth hunter NPC. The key features required were as follows:

- When out of combat, the NPC would continually patrol a specified route, stopping at each waypoint and performing some sort of idle action, like looking around.

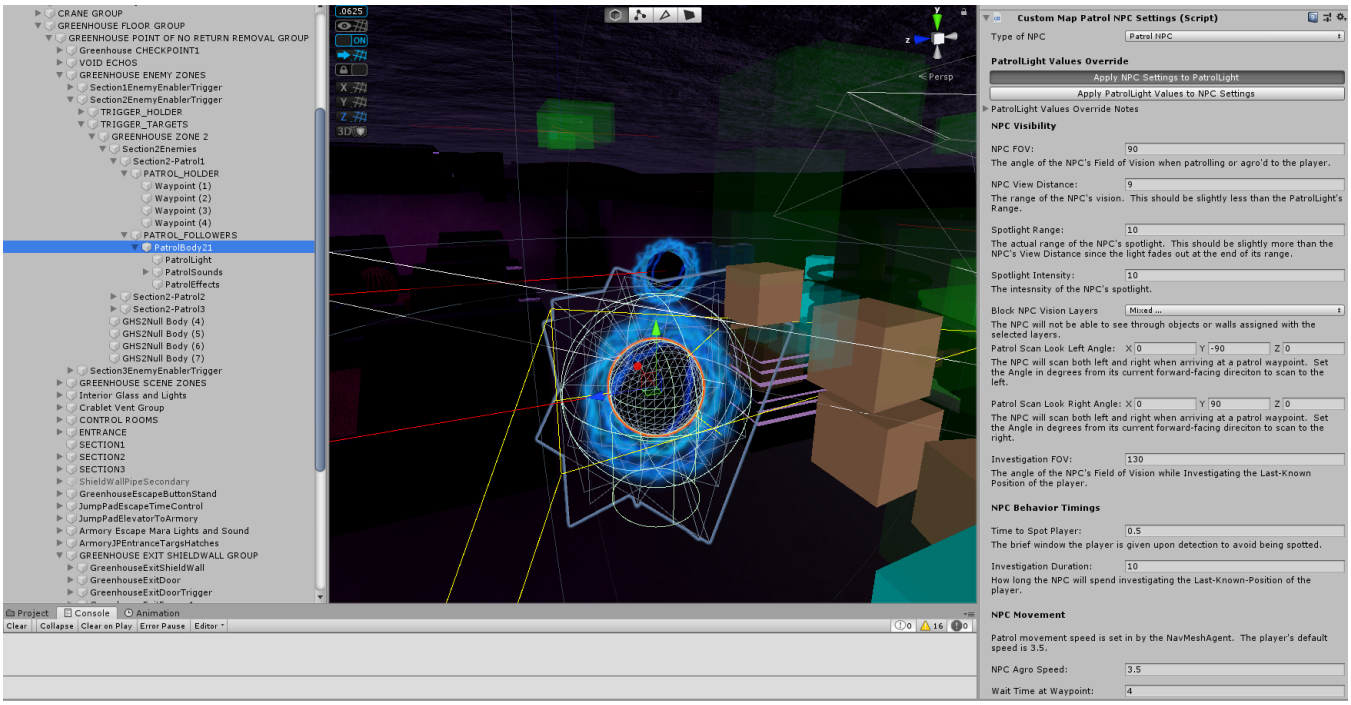
- The NPC needed to be able to spot the player but not agro to them immediately. Players needed to have a visual and audible warning with a small window of time within which they could react and hide before being attacked.

- Once agro’ed onto the player, the NPC needed to be able to navigate the map with ease. I would become well-versed with the Nav Mesh by the end of this process.

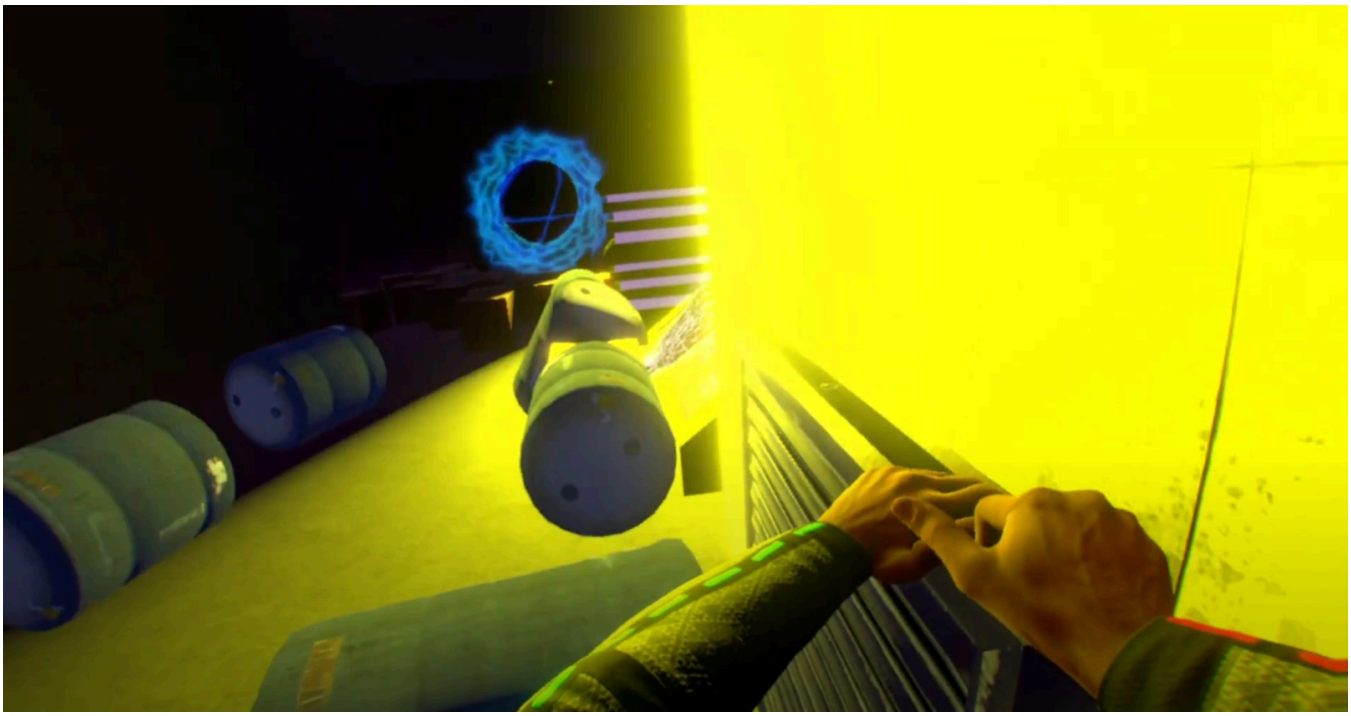
- If the player broke line of sight, the NPC would lose agro but rush to the player’s last-known position and enter an Investigation mode to try to reacquire the target. This mode widened the NPC’s FOV and checked all directions, making it very risky for the player to stay anywhere near the Stealth Hunter.

- If the NPC failed to find the player after the Investigation mode completed, it would return to its patrol loop.

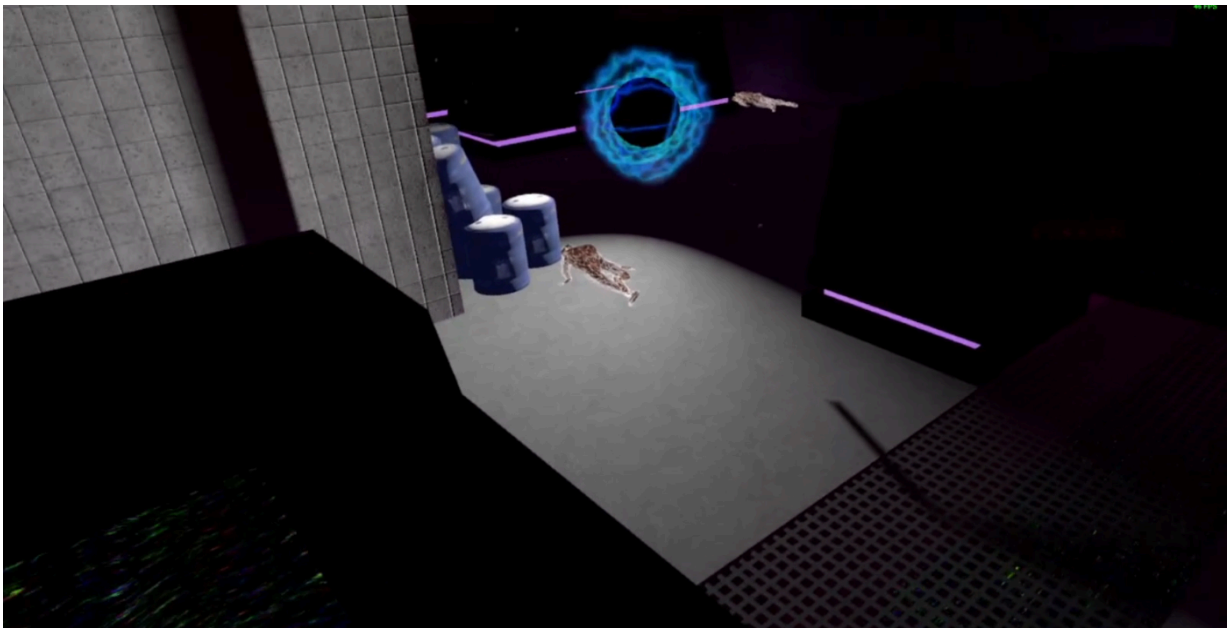
```
3 references
private void SetWaypoint()
{
    if (waypoints.Length > 1)
    {
        Vector3 targetWaypoint = waypoints[currentWaypointIndex];
        navAgent.SetDestination(targetWaypoint);

        //MelonLogger.Log("New waypoint [ "+currentWaypointIndex+"]: " + targetWaypoint.ToString());
        patrolMode = true;
    }
}

1 reference
private void PlayerSpotted()
{
    if (audioNpcPlayerSpotted.isPlaying == false)
    {
        audioNpcPlayerSpotted.Play();
    }
    spotLight.color = playerSpottedSpotlightColor;
    spotLight.spotAngle = originalViewAngle;

}

1 reference
private void AggroPlayer()
{
    aggrodToPlayer = true;
    spotLight.color = agroSpotlightColor;
    spotLight.spotAngle = originalViewAngle;
    if (audioNpcAgro.isPlaying == false && agroSoundPlayCount == 0)
    {
        audioNpcAgro.Play();
        agroSoundPlayCount++;
    }

    lastKnownPlayerPosition = player.position;
    navAgent.SetDestination(lastKnownPlayerPosition);
    transform.LookAt(lastKnownPlayerPosition);
    investigationTimer = timeToInvestigate;
}

1 reference
private void InvestigateLastKnownPosition()
{
    if (investigationMode == false)
```



Stealth in VR had limitations that flat-screen games rarely needed to consider. Most stealth action games positioned the game camera above the player's shoulder, which allowed them to see around corners and get an overview of the area they were traversing. Boneworks is necessarily first-person, so care was taken to provide visual and audio cues to the player that would allow them to sneak past these enemies successfully, while paying a high price if they tried to rush past them haphazardly.



Aesthetically, I wanted this entire section to place the player in darkness. The FOV of the NPC would be visualized by the arc of a Realtime spotlight. This would afford the player a way to track the NPCs even if they were around a corner and, based on the light's direction, know whether it was safe to poke out from behind cover. I also chose to make the NPCs impossible to destroy because I wanted the player to feel vulnerable and lean into the need to be sneaky.



The Greenhouse is the most "hot" or "cold" section of the map for players based on feedback. They either love it or hate it… though I suppose the same could be said for flat-screen stealth gameplay. I am happy to accept that some players simply do not like having to stealth.

However, to my great joy and satisfaction, I have multiple recordings from playtesters and others who have tried the map since its Beta 2 release in which some players become literally terrified during this section. You can watch their VR view shake as they tremble around corners, mutter with dread as they hide after nearly being spotted and scream at the top of their lungs whenever they get caught. Hearing from these folks that they found this section of the map to be one of the most intense and fun VR experiences they've had in the medium easily made the hours spent developing the gameplay and custom NPC for it well beyond worth it.

The validation ensures an expansion on these ideas, with lofty goals of maps with fully-destructible lights and proper low-light-to-bright-light and noise-based detection systems for enemy NPCs, all within Boneworks… a boy can dream.

Creating Custom NPCs - Plasma Turrets

One of my favorite features about Boneworks was how seamlessly it handled destructible objects. The Plasma Turret NPC in Melon Vault was my ode to destruction. I will say that the gameplay idea I had in my head wasn't quite realized, but it worked well enough that, from it, I've taken in several lessons and will be applying them to future custom NPC enemies.

```
if (player && hasBeenTriggered)
{
    if (autoFireMode == true)
    {
        timeToFireTimer += Time.deltaTime;

        if (timeToFireTimer > timeToFire)
        {
            FireProjectile();
            timeToFireTimer = 0;
        }
    }
    else
    {
        CanSeePlayer();
    }


    if (isScanning == true)
    {
        NPCSearchScanArc();
    }

    //playerVisibleTimer = Mathf.Clamp(playerVisibleTimer, 0, timeToSpotPlayer);
    timeToFireTimer = Mathf.Clamp(timeToFireTimer, 0, timeToFire);
    turnDurationTimer = Mathf.Clamp(turnDurationTimer, 0, turnArcTime);

    // Clamp Y axis, allow other axis rotation, adjust X axis to below head?
    transform.localEulerAngles = new Vector3(transform.localEulerAngles.x, Mathf.Clamp((transform.localEulerAngles.y <= 180) ? transform.localEule

    if (isScanning == true && npcCanSeePlayer == false)
    {
        turnDurationTimer += Time.deltaTime;
    }

    if (turnDurationTimer > turnArcTime)
    {
        turnDurationTimer = 0;
    }

    if (npcCanSeePlayer == true)
    {
        if (timeToSpotPlayer > 0)
        {
            playerVisibleTimer += Time.deltaTime;
        }
```

The vision was to have a very physics-based weapon… in this case, a high velocity, heavy as hell cannonball rigidbody wrapped in a plasma-looking particle that would smash through destructible cover whenever you were spotted. The goal was to change the layout and safe zones of the combat arena in a very dynamic way. At the end of the initial encounter with the turret, the intention was to have it go haywire and force the player to make a mad-dash to escape its rapid-fire onslaught and reckless destruction of every last bit of cover.

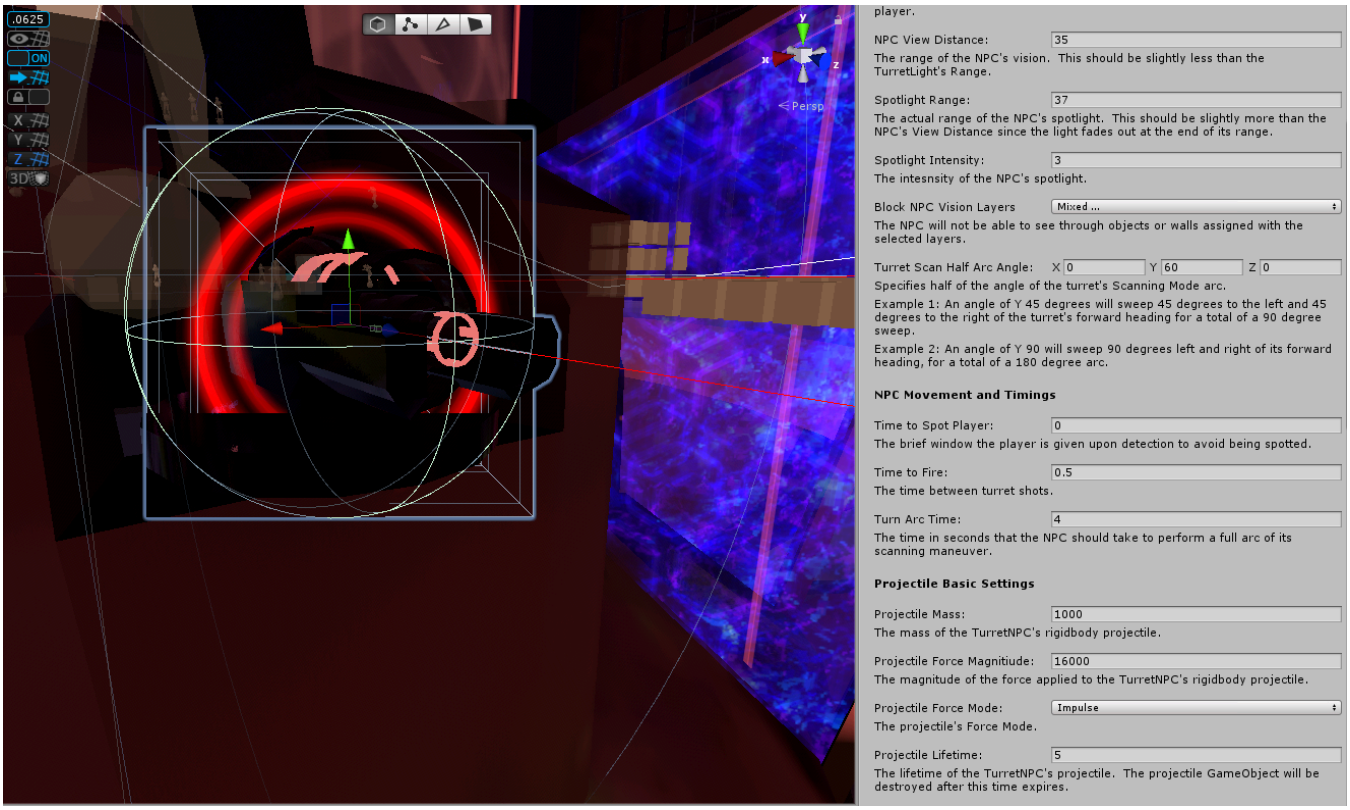
While these events did take place in the North section of the Melon Vault boss room, I found in playtesting that if I ramped up the damage of the Plasma Turret too high, the player would die so often that they would become frustrated. I lessened the damage so that the turret would still plow through crates and dynamically change the cover available, but would merely push the player around and cause recoverable damage to them. Perhaps one day, I'll release a "hard mode" that ramps this damage back to its original values.

The Turret had a two different modes:

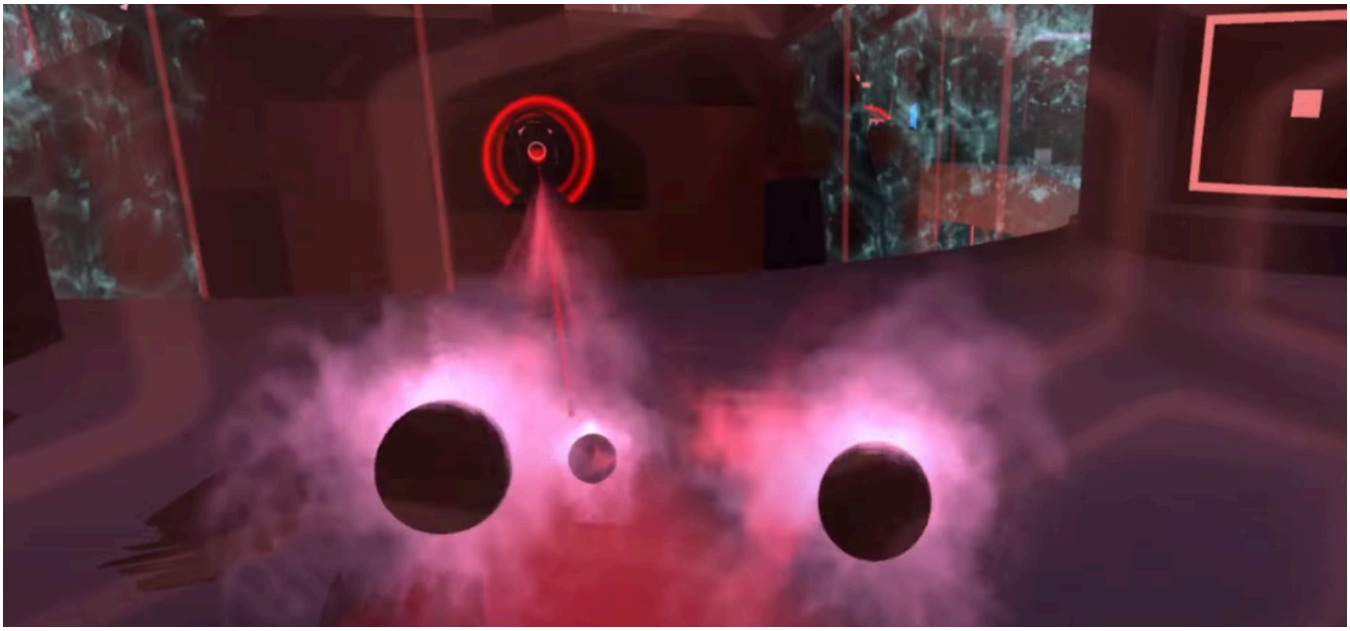
- The default Scanning mode would sweep back and forth looking for the player in an arc and period of time defined by the map designer.

- The Auto-Fire mode would continually fire projectiles and did not track the player.

The NPC's FOV, rate of fire, projectile mass, force at which the projectile was launched, how long a plasma ball lived before being destroyed and whether it should be affected by gravity were all configurable options. The modes could be swapped with triggers by the map designer, allowing the turret to hunt the player during one sequence or enter a clear-the-room style bombardment of plasma balls the next, based on player activity or progress. An added Powerable_DamageVolume component to the Plasma shots caused pulsing fire damage to the player if they lingered near them.



In the Boss room, the Turret's plasma cannonballs were considered unique objects for the Trigger-Rigidbody puzzles and had to be used to complete the first two areas. In both cases, the player had to coax the turret to fire in their direction while they stood near an exhaust-port-like target that could only be activated by the plasma turret's weapons. The second puzzle required the player to power up a barrier by completing a zipline and target shooting challenge, then action-hero grab an enormous plug that was hanging aloft in the area. The player's body weight was required to lower it into its receptacle, which activated a mobile barrier that could be moved along a track. Finally, the player needed to use it to reflect incoming fire from the turret at the correct angle in order to hit a plasma-ball-only target. Completing this sequence would drop the final barrier to the Boss combat encounter.



Blowin’ In the Wind - [CMIRigidbodyMover and CMIRigidbodySwinger]

One of the sections of Melon Vault that consistently received praise by playtesters was the Fan Room. While fans that were fully rotating in this section were handled by simple animations, the fans that swung back and forth were driven by an early Coroutine experiment I made well before I knew what I was doing. Throughout development, I always intended to update those routines to ConfigurableJoints, but never got around to it. The simple Sin(Time.time * swingSpeed) somehow still powered their rotation upon release. Fortunately, Mara and I intend to include a far more physics-friendly ConfigJoint version of these swinging fans when we push the next update to the CustomMapTemplate.

Null Thicc Bodies [NPC Scaling]

One of my favorite features that CMI provided fairly late in Melon Vault's development was the scaling of NPCs. As often as I had mowed down Null Bodies when making my video guides and various playthroughs, I wanted to see if I could modify the Boneworks NPCs to introduce the player to new and unexpected challenges. While some playtesters simply viewed the scaled-up NPCs as being bullet-spongy, they often overlooked how the scaling of those NPCs forced a change in sightlines and aiming priorities and motivated players to score headshots to offset the (optionally) increased health.

NPC scaling had been done by others through mods in the past, but they simply changed the scale of the prefab that spawned the enemy, meaning NPCs would often just fall over in a big mess, unable to move, react or be any sort of danger to the player. These enemies were also only spawnable by the Utility Gun or Easy/BoneMenu, and couldn't be a native spawn within a custom map.

```
1 reference
public void ReplaceZoneSpawnersWithScaledScriptableObjects(LocalizedText localizedText)
{
    if (CheckZoneSpawnerScale(localizedText))
    {
        string npcToScale = localizedText.key;
        Vector3 npcScale = localizedText.transform.localScale;
        string scriptObjTitle = npcToScale + " Scaled " + npcScale.ToString();

        foreach (string titleText in PoolManager._registeredSpawnableObjects.Keys)
        {
            if (PoolManager._registeredSpawnableObjects[titleText].title.ToString() == scriptObjTitle)
            {
                SpawnableObject spawnableScaledNPC = PoolManager._registeredSpawnableObjects[titleText];

                ZoneSpawner npcZoneSpawner = localizedText.GetComponent<ZoneSpawner>();
                if (spawnableScaledNPC != null && npcZoneSpawner.spawnObject.title == npcToScale && localizedText.gameObject.transform.localScale != new Vector3(1f, 1f, 1f))
                {
                    //MelonLogger.Log("Replaced Zone Spawner with scale " + npcZoneSpawner.gameObject.transform.localScale.ToString() + " with spawnObject with scaled version: "
                    //MelonLogger.Log("spawnObject UUID: " + spawnableScaledNPC.UUID);
                    npcZoneSpawner.spawnObject = spawnableScaledNPC;
                }
            }
        }
    }
}

2 references
public bool CheckZoneSpawnerScale(LocalizedText localizedText)
{
    Vector3 zoneSpawnerScale = localizedText.transform.localScale;
    Vector3 noScaling = new Vector3(1f, 1f, 1f);

    if (zoneSpawnerScale == noScaling)
    {
        //MelonLogger.Log("ZS " + localizedText.gameObject.name + " with scale: " + zoneSpawnerScale.ToString() + " should NOT be scaled.");
        return false;
    }
    else
    {
        //MelonLogger.Log("ZS " + localizedText.gameObject.name + " with scale: " + zoneSpawnerScale.ToString() + " matched for scaling.");
        return true;
    }
}
```

My attempt to solve these issues went through several iterations, but the version that made it into Melon Vault merely required the map designer to set the scale on the ZoneSpawner GameObject in Unity. That's it. CustomMapInteractions handled everything else, like dynamically creating the scaled up NPC's ScriptableObject, adjusting the muscle springs/weights/dampers in the appropriate ConfigurableJoints and rigidbody mass values so that the enemies wouldn't collapse under their own weight and, finally, caching those changes into a corresponding prefab. Subsequent NPCs of the same scale would reuse this cached Prefab. This solution was crude and still had limits, but it worked well enough for most enemies to provide interesting gameplay.

Melon Combat

Combining the Trigger-Targets, Trigger-Waypoints and various SceneZone triggers together powered almost all of the combat interactions in Melon Vault. The opening half of the map purposefully gave the player very little firepower (unless they went secret-hunting) so that once they reached the Armory and geared up, that sense of vulnerability was jettisoned and the player could unleash a torrent of bullets using whatever loadout they desired.

On paper, each combat section had a goal to introduce the player to a new facet of combat, though if this purpose was realized, it was unclear from playtesting, as most people just shot shit or died trying:

- The hallway just past Armory's blue barrier provided level sight lines and plenty of cover, with enemies being funneled into a door chokepoint.

- Turning the corner, the player now faced several enemies and uphill fights, as the Corrupted Nulls with throw attacks had upper ground and a bit of cover. The player was forced to expose themselves in order to place headshots. The high rate of fire from the Corrupted Nulls was intentional, with the goal to either push the player to stay mobile or fight from behind cover. Given that “cover” was often offered in the form of a destructible crate, it did not last long against an onslaught of throw attacks.

- Entering the Tiers room, the player now had a playground of jump pads to use as they tackled this multi-leveled combat scenario. The player started at the mid level of the arena and could choose to take a high path along the edges of the room or a low path through the room’s center. A grabbable set of cargo crates moved along the ceiling of the map and could be reached at high risk using jump pads on the lower level. Corrupted Nulls with throw attacks were scattered around the room to encourage the player to keep moving.

- A few moments of quiet greeted the player past the Tiers room. A Void Echo explained that GreasedScotsman was going to try “something” with the Nulls up ahead. This “something” was an explanation for a custom enemy fizzler I created to avoid NPC mobility problems that arose when Null Body corpses piled up on the ground. By default, Boneworks eventually removed dead NPCs from the scene as a matter of maintaining hardware performance. In my case, I needed NPCs to be immediately removed on death so that the remaining Null Bodies trying to attack the player would not trip over their fallen brethren, removing all hints of challenge or combat pressure whenever they did.

  The goal of this combat scenario was to thrust the player into close-quarters-combat, fully surrounded and in low-light conditions, and then offer them map elements like Void Pools and “Voidfalls” into which they could push the crowd of Nulls, if desired. Scaled-up Nulls abounded in this section as a way to force different aiming sightlines on a flat combat arena.

  The final section of CQC combat closed the walls in even further, leaving the player very little wiggle room around a constant rush of Null Bodies. Further, Nulls now dropped from pipes in the ceiling and moved much faster than normal. An unsuspecting player could find themselves surrounded, often backing into a corner where they were forced to aim well, holding what little ground they had left. I tweaked the faster Nulls such that, if they reached their top movement speed on approach, they would actually dart past the player and then smack them in the back of the head. This meant the player often had to twist and turn, and could not rely on being able to keep all targets ahead of them.

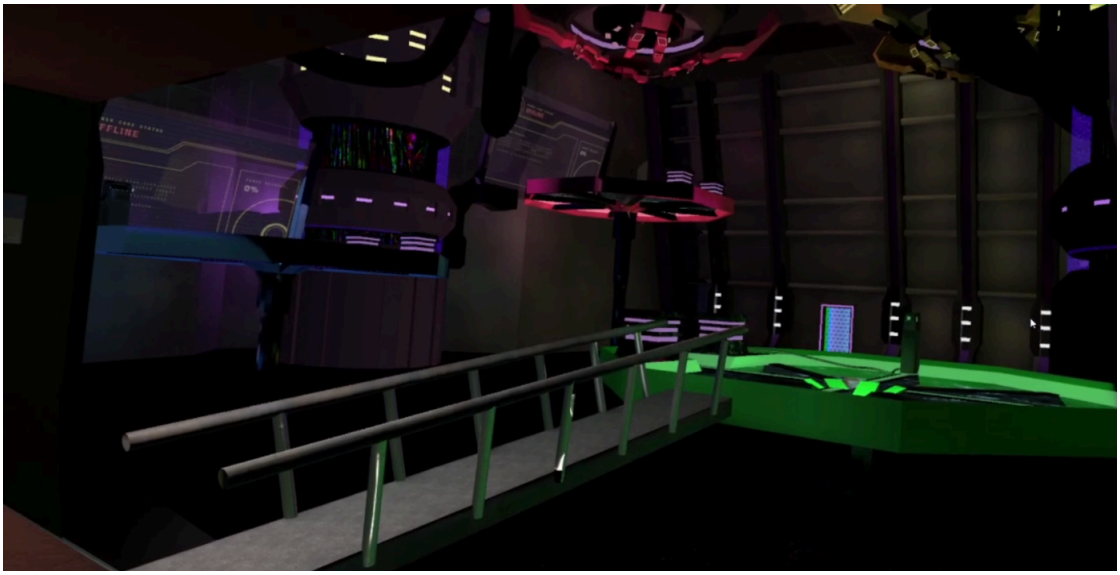
  That was the intent of the encounter, anyway. Sometimes, players just whacked Null Bodies with the frying pan until nothing was left but orange jelly… which was, is and always will be the beauty of Boneworks gameplay. There’s never a “right way” to play the game.

- The Mailroom was the final combat section before the boss. This area was all about the player having a badass Hollywood-esque slow-motion-slide-past-enemies-as-you-blew-them away type of experience. It served as a (hopefully) thrilling capstone before a forced lull in combat provided by the Reactor Room's Golf Puzzle.

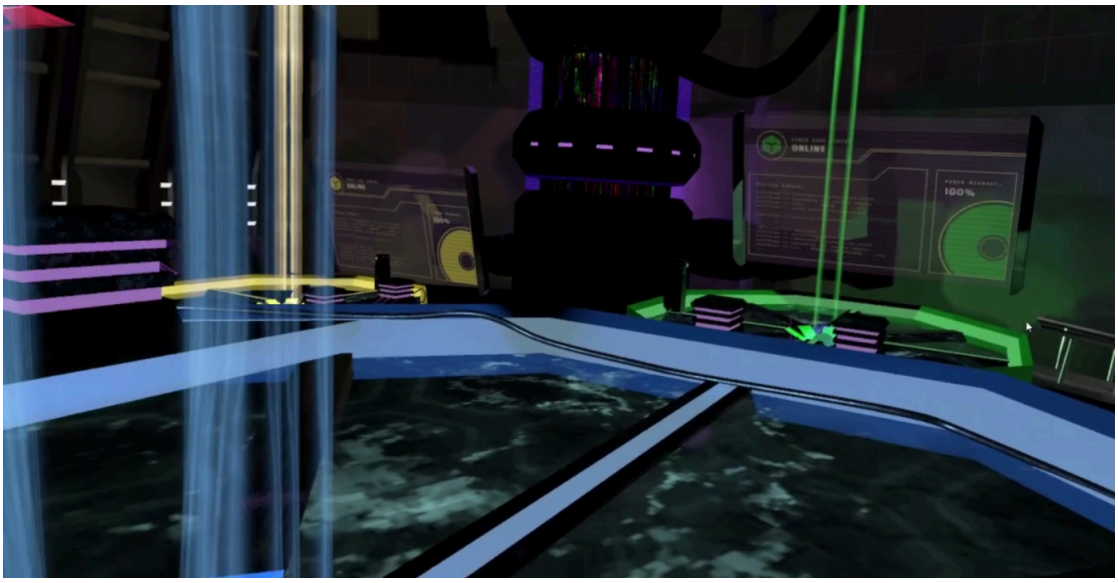
Powering Puzzles and Dropping the Lever

The earlier Lever "puzzles" in the Greenhouse Stealth sections were achieved by simply modifying the ConfigurableJoint on the Lever so that it would drop under the influence of gravity. Players had to come up with creative ways to keep the lever aloft so they could simultaneously reach a nearby button normally blocked by a barrier that only dissipated when the lever was in its highest position.

What may not be obvious, however, was that all of the button, slider and lever actions in the map did not use SLZ's native electricity/power system. At the time of Melon Vault's development, there was no way to inject custom events into the various OnPress and PowerLever actions that these items typically activated. Sure, one could drive a ConfigJoint value or toggle an object on or off, but anything more complex that required custom code to function was inaccessible. To get around this problem, all of the CMI versions of buttons, dials, levers and sliders actually contained small triggers and hidden unique activator colliders that were used to fire off the desired CMI events.



The Golf Room was the map's main puzzle, and brought a number of triggers into play: Jump pads launched the "golf" balls and the player across the Void to each new area. A series of buttons on each platform enabled the corresponding jump pads. The TeleportObject triggers provided ways to recover any golf balls that were lost to the Void. A RigidbodySocket trigger only accepted the correct golf ball into its hole and locked it into place. A series of custom toggle triggers fired when each golf ball reached its target and enabled the particle effects and monitor screens that indicated the player's progress. A set of colored exit barriers dropped as each golf ball was seated into its hole.



Finally, The Boss Fight - A Stage by Stage Breakdown

Melon Vault: Showdown's main Boss was a 6x scaled-up Omni with far superior aiming capabilities than a normal Omni Projector, had a butt-load of health and was safely tucked away behind a shield that blocked all ranged weaponry. The fight was split up into multiple stages, successful only if the player did the following:

1. Survived the initial Omni Projector Ambush.

2. Navigated to the Blue Power Reactor only reachable using the player's jumping momentum and a series of jump pads.

3. Smashed the safety glass of the Blue Reactor and raised the slider lever to begin an overload sequence. This overload also opened the path to the Yellow Power Reactor.

4. As the Blue Reactor detonated, a wave of Omni Projectors appeared and several got blown off of the Blue Power Reactor platform.

5. The Blue Reactor's explosion temporarily dropped the shield surrounding the boss, providing a limited window to shoot them. The player was incentivized to dispatch the reinforcement wave of Omnis quickly to maximize direct boss damage. However, with the barrier dropped, the player had to take into account the Boss's firepower.

6. After a minute, the shield was restored and another wave of Omnis appeared.

7. The path to the Yellow Power Reactor was a mobile platforming and jump pad challenge.

8. Detonating the Reactor was accomplished in the same way as with the Blue Reactor, which would cause the Boss's shields to drop, and a wave of reinforcements to spawn.

9. After a minute, the Boss's shield was restored once more and yet another Omni wave was spawned.

10. Destruction of the Yellow Reactor opened the path to the final Red Power Reactor and teleported the Ammo Dispenser into the Red section so there was never a way for the player to run out of ammo.

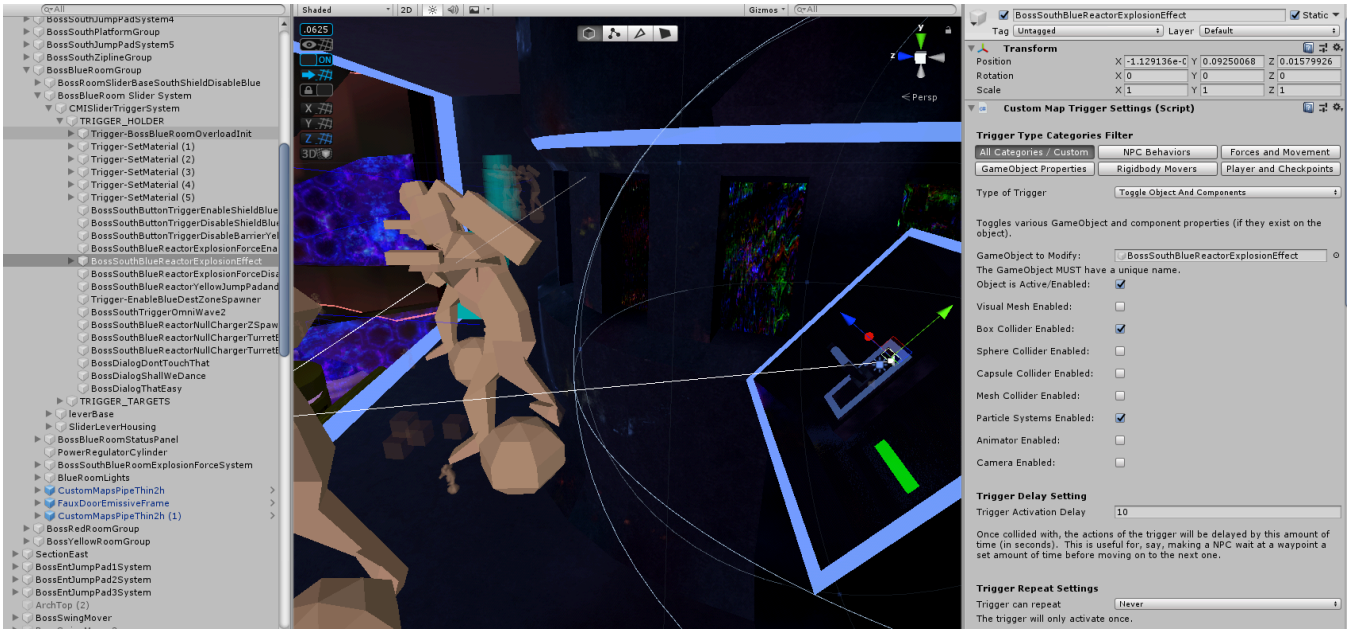
11. A small ladder climb, a huge jump pad launch, and an extremely long zipline slide formed the path to the Red Power Reactor. Along the way, the player would see droves of Null Bodies throwing themselves at the boss to little effect thanks to a battery of Plasma Turrets. These would detonate as the player approached the huge jump pad launch to the Red platform.

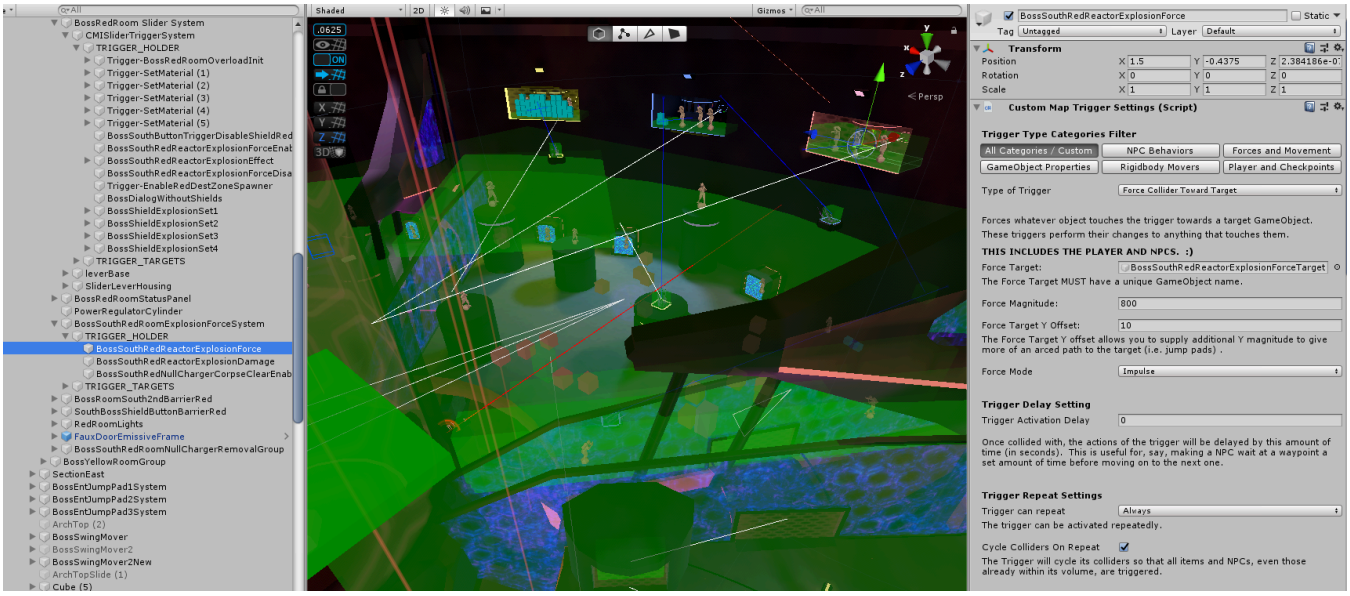
12. Overloading this final Red Reactor would keep the Boss's shields down for good. As it detonated, the final wave of Omni reinforcements would spawn.

Fingering the Pulse [Trigger-MonitorZoneSpawner]

The boss's health was tracked by the CMI [Trigger-MonitorZoneSpawner] trigger. This particular behaviour could better be handled with subscribed events, but I didn't yet have a clear idea of how I could provide such monitoring events as a Unity tool that map developers could easily apply.

At certain milestones of health, the boss would utter voice lines indicating the damage the player had inflicted. When the boss fell below 30% health, Null Bodies and Corrupted Nulls started spawning from above and would assist the player in killing their shared enemy. Upon death, the music would stop and a now-familiar Void Fissure would appear. Touching it caused the Finale dialog to trigger, where GreasedScotsman thanked the player and explained the real purpose behind his reasons for bringing the player to Melon Vault. After a final portal animation played, if the player stepped through, they would be teleported to the Thank You/Credits sequence video.



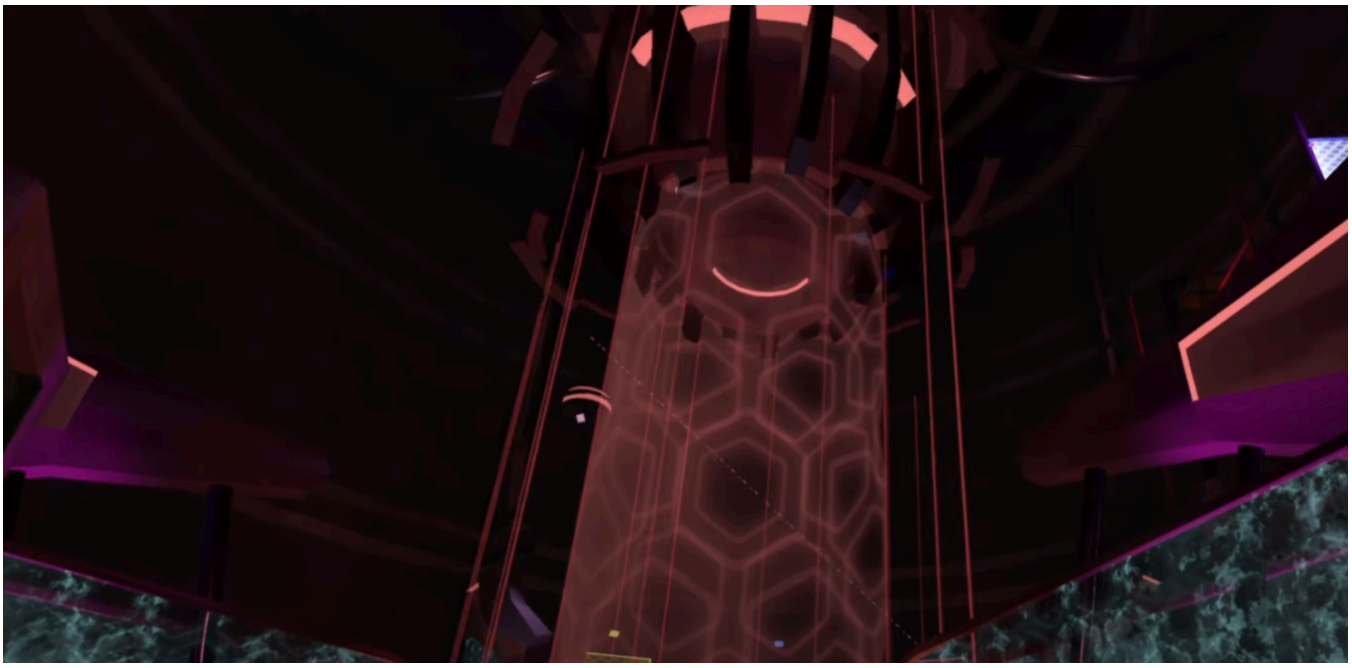


If You Can't See It, Is It There?

Partway through Melon Vault's development, I hired TabloidA and Maranara as environmental artists because I lacked the modeling skill and artistic eye to provide the amazingly detailed vistas that the final version of the campaign showcased. (My nascent art, texture and modeling prowess is featured in the entirety of the Golf Room, most of the Greenhouse and the "Boss Slide n' Climb" sections). As the team went to work replacing my probuilder box placeholders and the geometric complexity of the level increased, we noticed a drop in performance as if Unity's occlusion wasn't working as intended.

Occlusion was definitely happening, as forgetting to bake Occlusion was glaringly obvious and even more performance-sapping. However, some parts of the map, no matter how hidden, even according to Unity's various occlusion visualizers, still seemed to drag on performance. After tons of testing and working at the problem and absolutely refusing to compromise on the incredible visual quality that Tabloid and Maranara were producing, I came up with an idea for trigger-based occlusion.

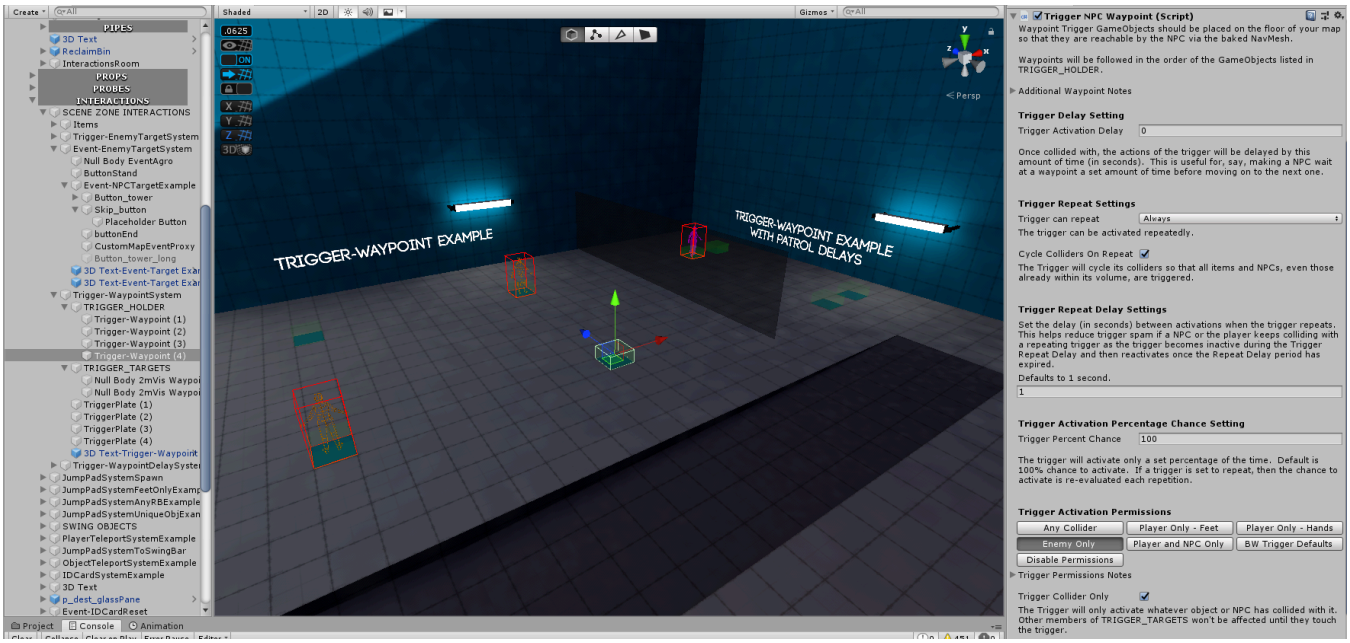
When the player first spawned into Melon Vault, almost every section of the map that they could not see from the spawn Apartment's vantage point had a HideOnAwake component. As they progressed, these hidden sections became enabled. Similarly, as points of no return were passed, like jumping out the window onto a set of I-beams near the Crane, entire sections of the map that would not be visited again got removed. By the end of Melon Vault, pretty much only the boss room remained active. This system utilized carefully-placed SLZ PlayerTriggers that showed and hid each section at the appropriate point of progress. The performance gains were quite noticeable, and if later versions of Unity exhibit this strange not-quite-providing-occlusion behavior, I intend to flesh out this system and make it available in future CustomMapTemplate updates.

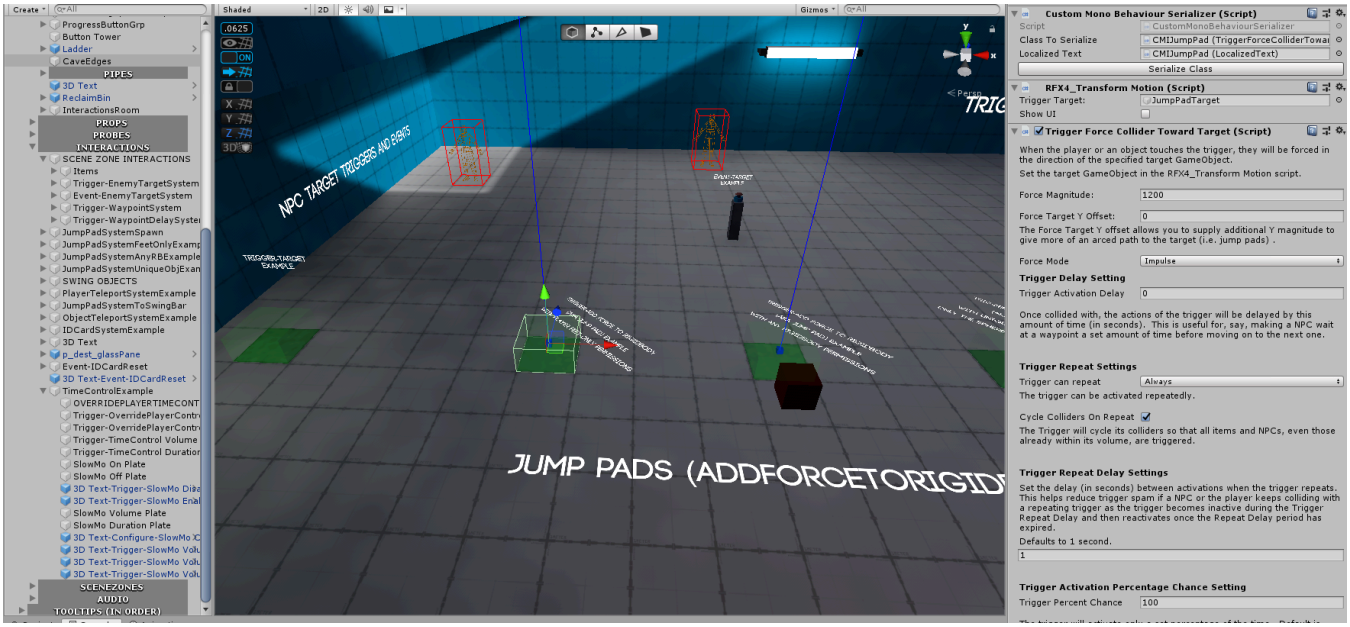


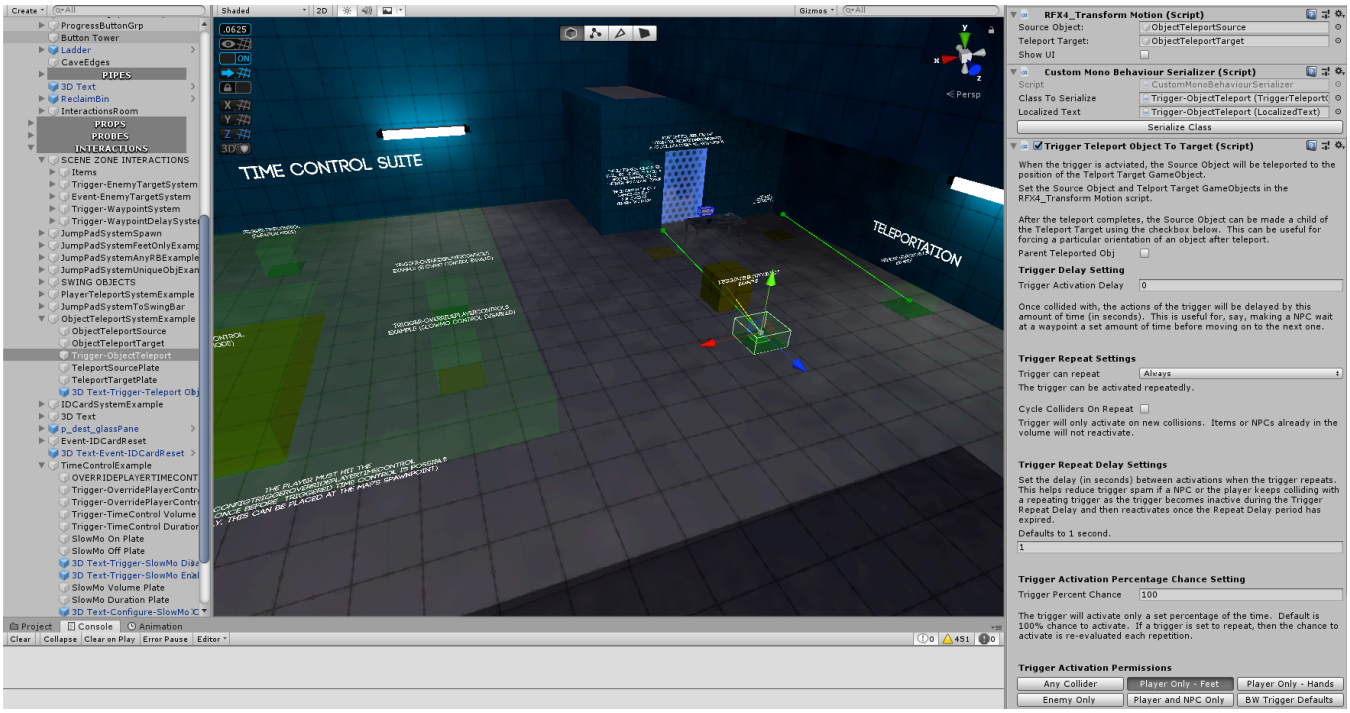
Holy Hell, You Made It!

Thank you for your interest in learning how the Melon Vault sausage was made. I will eventually have a full Custom Map development video series that tackles teaching folks all of the new tools that Maranara and I have been creating for the upcoming CustomMapTemplate release. I started on this endeavor but quickly realized we were about to paradigm shift the tools, like so:

I'm happy to report that amazing work by gnonme, trevtv and Maranara have brought the ability to write and apply custom Monobehaviours directly in the Unity editor, apply them to objects throughout your custom map, and Boneworks will not strip them away. Mappers can finally author custom code directly in Unity without the need for it to be handled by additional mods. All of the tools that make up CustomMapInteractions will now be integrated into the Template and available as custom Monobehaviours. As a bonus for making it this far into the document, I offer a sneak peek at the upcoming template and tools. We can't wait to see what mappers and artists can do with these tools once we get them released!


TRIGGER-WAYPOINT EXAMPLE
TRIGGER-WAYPOINT EXAMPLE WITH PATROL DELAYS
Trigger NPC Waypoint (Script)
Waypoint Trigger GameObjects should be placed on the floor of your map so that they are reachable by the NPC via the baked NavMesh.
Waypoints will be followed in the order of the GameObjects listed in TRIGGER_HOLDER.
Additional Waypoint Notes
Trigger Delay Setting
Trigger Activation Delay
Once collided with, the actions of the trigger will be delayed by this amount of time (in seconds). This is useful for, say, making a NPC wait at a waypoint a set amount of time before moving on to the next one.
Trigger Repeat Settings
Trigger can repeat
Always
The trigger can be activated repeatedly.
Cycle Colliders On Repeat
The Trigger will cycle its colliders so that all items and NPCs, even those already within its volume, are triggered.
Trigger Repeat Delay Settings
Set the delay (in seconds) between activations when the trigger repeats. This helps reduce trigger spam if a NPC or the player keeps colliding with a repeating trigger as the trigger becomes inactive during the Trigger Repeat Delay and then reactivates once the Repeat Delay period has expired.
Defaults to 1 second.
Trigger Activation Percentage Chance Setting
Trigger Percent Chance
The trigger will activate only a set percentage of the time. Default is 100% chance to activate. If a trigger is set to repeat, then the chance to activate is re-evaluated each repetition.
Trigger Activation Permissions
Any Collider
Player Only - Feet
Player Only - Hands
Enemy Only
Player and NPC Only
BW Trigger Defaults
Disable Permissions
Trigger Permissions Notes
Trigger Collider Only
The Trigger will only activate whatever object or NPC has collided with it. Other members of TRIGGER_TARGETS won't be affected until they touch the trigger.


NPC TARGET TRIGGERS AND BEAMS
JUMP PADS (ADDFORCETORIGID
Custom Mono Behaviour Serializer (Script)
Class To Serialize
Localized Text
Serialize Class
RFX4_Transform Motion (Script)
Trigger Target:
Show UI
Trigger Force Collider Toward Target (Script)
When the player or an object touches the trigger, they will be forced in the direction of the specified target GameObject.
Set the target GameObject in the RFX4_Transform Motion script.
Force Magnitude:
Force Target Y Offset:
The Force Target Y offset allows you to supply additional Y magnitude to give more of an arced path to the target (i.e. jump pads) .
Force Mode
Impulse
Trigger Delay Setting
Trigger Activation Delay
Once collided with, the actions of the trigger will be delayed by this amount of time (in seconds). This is useful for, say, making a NPC wait at a waypoint a set amount of time before moving on to the next one.
Trigger Repeat Settings
Trigger can repeat
Always
The trigger can be activated repeatedly.
Cycle Colliders On Repeat
The Trigger will cycle its colliders so that all items and NPCs, even those already within its volume, are triggered.
Trigger Repeat Delay Settings
Set the delay (in seconds) between activations when the trigger repeats. This helps reduce trigger spam if a NPC or the player keeps colliding with a repeating trigger as the trigger becomes inactive during the Trigger Repeat Delay and then reactivates once the Repeat Delay period has expired.
Defaults to 1 second.
Trigger Activation Percentage Chance Setting
Trigger Percent Chance


TIME CONTROL SUITE
TELEPORTATION
RFX4_Transform Motion (Script)
Source Object: ObjectTeleportSource
Teleport Target: ObjectTeleportTarget
Show UI
Custom Mono Behaviour Serializer (Script)
Script: CustomMonoBehaviourSerializer
Class To Serialize: Trigger-ObjectTeleport (TriggerTeleportC
Localized Text: Trigger-ObjectTeleport (LocalizedText)
Serialize Class
Trigger Teleport Object To Target (Script)
When the trigger is actviated, the Source Object will be teleported to the position of the Telport Target GameObject.
Set the Source Object and Telport Target GameObjects in the RFX4_Transform Motion script.
After the teleport completes, the Source Object can be made a child of the Teleport Target using the checkbox below. This can be useful for forcing a particular orientation of an object after teleport.
Parent Teleported Obj
Trigger Delay Setting
Trigger Activation Delay: 0
Once collided with, the actions of the trigger will be delayed by this amount of time (in seconds). This is useful for, say, making a NPC wait at a waypoint a set amount of time before moving on to the next one.
Trigger Repeat Settings
Trigger can repeat: Always
The trigger can be activated repeatedly.
Cycle Colliders On Repeat
Trigger will only activate on new collisions. Items or NPCs already in the volume will not reactivate.
Trigger Repeat Delay Settings
Set the delay (in seconds) between activations when the trigger repeats. This helps reduce trigger spam if a NPC or the player keeps colliding with a repeating trigger as the trigger becomes inactive during the Trigger Repeat Delay and then reactivates once the Repeat Delay period has expired.
Defaults to 1 second.
1
Trigger Activation Percentage Chance Setting
Trigger Percent Chance: 100
The trigger will activate only a set percentage of the time. Default is 100% chance to activate. If a trigger is set to repeat, then the chance to activate is re-evaluated each repetition.
Trigger Activation Permissions
Any Collider
Player Only - Feet
Player Only - Hands
Enemy Only
Player and NPC Only
BW Trigger Defaults

-- End of Line --