

How to Jedi with the Force

By Alex Shilts, Cohort 23



G-Sidious approves your tutorial choice, but not your youngling murder count.

So, you want to make Gordon Freeman a Jedi? That's kind of weird thing to specifically want, but okay! I guess I'll run you through it.

In this How To Jedi tutorial, I'll be assuming that you have a passing understanding of the Hammer editor, namely how to **create Entities**, **convert BSP to the func_**, and **generally work with Hammers "scripting system"**. After the first few classes of LD6, you should be close enough for government work. If you aren't attending the SMU Guildhall (*why are you even here?*) and don't have those skills at least to a 3rd grade level, this tutorial may be a bit headsplodingly overwhelming. Don't get brain juice on your friends, go read more basic guides.

A quick note: Full disclosure, Hammer is an awesome but quirky editor. You will connect a lot of very logical solutions to your problems, and they won't work because of some weird way the Source engine handles things.

I also noticed how many students solve the same problems over and over again in the same way, losing the same amount of time. To help alleviate this, I'm going to have:

HIGHLIGHTED HAMMER QUIRKS! *Fanfare

If you see bold red text anywhere in a section, it indicates a strange little "thing" in the engine that a lot of people struggle with that seems obvious but is actually convoluted. Hopefully this will help!

ALRIGHT GUYS, LETS DO THIS.

Click on Image for link

So you want to be a Jedi, kid? Well, woop-de-doo.

...so do I, but in this tutorial we only focus on 4 aspects related to the Force:

1. Force Jump
2. Force Dash
3. Infinite Force Sprint
4. Force Push

*Yeah, I know you would also need a **Lightsaber**. I made one for my level and it was awesome, but also **EXTREMELY COMPLICATED** and would need more flow graphs than I'm willing to make. So just.... appreciate what you have. Greedy little...*

Making a Jedi with the Force in Hammer requires a decent understanding of 13 main entities (*and a desire to get down with the func_*):

Note: All of these entities have a great deal of applications outside of this tutorial, and will likely prove useful regardless of your project.

1. [game_ui](#)
2. [logic_relay](#)
3. [logic_timer](#)
4. [logic_case](#)
5. [trigger_push](#)
6. [logic_auto](#)
7. [math_counter](#) (*wid cute lil' sheeps*)
8. [filter_activator_name](#)
9. [point_clientcommand](#)
10. [point_template](#)
11. [env_entity_maker](#)
12. [env_physexplosion](#)
13. [filter_damage_type](#)

Comfort: If it seems like a big list, bear with me. A lot of these are very simple entities, and their uses are (on average) extremely straightforward.

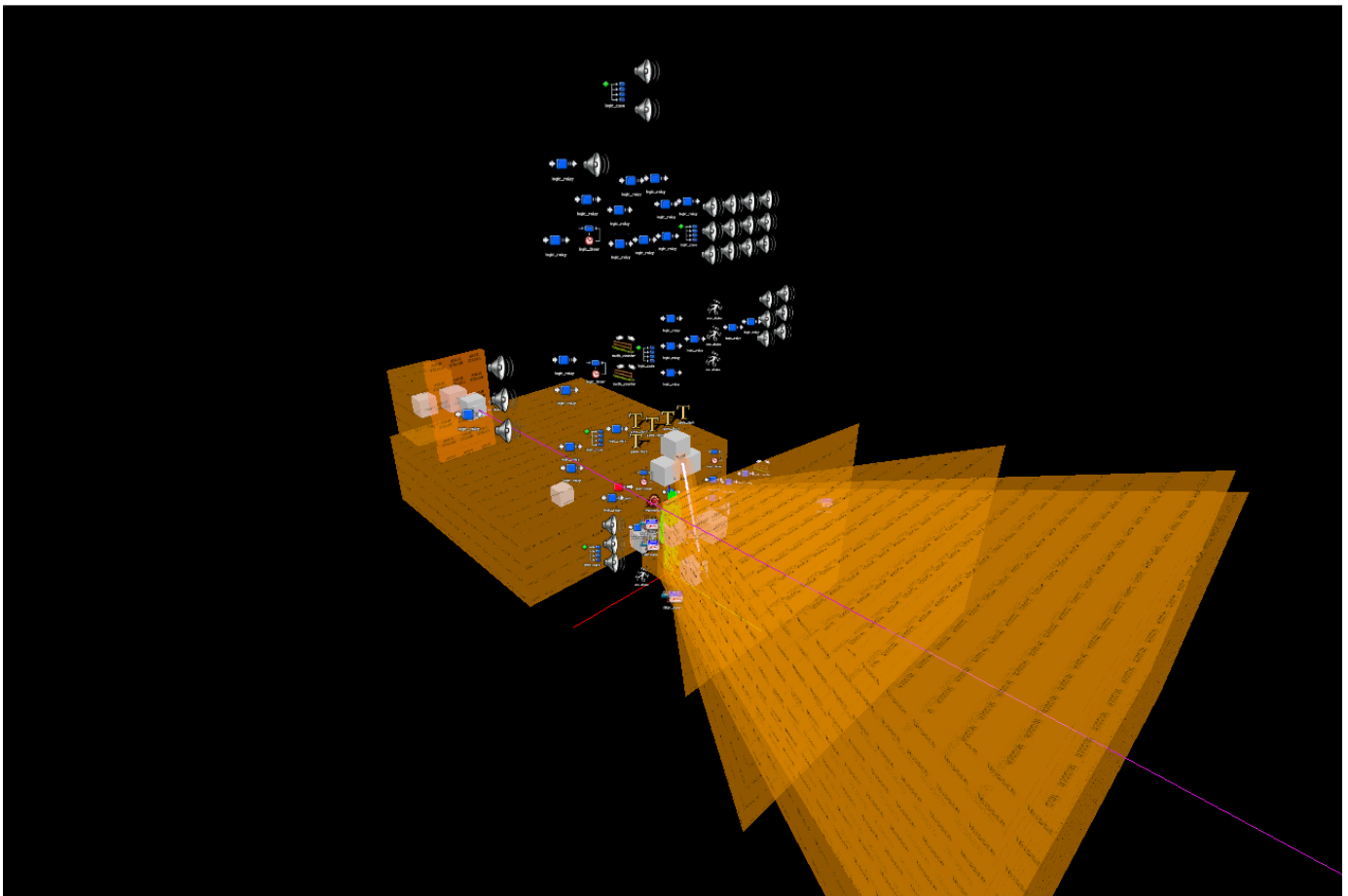
So try and familiarize yourself with these as much as possible through the [Valve Developer Community](#). Each name is hyperlinked, so do dat.

I'll try and explain as much as I can as we go, but I only have like, a day to write this thing and **AIN'T NOBODY GOT TIME FUH DAT**.

You also probably want a degree in Electrical Engineering. So you know... get on that. *I'm only kind of joking.*

-----ON TO THE TUTORIAL-----

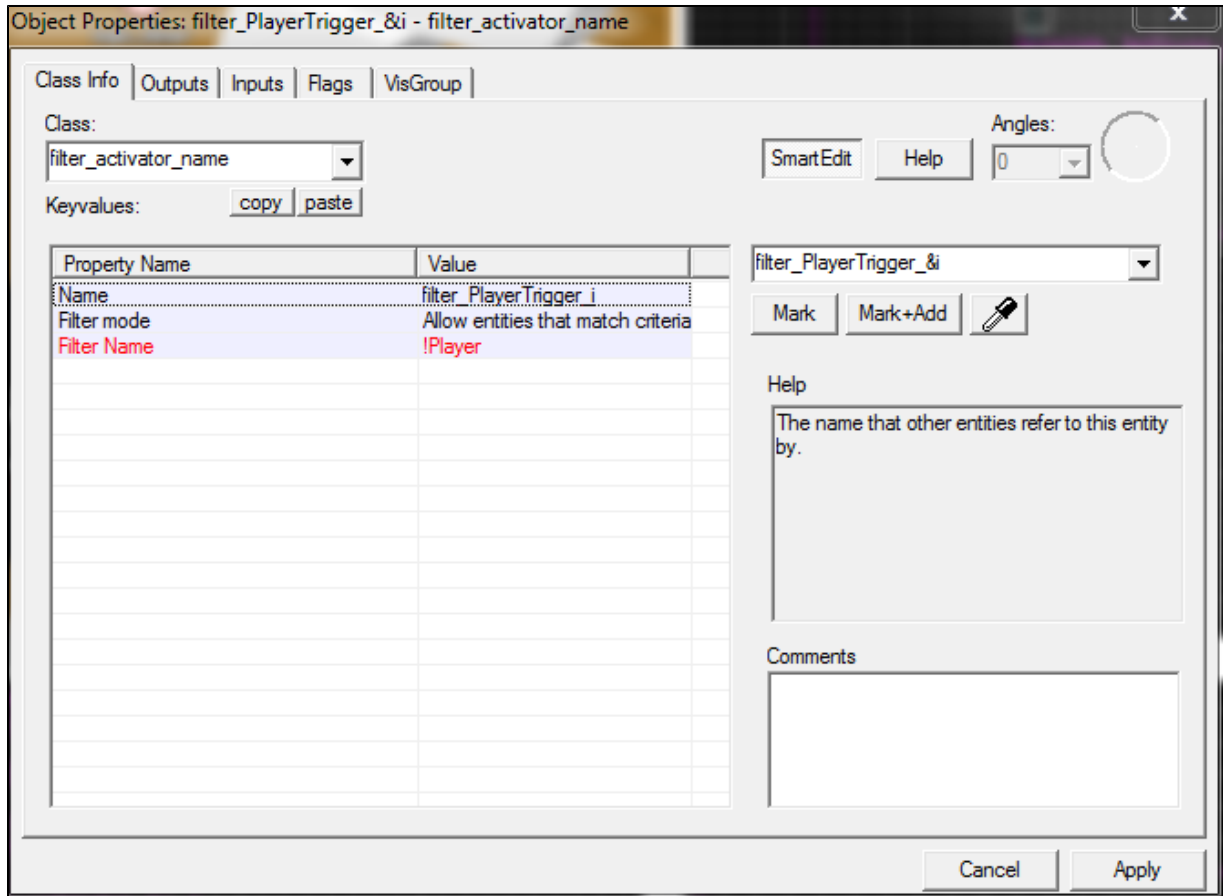
Before we start, here is a quick look at what the final player prefab looked like in my level:



While this tutorial only covers about 2/5ths of what you see, look closely at the prefab. The player start is in there, deep inside a snuggley coat of func_. As you can see, **organization is critical for this system**, as there is a lot of necessary overlap of entities and a few complex logic loops. EVERYTHING will be built around the **info_player_start** entity.

Let's get started! First up, a little preparation. These steps are useful for any project, not just this one:

1. Make a **filter_activator_name** for all of the triggers that can only hit the player.
 - a. Set the "Filter Mode" to "Allow entities that match criteria"
 - b. For "Filter Name", type in "**!Player**" (**No quotes, for the love of JEBUS. Using quotes in any dialogue box for any entity will corrupt your game. It's fixable, but yeesh.**)
 - c. **QUIRK NOTE: The player character can be referenced by any entity using the "!Player" reference name (No quotes, for the love of GOD). However, you cannot use this to parent ANYTHING to the player explicitly. To parent an object to the player, you must manually do so at run time using a logic_auto (Basically OnBeginPlay) that calls the SetParent functions on the desired entities. The entity that !Player references doesn't actually exist until you compile the game. WOO LOGIC!**



- d. This ensures that any trigger using this filter will only ever be activated by the player.
2. Make another of these filter_activator_name thingamajigs, but use this one to "**Disallow entities that match criteria**" for "Filter Name": "**!Player**"
 - a. This serves as your filter for all triggers you want to ignore the player completely. Its Class Info page is very similar to the above image.
3. Get yourself a brand new **game_ui**, the most beautiful entity in the... room.
 - a. If you set the player as the activator using a **logic_auto**, you can call inputs using the main Hammer control events, **including Left and Right mouse click**.
 - b. In the **logic_auto**, send the "Activator" input to the **game_ui** with the "**!Player**" Parameter
 - c. **Quirk: The "Pressed-" and "Unpressed-" outputs from the game_ui represent most if not all of the available KeyDown and KeyUp events that you can actually distinguish. If you want different things to happen on keyDown and keyUp, you must use these inputs from a game_ui. Everything else just calls the keyDown twice.**
4. If you plan on doing any custom key bindings or any fundamental changes to the player character, make yourself a **Config File**.
 - a. These are pretty easy to use, and use very basic commands to do some really cool things.
 - b. Head to wherever you installed Half Life 2 (***cough cough SteamLibrary\SteamApps\common\Half-Life 2\ cough***) and find **le p2lcfg**. If the folder isn't there, MAKE ONE. YOU ARE A STRONG PERSON WHO DON'T NEED NO PRIOR FOLDER STRUCTURE.
 - c. In this folder, make a basic text document (**Right click -> New -> Text Document, jeez**)
 - d. Name it **LastName_WhateverTheFlipYouWant.cfg**. **The file extension .cfg is the important bit.**

- e. **Quirk: You can use quotes in these documents, so any command you want to use that needs quotes should go in here. There won't be that many that aren't key bindings, so don't freak out.**
- f. In this file, to make entities fire inputs when you press a key, use the format
 1. **bind <key> "ent_fire <name of entity> <name of input>"** (Only use quotes if you have spaces in the binding argument. e.g. "**bind q "ent_fire Jedi_UI_* FireUser2"**" but not **bind e +use**)
5. Make yourself a **point_clientCommand** and drop that shiz in your level ASAP. This will serve as the ferryman between you and the *underworld realm of the inner Source engine*.
 - a. If you just made a config file, fantastic! You can now use a logic_auto to call the **Command** input with the Parameter **exec <name of Config File>**
 - b. On startup, all of your bindings will be executed and your controls will function!
 - c. **Sort of a Quirk, but mostly housekeeping: You will need to create another Config file that unbinds all of your keys and restores the control defaults. You will call the exec input for this new config from the trigger that ends your level, so the bindings don't persist between games. Professor Ouellette has a number of pet peeves concerning leftover bindings, so don't be that LD.**

Enough boring, let's exciting!

First things first, let's get Gordon up and moving like a 2 Liter Jedi Coke bottle full of Darth Mentos!

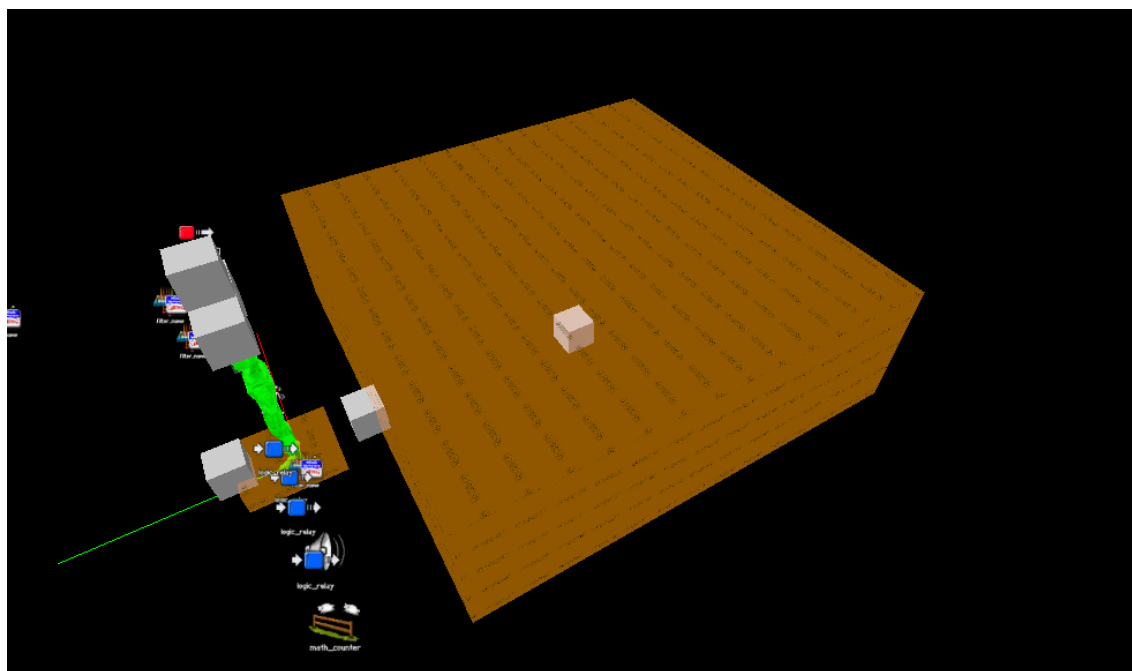
As stated earlier, for my version of Jedi Freeman, I gave him three cool ways to Force himself through the environment:

- **Force Jump**
- **Force Dash**
- **Unlimited Sprint**

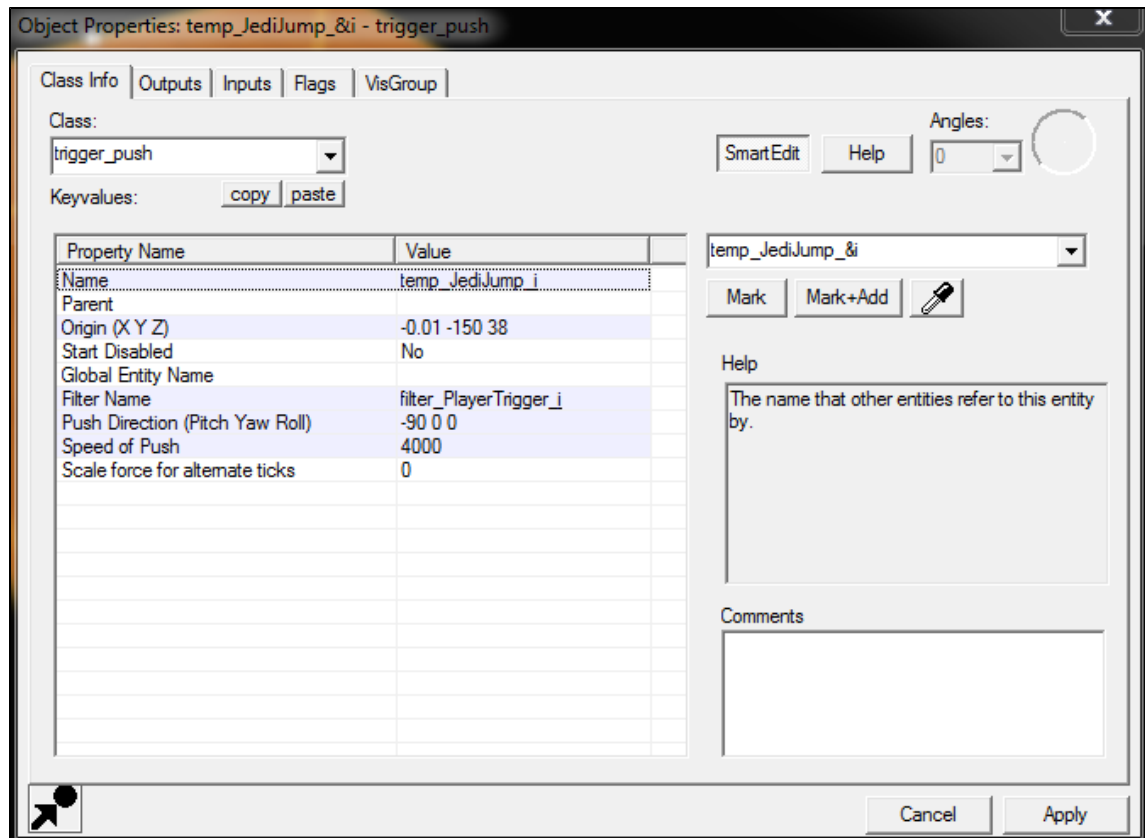
Let's break them down with the func_

Force Jump

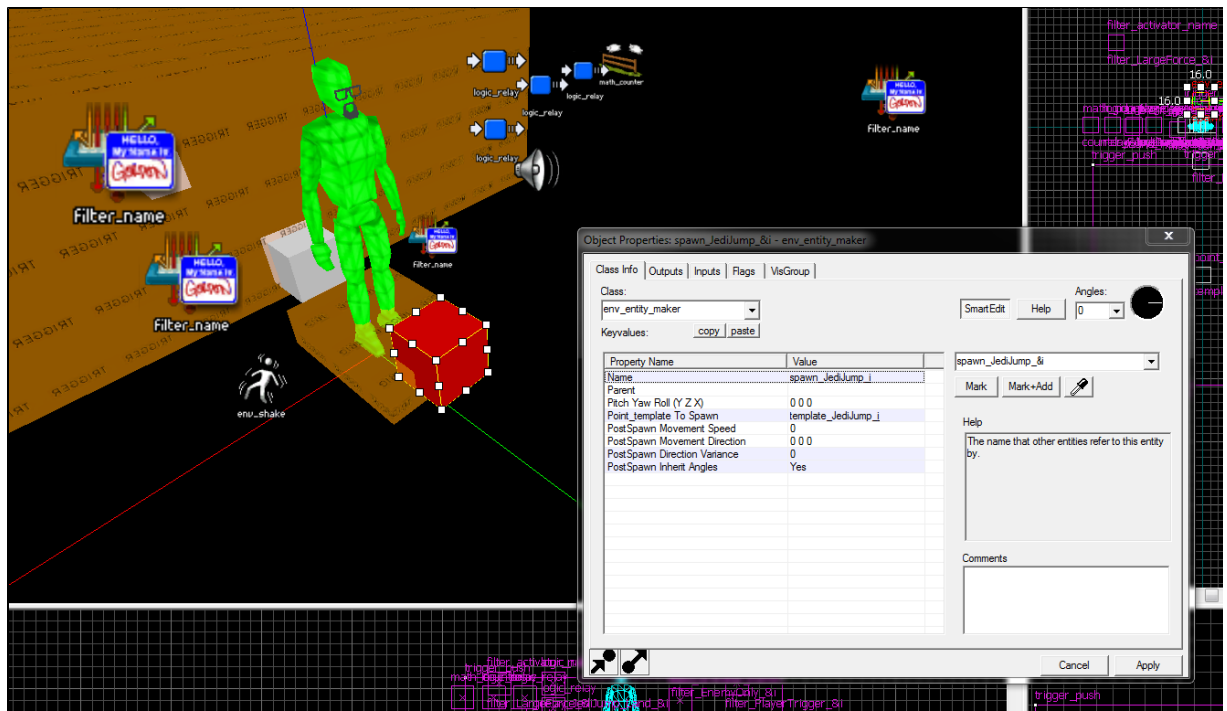
1. Okay, so the Force Jump sound easy enough, right? I'll just make Gordon jump higher! **Hehehe so cute.**
2. First thing you need a **trigger_push**. A **trigger_push** is a converted BSP, so you can make it any shape you want before you convert it to an entity (Ctrl+T). Remember these. Push Triggers are big time super nice for Jedi.
 - a. In this case, I made the trigger very wide as this makes it more effective (**you'll see why in a minute**), about **256 x 256 x 64**. Shown below:



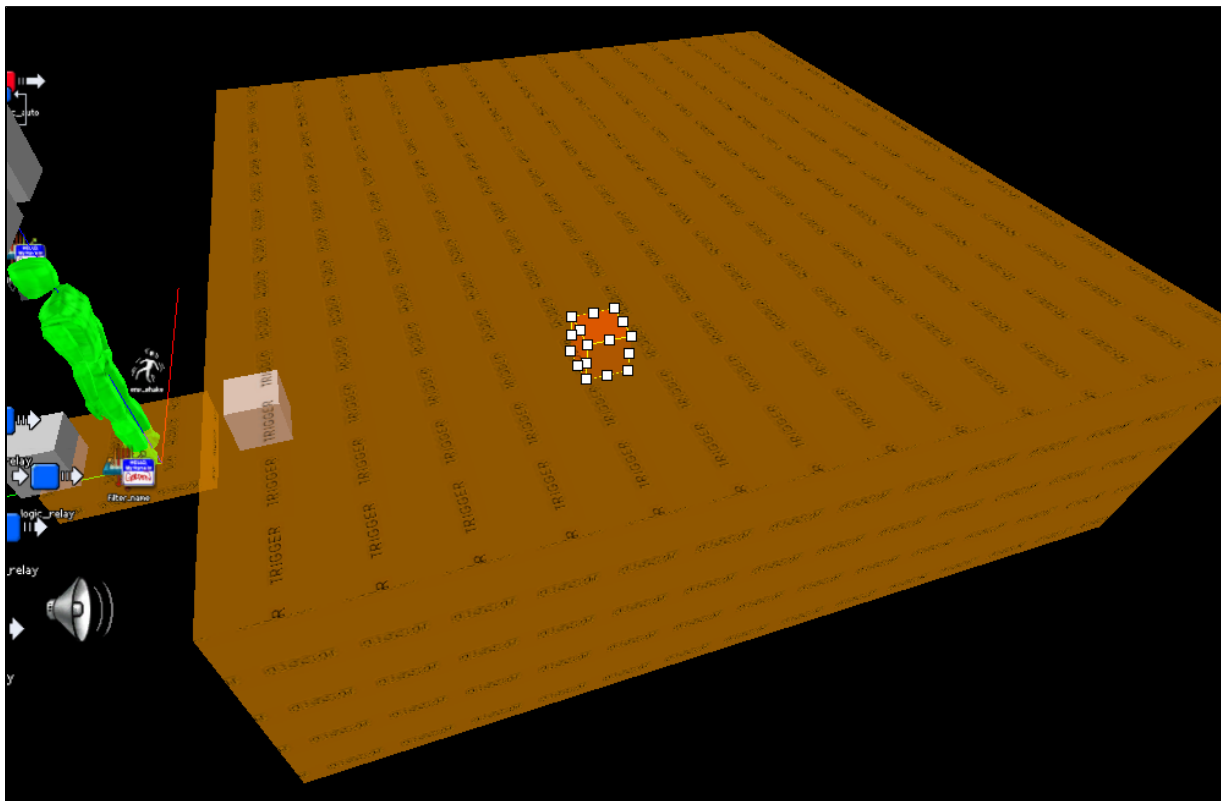
- b. For this to jump the player to a decent height, you'll want these values:



- c. Pop that **Player-only filter** you made earlier into the "**Filter Name**", so these triggers will only affect the player. That is, unless you want every physics object near the player to fire upwards like a bullet every time they jump.
 - d. A "**Speed of Push**" of about **4000** serves as a decent jump height. **Henceforth, all force units in Hammer shall be referred to as "Hammers"**. This trigger applies **4000 Hammers** to the player.
 - e. **QUIRK ALERT: There is no "Impulse" ability for a push trigger. You can get one from an env_physexplosion, but it can be extremely unpredictable, as we will see later. For vertical forces (and ONLY vertical forces), you want a trigger_push that starts disabled, and you quickly enable and disable with a very short delay (usually around 0.1 seconds).**
 - f. Alright, so you have a push trigger, and you want to push the player. Well, too bad. There are more steps. This next step in particular has a lot of quirks. Like... a murder happened there or something.
 - g. **UNCOMFORTABLE AMOUNTS OF QUIRKS: You are immediately going to think, "Oh, I'll just parent this trigger to the player and tie the enable delay to a relay that I bind to my config file! I'm super smartastic!" You are wrong.**
 - i. **QUIRK: Triggers, being annoying little brats, refuse to affect pretty much anything in their parenting hierarchy, especially the player. Parenting any trigger to the player removes all effects. So how do we fix this?**
 - ii. **QUIRKY SOLUTION: Don't parent it to the player, ever. We get to SPAWN A TRIGGER EVERY TIME WE PRESS THE SPACE BAR. I know, its great.**
 - iii. **OTHER QUIRK: No push trigger, or potentially any direction-based entity for that matter, inherits the parent's rotation in terms of their forces when spawned. EVER. They spawn with their default template orientation and you just get to deal with that. However, for vertical forces, we can still work with this. Since Jump always goes up, we don't give a flying fippidy-doo-da that Hammer is a gibbering psychopath using its best friend's underwear and some tinsel as a fancy hat.**
3. So, now that all of that is off of my chest, lets spawn a push trigger! First up, you need an **env_entity_maker**. Below is about where I placed the thing relative to the player spawn. Since we can't parent the trigger, and we've made it **VERY WIDE (see, told you it would make sense. Yeah.)** Placing the maker in front of the player is easy to see and ensures that even a fast moving player will still be in the trigger during its lifetime.



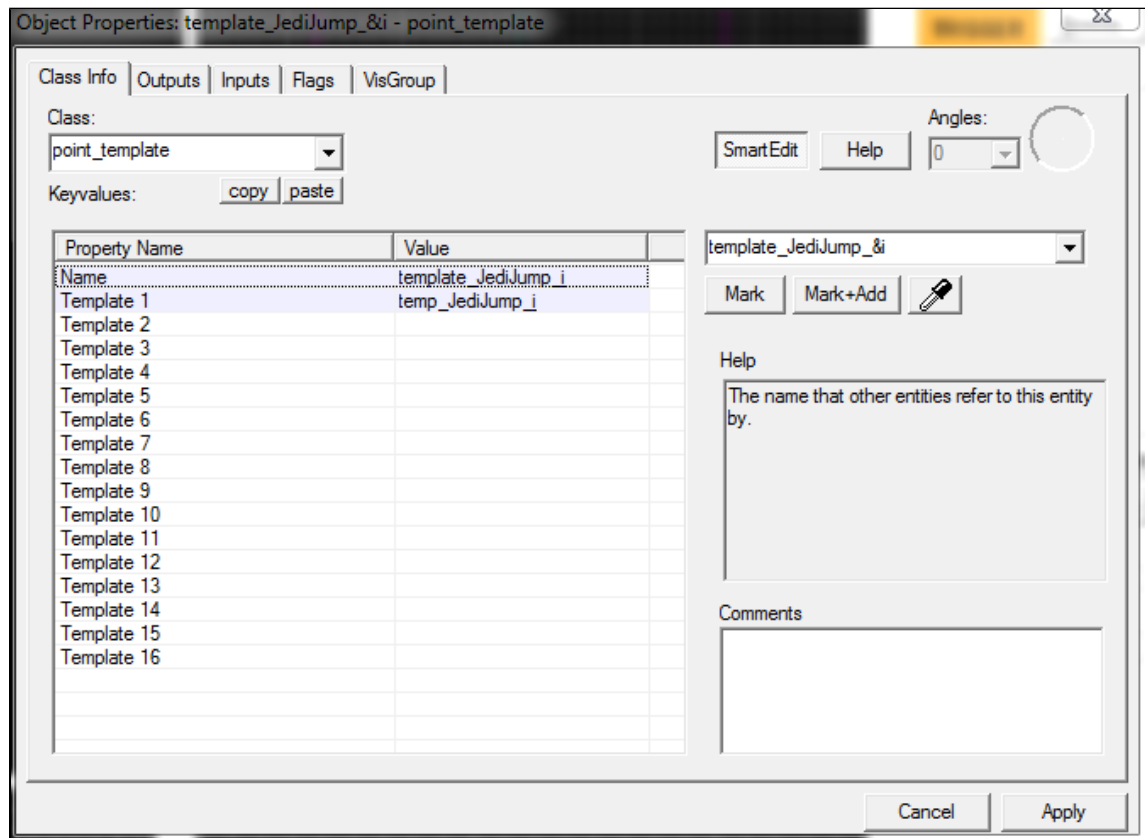
4. Now, we place a **point_template** at the center of the jump trigger we placed, close to the floor.



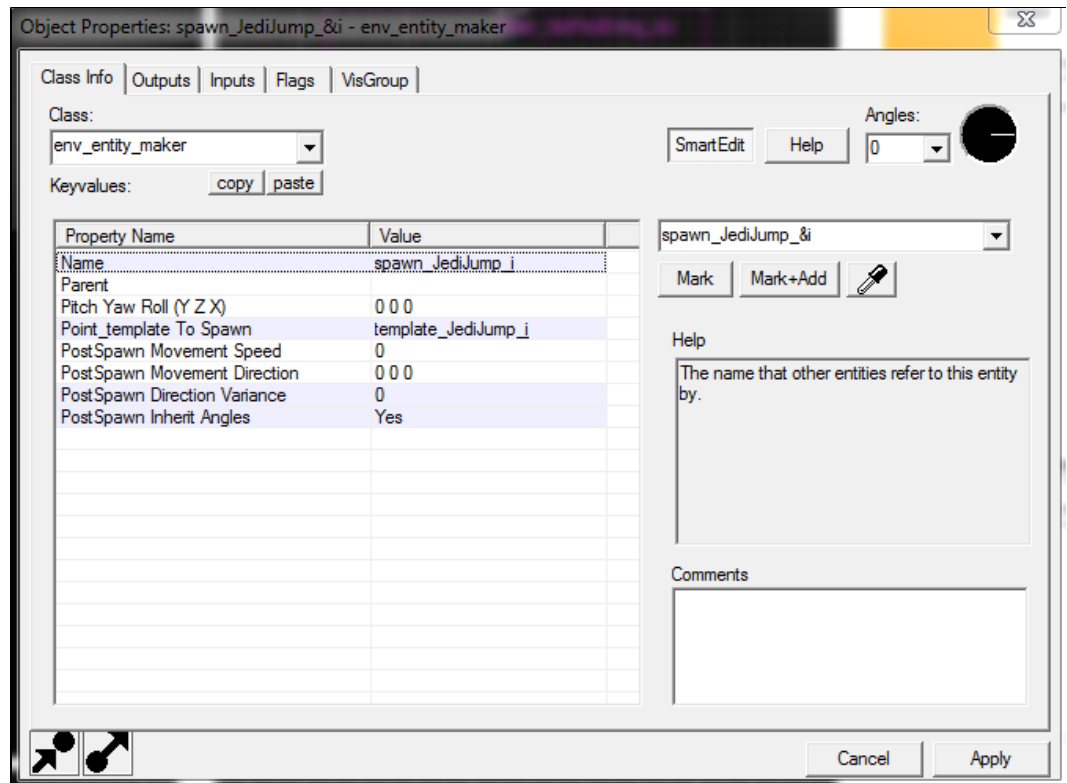
5. This is going to serve as the- oh wait, hang on...
 - a. **QUIRK ALERT:** The location of the point_template will serve as the origin point for the spawned version. The entity_maker will spawn the new push trigger using itself as the "new origin" of the point_template push trigger. Still confused? Yeah, me too. Basically, try and imagine that the point_template is actually the entity_maker, and then try and imagine where the player will be relative to it. You want the template close to the floor so the player spends as much time in the trigger as possible.
6. Now, let's link these crazy kids together!
 - a. First off, name the entities. Always name your entities. **You can't reference them from other entities otherwise. The**

editor will just silently fail and laugh at you and your stupid hair.

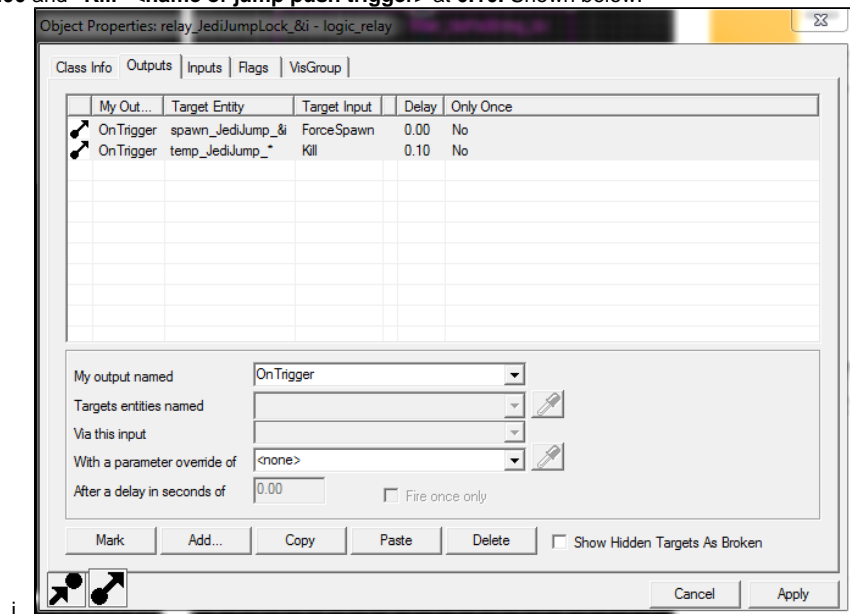
- b. Assign the push trigger to one of the many empty template slots in the point_template. Just pick one. Literally any one. I wish I could tell you it mattered. Be a rebel and use Template 16. Shown below:

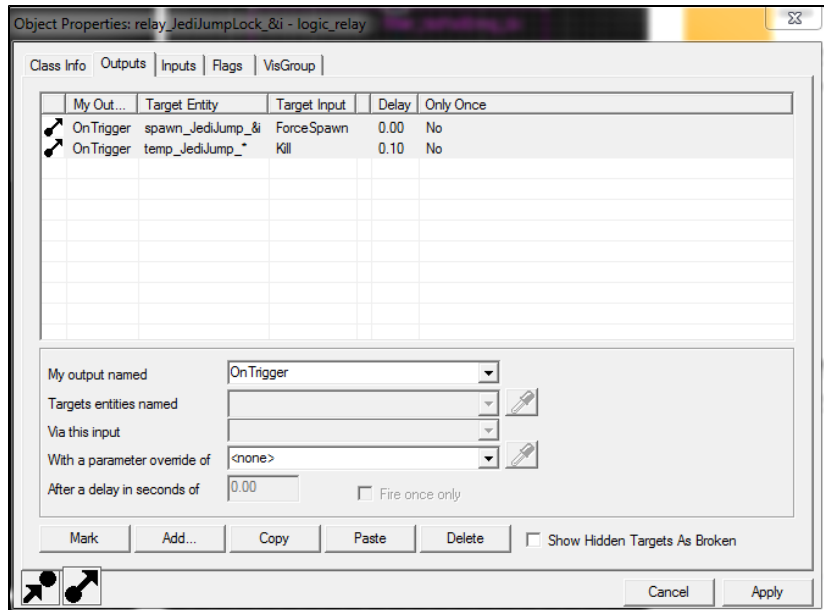


- c. Link your **env_entity_maker** to your shiny new **point_template** in the "Point_template To Spawn". Shown below:
- i. **Go ahead. Change "PostSpawn Inherit Spawn Angles" to "Yes". You're probably thinking, "Hey, maybe they updated this entity since this tutorial! I bet this guy is really disappointed! Lol bad luck bro." Well, for your information BRO, all triggers inherit the player's spawn direction for their volume shapes, but the directions they apply their forces stay global. And Hammer hasn't been updated since 2005. So shush.**

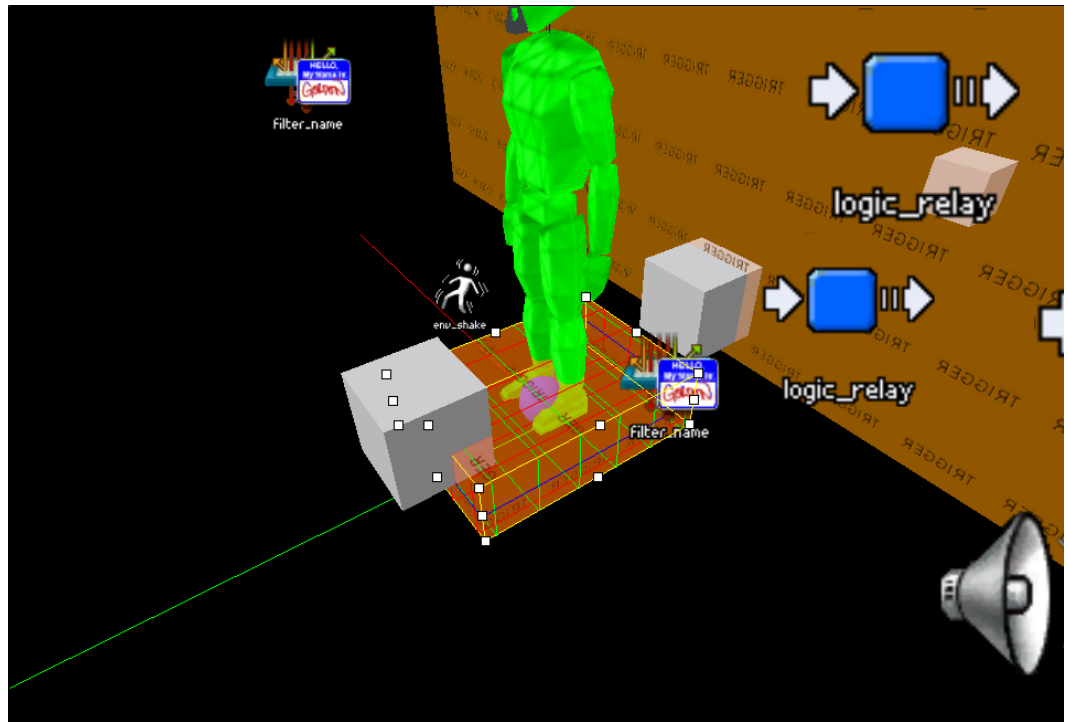


7. Next up, we need to bind the jump button to spawn the Force Jump push trigger.
 - a. Create two relays: One that sends the "**Force Spawn**" input to the **entity_maker**, and another one that toggles the first one. Why do we do this?
 - i. **QUIRK: Key bindings in Hammer, at bare minimum concerning the almighty Space Bar, are called on both keyUp and keyDown. This leads to some weird duplicate effects in an already hard to control system. So toggle that shizzle.**
 - ii. The **logic_relay** that DOES THE TOGGLING needs to send the "**Toggle**" input at **0.00** and "**Trigger**" the other relay at **0.01**, or at least some small amount of time greater than 0. This makes the actual "Jump" event only happen on Space Bar Down, which feels fairly natural. The relay BEING TRIGGERED should "**ForceSpawn**" to the **entity_maker** at **0.00** and "**Kill**" <name of jump push trigger> at **0.10**. Shown below:

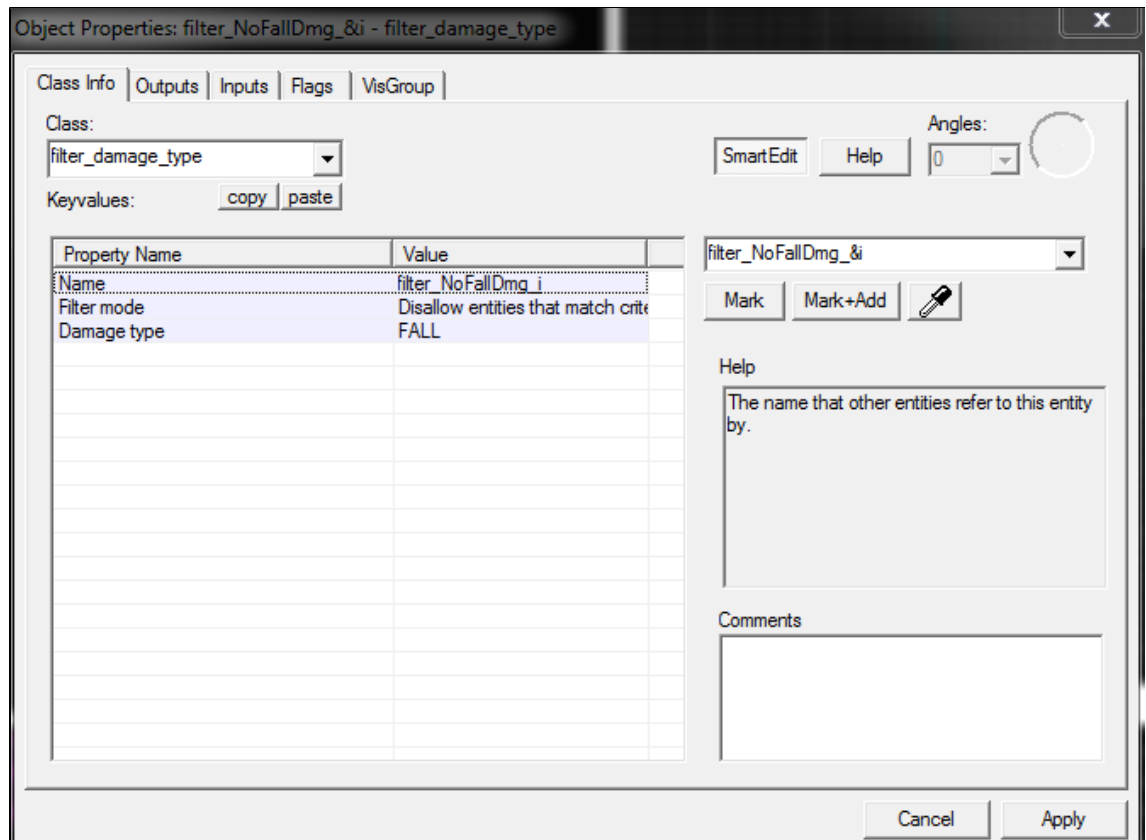




8. Next, **ONTO THE CONFIG FILE!!** Assuming you wish to follow the PC control convention established at the dawn of time by Computer Jesus, add the line **bind space "+jump; ent_fire <name of jump trigger that toggles> Trigger"**
 - You'll notice that we bound both the relay trigger input **AND** the standard Jump command. Hmm...strange no?
 - i. **QUIRK: The Player will not be properly affected by push triggers until "Activated" by the Havok physics engine. This means that unless you slap Gordon in the face with an office chair every time you want to jump, you need to get him off the ground to ensure proper force application. Binding the +jump event to the relay input ensures that the player is airborne when we enable the push trigger. Wait, airborne seriously has an "e" at the end? Well, I've been saying it wrong.**
9. Alright, so now (hopefully) we have:
 - a. A **trigger_push** (that, as a template, will be deleted on game start, so no worries on where you place it)
 - b. A **point_template** that references the template Push Trigger
 - c. An **env_entity_maker** that spawns the push trigger from the **point_template**
 - d. Two **logic_relays** that toggle to remove duplicate key presses and call the "ForceSpawn" input on the **env_entity_maker**
 - e. A **key binding** in our **config file** that calls the toggling **logic_relay** on **Space Bar Down**.
10. Now, test it out! When you press Space Bar, you can now get a nice little boost into the sky. Here is where you can play with the values and find what feels best for you. Here are the major tweak points:
 - a. Adjusting the delay until Kill gives the player more time to be pushed.
 - b. The height of the Trigger keeps the player inside the trigger volume for longer, a similar effect to the delay until Kill, but less limited. You want to adjust both for more jump power.
 - c. Adjusting the Speed of Push applies more force while the player is in the trigger, and FYI it accumulates per frame
11. **But what's this?** You can jump **FOREVER** and eventually punch the Sun in its ugly Skybox face?!
 - a. **QUIRK: "Oh, well I'll just make it to where the player can only jump when they are on the ground. Good thing I always think things through!" No. I am rapidly losing hope that you can be saved.**
 - i. **There is no currently known way to backpack off of the built-in jump restriction from Source games. The player somehow magically knows when they are on the ground, but you never will. Unless you:**
 - b. **COAT THE WORLD IN A THICK, FILMY LAYER OF TRIGGERS THAT RE-ENABLE THE TOGGLING LOGIC_RELAY AND MAKE YOU GET ALL STICKY!**
 - i. That's right. Remember all of that free time you thought you had? NOPE.
12. **Nah, its actually not that bad.** Here's how you do it:
 - a. Have the **logic_relay** that spawns the jump trigger send the "Disable" Output to the Toggling relay when it gets triggered.
 - b. Make a **trigger_multiple** out of **BSP**, about **8 units high** (width doesn't matter), assign it the **Player-Only filter** you made earlier, and assign the "OnStartTouch" Input to "Enable" the Toggling Relay.
 - i. Also set the "Delay Before Reset" to 0.1. Otherwise the re-enabling feels sluggish and unresponsive.
 - c. Shift+Click this base **trigger_multiple** like a **BOSS** across every **BSP surface** in your level the player can reach. *It goes faster than you think. (Be smart and put them in a separate Visgroup. It comes in handy later.)*
 - d. For Mesh surfaces, you can actually parent a small trigger to the player's feet. But what's this?
 - i. **QUIRK:** Just kidding, this one actually works pretty well.
 - ii. Make a small **trigger_multiple** and **place it at the player's feet**. Assign it the **Non-Player filter** you made earlier, and give it the same logic as the standard reset triggers that coat your level.
 - iii. For this trigger, as stated earlier, use a **logic_auto** to parent this to the player. Typing it manually won't work because *Hammer*.



- iv. **HAHAHA THERE WAS A QUIRK:** This is a Hammer-wide issue that you have no control over, but the camera and the player model (aka the !Player parenting reference point) rotate at different rates. This leads to some WEIRD angular movement of anything you parent to the player when you pan the camera vertically. Gordon actually just inhumanly leans back like a First Grade Neo awkwardly dodging slo-mo spitballs. In this context, the trigger will only work PERFECTLY when you are looking straight ahead. The longer you make it back to front, the less of a problem this is, but it also becomes oddly exploitable (you can get a bunch of free jumps by jumping next to a wall with meshes vertically inside it because the trigger reaches so far in front of you).
- 13. The final element you need for this amazing jump is to remove all fall damage. You will die SO EASILY now that you can jump three times as high as normal. I know, weird.
 - a. Add a **filter_damagetype** entity to your map, doesn't matter where, but (as seen above, its the little grey box behind the player) I placed mine behind the player for organization.
 - b. Set the "Filter Mode" to "Disallow entities" and the "Damage type" to "FALL". Shown below:

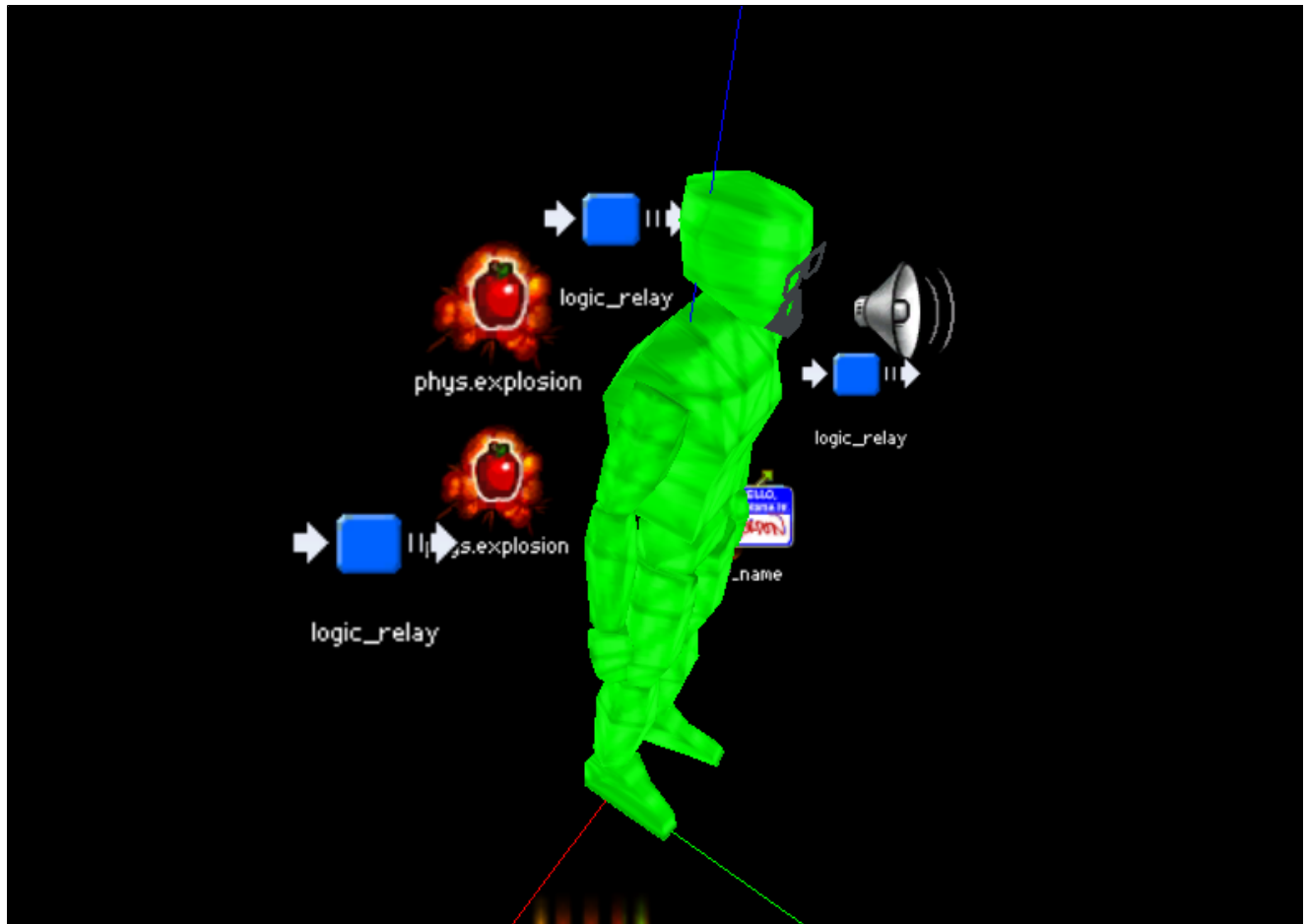


- c. Last, in a **logic_auto** entity, send a "**SetDamageFilter**" input to "**!Player**" with the **Parameter** as **<name of your fall damage filter>**
- d. And your slightly incompetent Jedi is now immune to all fall damage. You'll still hear his shins juicily snap when he lands though. *So, you know. That's fun.*
14. Once you have your safeguards against infinite jumping and the numbers feel tweaked to your flavor, you have successfully implemented a super jump. What's that? You have very little air control?
 - a. **What editor did you think you were working in? But that is a nice SEGWAY TO:**

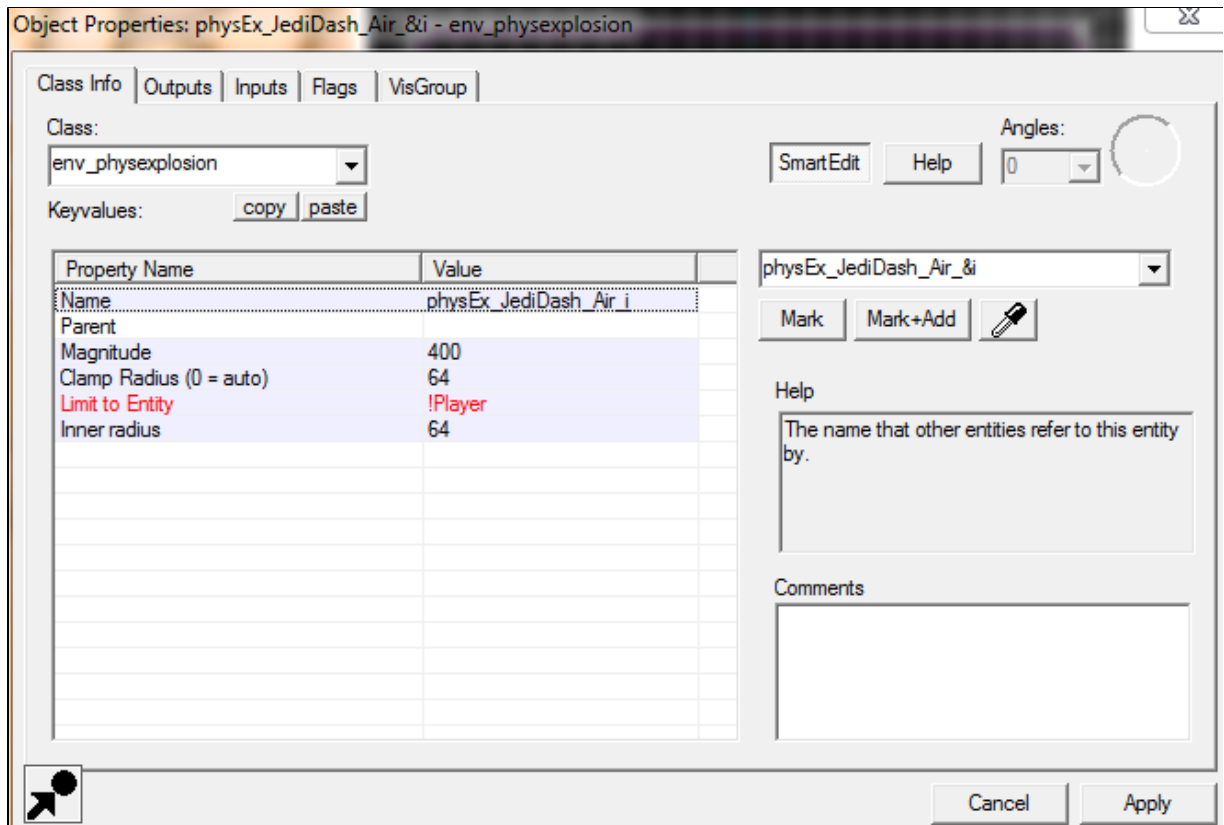
Force Dash

This one is much easier than Force Jump, I promise. It's also hilariously hard to control. So its fair.

A good physics impact force that actually inherits the players desired movement direction can only be found in the mysterious `env_physexplosion` (*phys explosion, not phy sexplosion*). My final arrangement is shown below:



1. You will notice that there are two **env_physexplosions** behind the player, and two separate **logic_relays**. What's the deal with that?
 - a. **QUIRK: When the player is grounded, an env_physexplosion applies a much more muted force. This is likely a similar issue to the need to jump before triggering the jump push_trigger. To have a consistent distance between grounded and mid-air dashes, you need two separate entities.**
 - b. So, create two separate physexplosion entities. Make sure they are close enough that the player is within their "Inner Radius" The Class Info will look similar to below




- c. For the mid-air **physexplosion**, you want a relatively low "**Magnitude**". Even **400** is a really powerful dash with the right amount of momentum. **These entities apply a really strong impact force that scales drastically (at least, in mid-air) with how fast the player is already moving.**
 - d. As shown above, I chose to limit the explosion to only affect the "**!Player**". This is not required, and really doesn't change the performance. However, the impulse is radial, so if you leave this zone blank you can have physics objects explode away from you when you dash. *Just saying.*
 - e. You can duplicate this entity and change the "**Magnitude**" to a much higher number. My ground-splosion emitted about **1200 Hammers** of force, but please, **tweak these numbers to your liking**. They will most likely have a difference of around 1000 between each other.
 - f. Don't forget to parent them to the player with a **logic_auto!** (**Since that's the only way you can do it**)
2. You'll then want two **logic_relays** that individually trigger the separate **physexplosions**. They will look similar to this, without the "**playSound**" Output (*Unless you already have sounds ready. Aren't you a trooper*):

Object Properties: relay_JediDash_Floor_&i - logic_relay


Class Info | **Outputs** | Inputs | Flags | VisGroup

	My Out...	Target Entity	Target I...	Delay	Only Once
⚡	OnTrigger	physEx_JediDash_Floor_&i	Explode	0.00	No
⚡	OnTrigger	sound_ForceDash_&i	PlaySound	0.00	No

My output named:



Targets entities named: 

Via this input:

With a parameter override of: 

After a delay in seconds of: ☐ Fire once only

☐ Show Hidden Targets As Broken

- a. You'll want the relay responsible for the mid-air dash to start disabled.
3. Next, you need a main relay that will interact with your config file. This relay will look something like this:

Object Properties: relay_JediJump_InAir_&i - logic_relay

Class Info | **Outputs** | Inputs | Flags | VisGroup

	My Out...	Target Entity	Targ...	Delay	Only Once
	OnTrigger	relay_JediDash_Air_&i	Toggle	0.00	No
	OnTrigger	relay_JediDash_Floor_&i	Toggle	0.00	No

My output named:

Targets entities named:

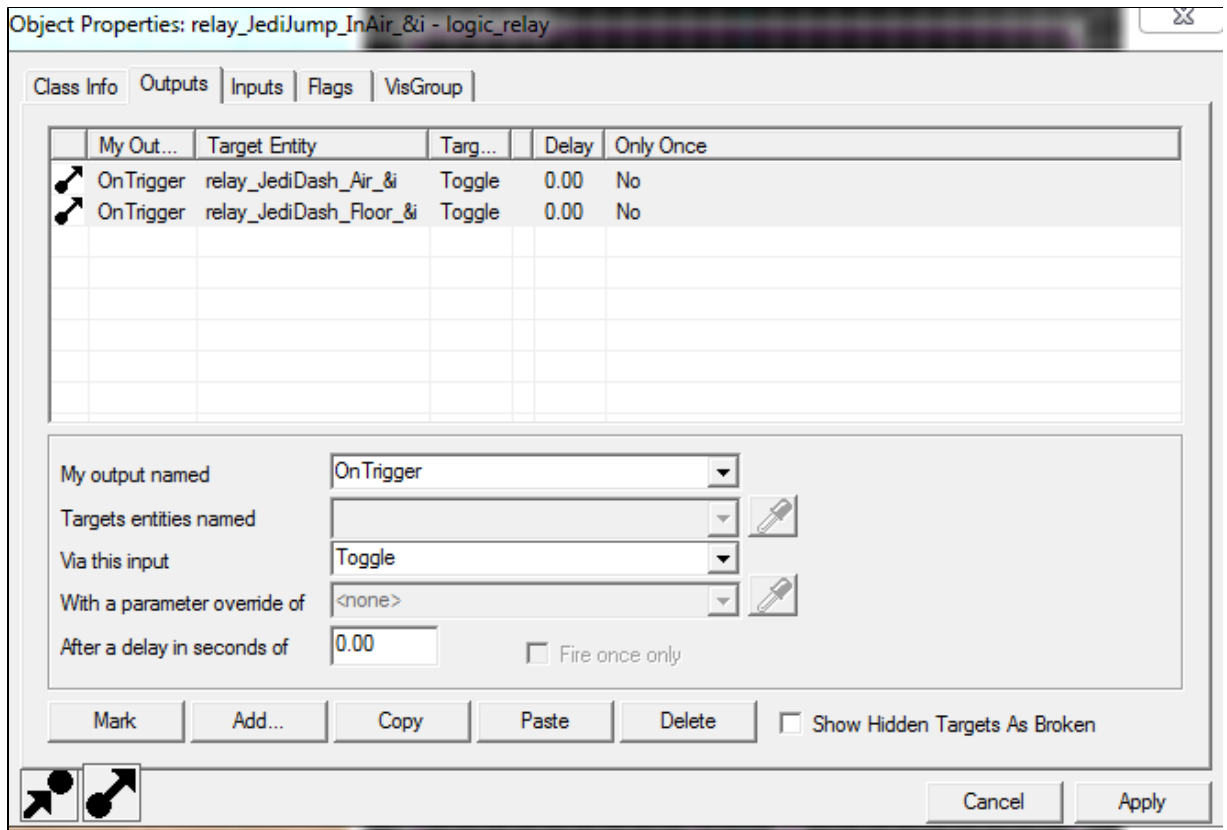
Via this input:

With a parameter override of:

After a delay in seconds of: ☐ Fire once only

☐ Show Hidden Targets As Broken

- a. This relay fires both dash relays every time, and has a built in self-disable cool down of 3 seconds. Again, **tweak this number to whatever feels best for you.**
2. By now, you are probably wondering: "**Wait a minute, how do the relays know when to alternate between each other to change the Dash intensity?**" *You just solved that problem in the previous tutorial section! What's that, you just skipped straight to the Dash? Lol*
 - a. If you skipped the previous section on creating a force jump system, revisit it and skip to the part where you **coat the world in triggers to re-enable the jump relay. This system is exactly what you need.**
Just have the triggers send a **Trigger** output to a special relay that **toggles both Dash relays**. It will look similar to below:



- c. Call this on the "StartTouch" and "EndTouch" Outputs for each floor trigger. You can bulk select all of them (in that Visgroup that you responsibly created, right?) and add these outputs all at once. No seriously, its that easy.
3. Finally, just add the key binding to your Config File. The line, again, will look like: **bind <key> "ent_fire <name of Dash Activate Relay> Trigger"**
 - a. I chose "Q" as my dash key due to its *close proximity to WASD and lack of use in Vanilla Half-Life 2*, but pick your own poison.
4. **What's that? The impulse force doesn't update its orientation in mid-air until you land? MUAHAHAHAHA seriously let me know if you solve that. It's really annoying.**

Infinite Force Sprint

Full disclosure, this one is a joke. Seriously, *its barely even a section*.

It doesn't even deserve a numbered list.

It needs no pictures, as there are less than 1,000 words needed to describe it.

You need a **point_clientcommand** and a **logic_auto** (If you don't have these, make them. Forge your own path of RIGHTEOUS LOGIC).

In the **logic_auto**, send the **point_clientcommand** the Output: **command** with the *Parameter* **sv_infinite_aux_power 1**

There you go. **Be sure to set it back to 0 in your Unbind Config File.**

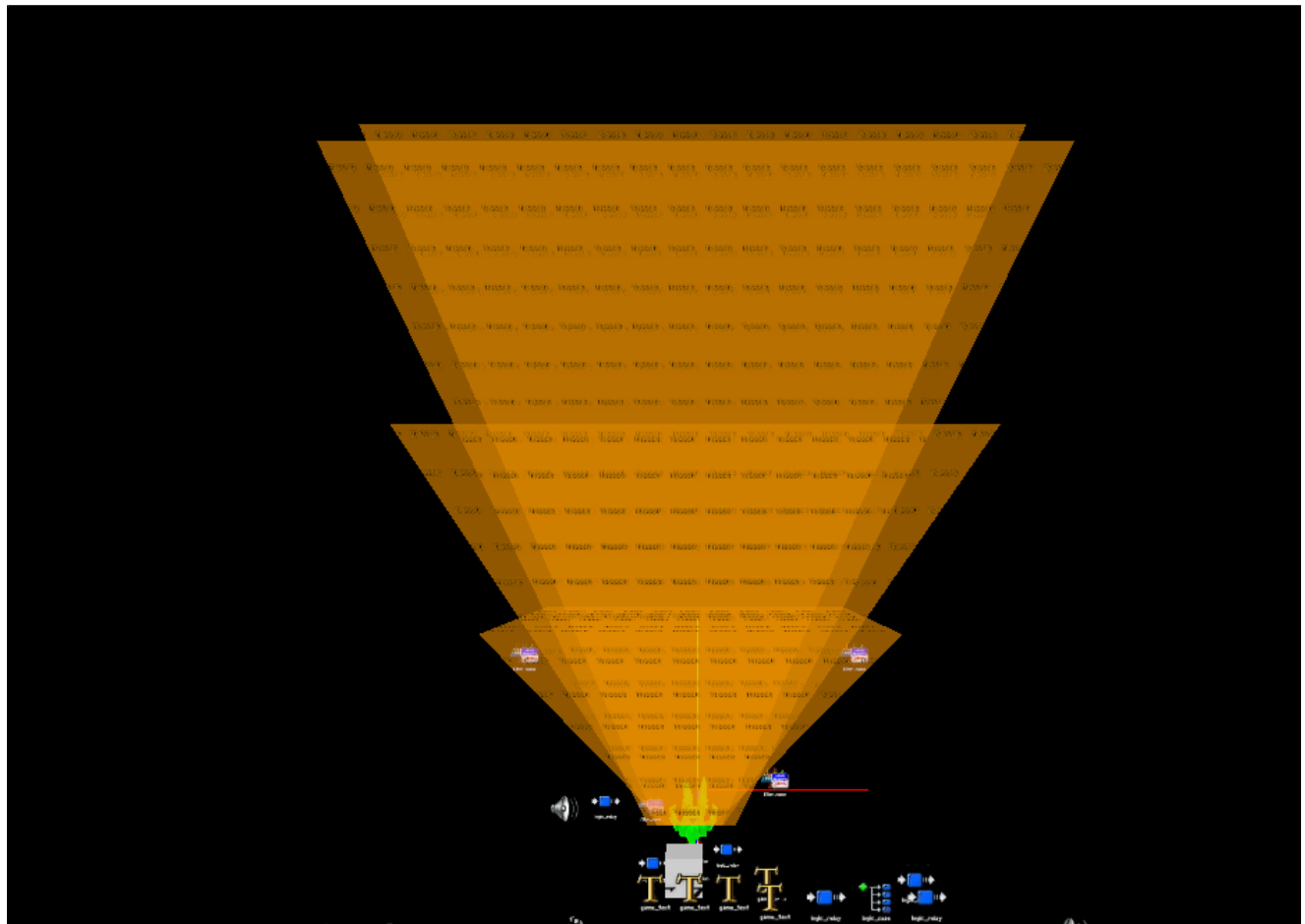
Force Push

Here be Dragons, friend. Not as many Dragons as the **func_brush Lightsaber** tutorial I chose not to write, but still. **BEWARE.**

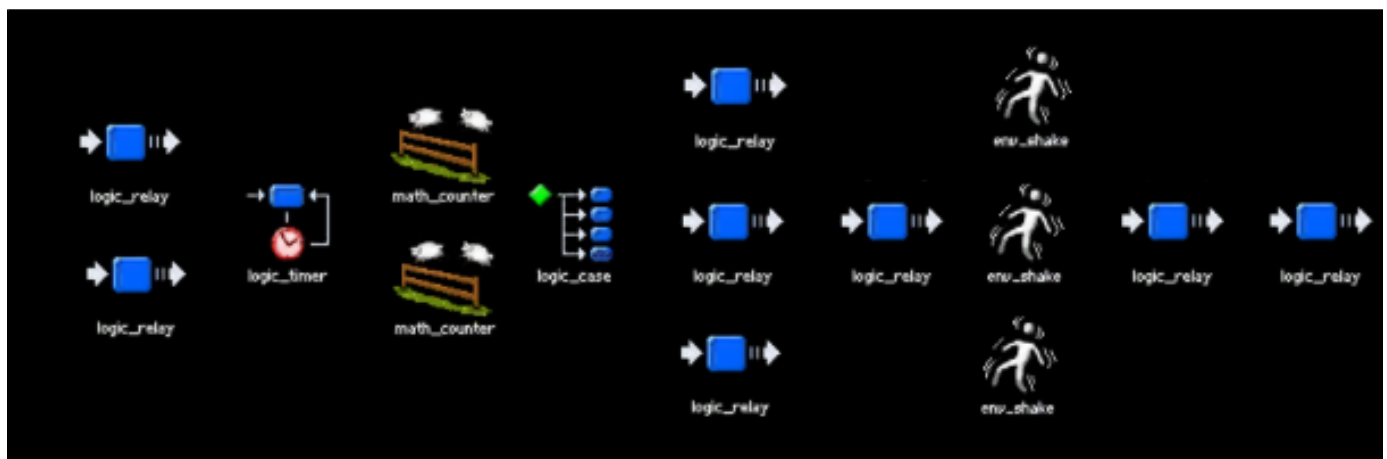
Now that we've gotten the player jumping and dashing around like a Super-Sayan monkey with ADD-Rabies, let's give them a weapon. Here, we will give the player a multi-stage, chargeable Force Push that gives the player a great deal of impact on the game world. **Half Life 2 is a very Physics heavy game, and it is wise to take full advantage of that.**

The difficulty of this concept depends on how far you want to go with it. In this tutorial, I will cover a basic triple-tier force push with a charge timer. You can expand this to as many levels, strengths, and crazy other applications your crazy eyes desire.

- This funnel cake of madness is the final layout of my 3 main Force Push triggers (technically there are 4 in there, but the last one was special and isn't relevant). I chose the increasing size of each trigger to correspond with a larger push force and effect range. *It also looks a lot like a Dragon Shout, so at least two birds were killed in this process.*



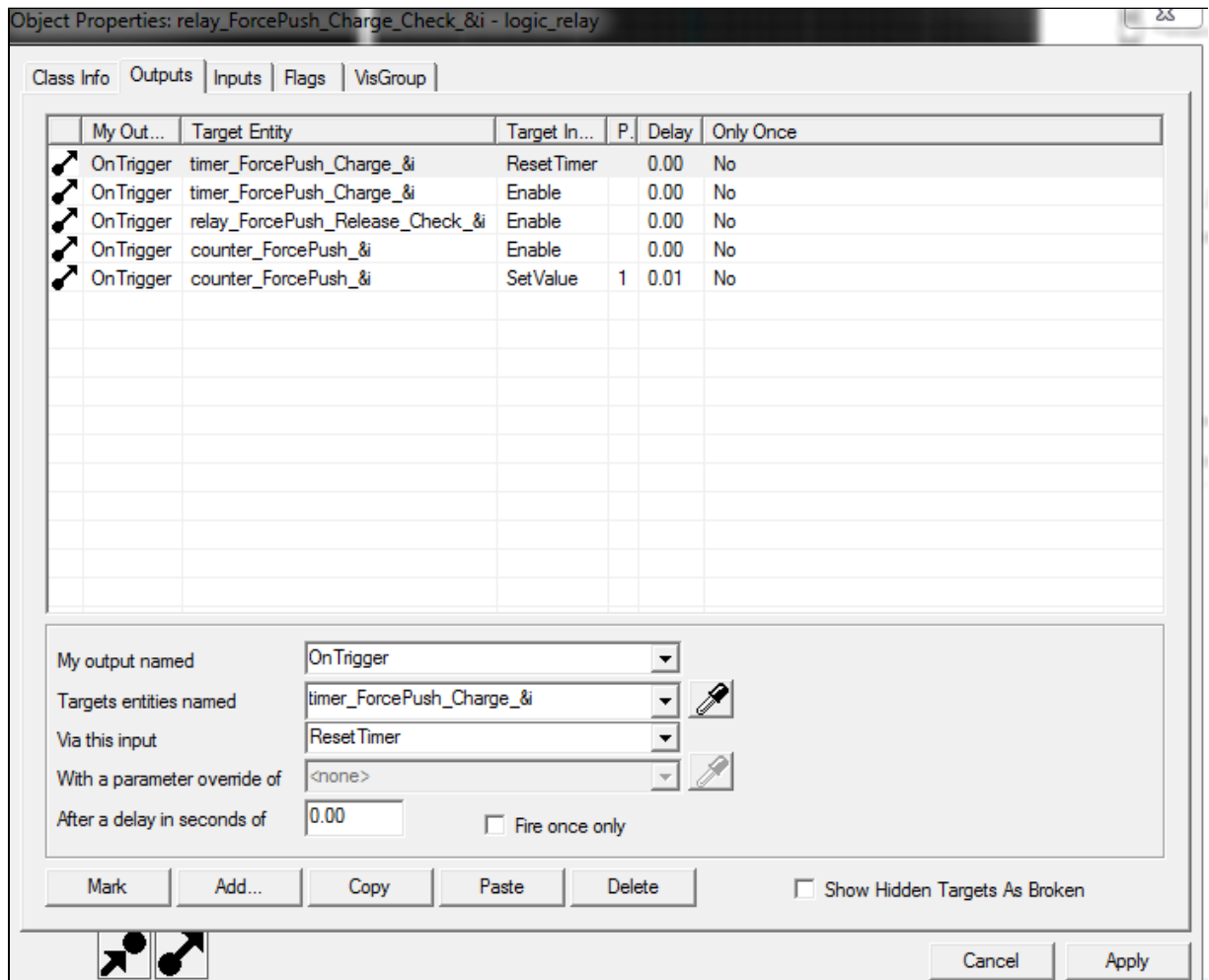
- Here is what the final logic sequence will look like, give or take (ignore the little **env_shake** dancing people):

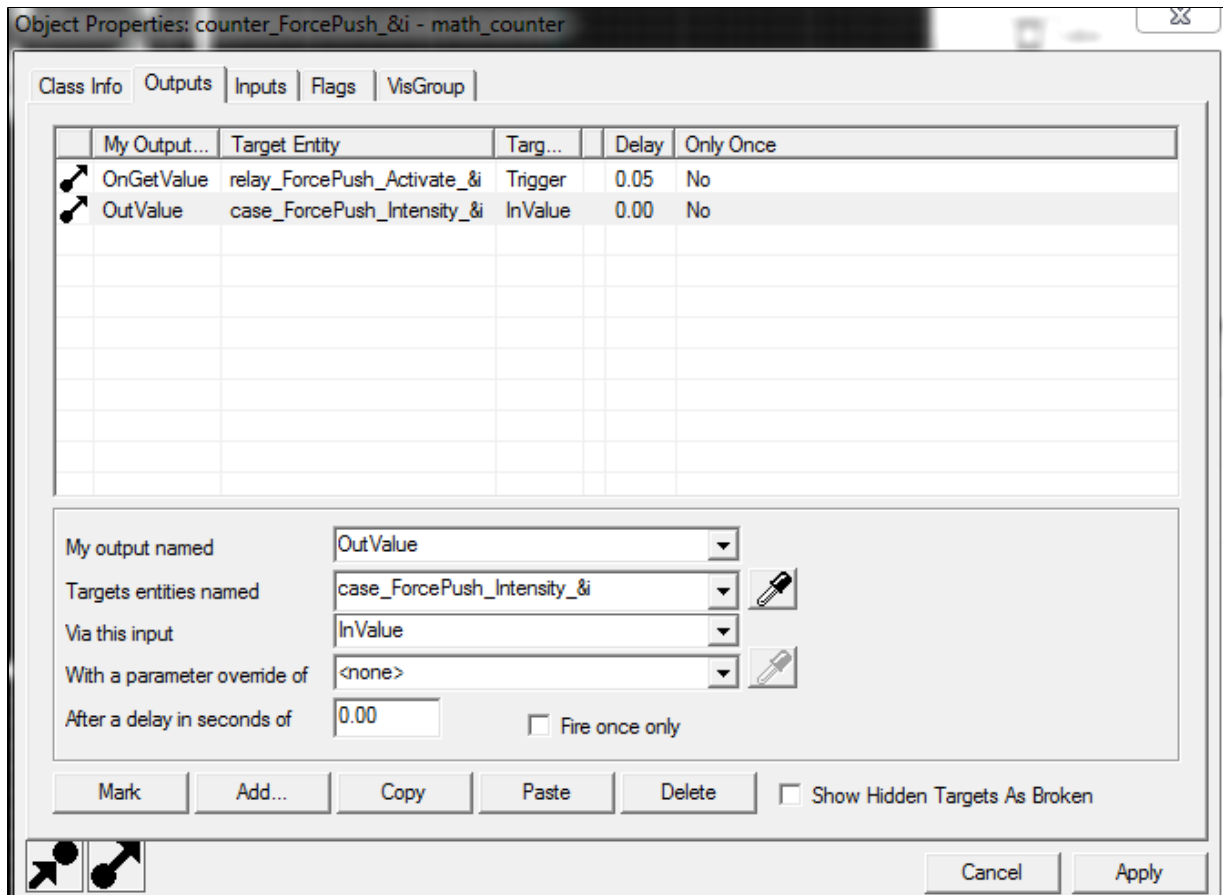


- *Alright, so let's snort some confidence and light this candle!*

1. First, a quick run-down of the logic we'll be making here:
 - a. We use the Pressed and Unpressed events from a **game_ui** to trigger two separate **logic_relays**, one that starts the charging and one that releases the Force (a.k.a. **JOHN CENA**)
 - b. On Charge start (whatever input you chose goes down), a **logic_timer** is enabled that, every **x seconds** (x is up to you, I used 0.65), adds 1 to a **math_counter**.

- i. Each time this counter gets updated, it calls an output called "**OutValue**", which uses the "**InValue**" input of a **logic_case** to pass its value along.
 - ii. The **logic_case** then uses that number to activate a **case** output to enable one of three **logic_relays** and disable the remaining two. Each of these relays contains the logic to activate one of the three Force Push Triggers
 - c. On Charge Release, the charging timer is disabled and a main Activator relay gets triggered, calling the trigger input on all three force push relays. Only the one that is enabled has its logic called. These usually activate their specific push trigger for about 0.1 seconds
 - i. You likely want to impose some type of cooldown here, so the Activator relay will disable the Charge relay on trigger, and re-enable it after a set delay time (1.25 seconds feels pretty good).
 - d. For usability, if the player attempts to restart Charging before the cooldown is over, I added a special logic relay at the end. This special Restart relay is told by the Activator relay to re-trigger the Charge relay after the cooldown ends. It starts disabled by default, and the moment it fires, it re-disables itself.
 - i. We enable and disable the Restart relay on mouse up and down, so if the player has the mouse held down (Pressed has been called but not Unpressed), it will automatically re-trigger the relay.
 - ii. This small addition made using the charged force powers feel 200% better. Turns out people really hate to wait.
2. First off, let's make the **trigger_push** volumes for our actual Force Powers. Make these out of **BSP** in whatever shape you want. I chose the funnel shape as that feels the most intuitive and functions the best with the weird camera-panning that Gordon does.
- a. You will need a separate **BSP** for each different charge level of your Force Powers. Shape them with the **Vertex Tool**, then convert them to **trigger_push**.
 - b. Now, any good scripter worth their salt understands how redundancy is great for efficiency in logic systems. So you may be asking, "Why not make just one volume and adjust its size and Push Speed dynamically at run-time?" Don't worry, you'll learn how this works eventually.
 - c. **QUIRK: Hammer doesn't allow the changing of many random key attributes of converted BSP entities (and others for that matter). You can only set them explicitly in the editor and make separate versions entity for every required variation.**
 - d. So each **trigger_push** will be a different level of your charged Force Push. Name them appropriately, as referencing is key for this next section.
3. Next, we need the relay that starts it all! Create two logic relays that look like the ones below, one for Charging and one for Releasing, respectively. The Releasing Relay starts disabled.
- a.






6. The **logic_case** is next, controlling which **logic_relays** are enabled at each charge level.
 - a. It will have as many cases as you have charge levels, each with an appropriate charge number as their value. The associated number determines which Case Output is called, so map it appropriately.
 - b. You can also use this to apply any charge effects or sounds, like **env_shake**, **ambient_generic**, or **env_screenoverlay**. These are basically free events for when the player reaches a certain charge level, so use them wisely.

Object Properties: case_ForcePush_Intensity_&i - logic_case


Class Info | Outputs | Inputs | Flags | VisGroup

My Outp...	Target Entity	Targ...	Delay	Only Once
OnCase01	relay_ForcePush_SetPower_Low_&i	Enable	0.00	No
OnCase01	relay_ForcePush_SetPower_Med_&i	Disable	0.00	No
OnCase01	relay_ForcePush_SetPower_High_&i	Disable	0.00	No
OnCase02	relay_ForcePush_SetPower_Med_&i	Enable	0.00	No
OnCase02	relay_ForcePush_SetPower_Low_&i	Disable	0.00	No
OnCase02	relay_ForcePush_SetPower_High_&i	Disable	0.00	No
OnCase03	relay_ForcePush_SetPower_High_&i	Enable	0.00	No
OnCase03	relay_ForcePush_SetPower_Low_&i	Disable	0.00	No
OnCase03	relay_ForcePush_SetPower_Med_&i	Disable	0.00	No

My output named



Targets entities named 

Via this input

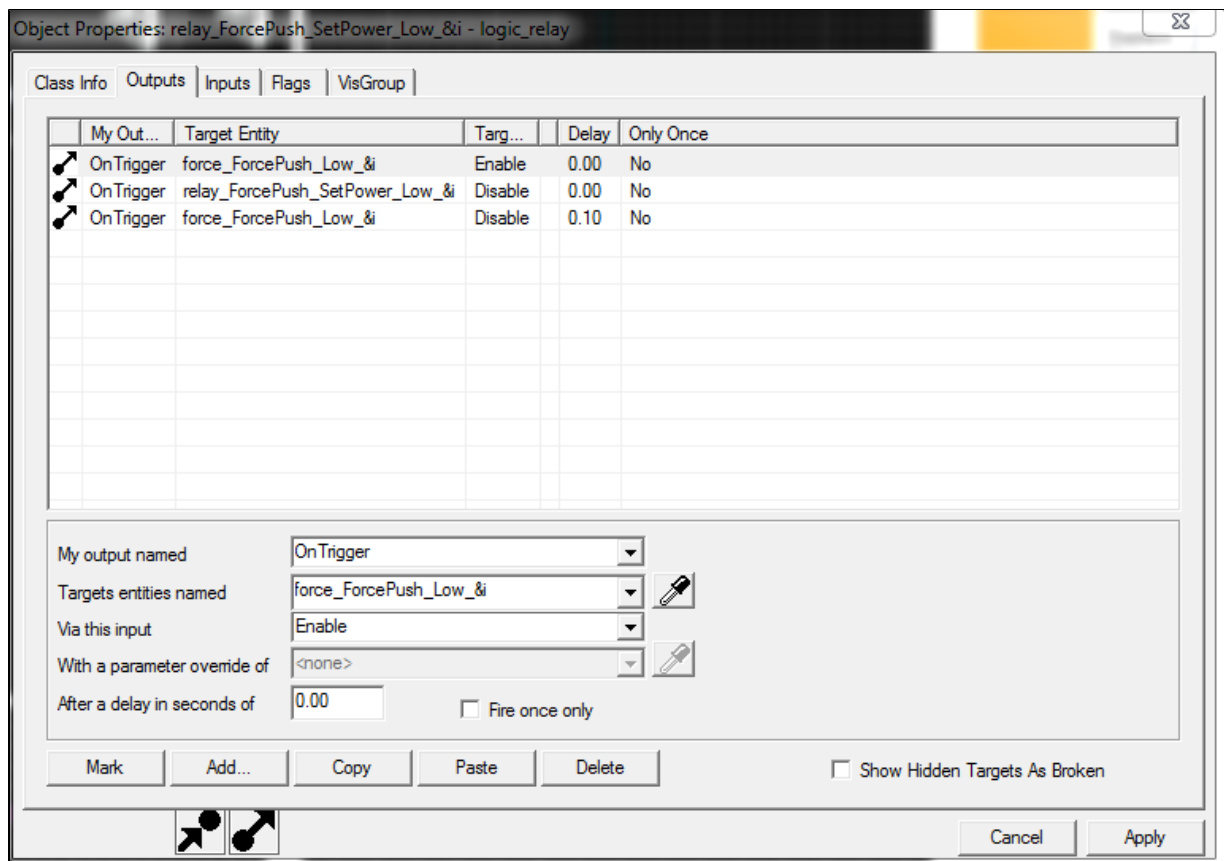
With a parameter override of 

After a delay in seconds of ☐ Fire once only

Mark Add... Copy Paste Delete ☐ Show Hidden Targets As Broken

  Cancel Apply

7. The case enables/disables the relays that control which push trigger activates. The following is only one of the 3 relays, but they will all always use the same logic pattern to activate its appropriate trigger_push.
 - a. Replace the force_* with whatever you called each **trigger_push** for each individual charge level.
 - b.



8. Once the player releases the Force, we immediately trigger the Activator Relay!
 - a. This activates all of the relays responsible for enabling the **trigger_push** entities.
 - i. You can see that, in the first output line of the example picture, you can use an asterisk at the end of a shared entity name. For Example, you could call all the charge level relays **chargeRelay_#** where the **#** is the **charge level** and use **c** **hargeRelay_*** to call the "Trigger" input for each one with one Output line.

Hopefully you had some fun and learned a lot more about Hammer than you actually needed to do the specific task you came to this guide for. Its a great engine that is worth the time it takes to build in it.

Either way, here's Kermit being way more excited about your success than you are!

Also, listen to [duel of the Fates](#) while watching this. Seriously.



Click on .Gif for reference link

 [How to Jedi with the Force](#)

 [Making Custom Materials \(Of Various Complexities\) for Hammer](#)