Version 0.1a
22 Dec 2019

Quake 1 Dev Kit for Dynamic Map Logic

Authored by Qalten
(Qalten#6722)

Quick Usage Tutorial

This document will briefly demonstrate how to setup CPQ entities in a Quake 1 map. For this guide, we will be using TrenchBroom (v2019.6) along with the q1_cpq.fgd entities definition file included in the dev kit. All references are as of the version number listed on the front page of this document and may be superceded or deprecated in future versions.

This document assumes a basic working knowledge of mapping with Quake 1. If you're looking for a walkthrough on mapping itself, dumptruck_ds has an awesome YouTube playlist of videos that can be found here.

Adding Registers

The most crucial entity in CPQ is the **info_register**, a very simple point entity that stores a float value (using the predefined **count** field). Add this to your map, and make sure it has a useful **targetname**. For our example, we'll call ours *quickRegister*. It can be placed anywhere in the map, and does not have any sort of trigger from being touched.

Setting Registers

Next, we need a way to start putting some values into *quickRegister*. Let's create a **trigger_setregister**, a brush entity that, when triggered, will set the current value of *quickRegister*. As all **triggers** in CPQ are based off of the normal Quake triggers (specifically, **trigger_multiple**), they will share the same fields and mechanics, along with some new ones. For our new **trigger_setregister** brush, find the **target_reg_1** field and enter *quickRegister*. This establishes that we want to go find the *quickRegister* register, and however we derive our value, want to set it there. Now all we need to do is define how we're setting the value. There are two ways to do this, one is to simply set the **count** field to a number (for example, 5). When triggered, this brush would simply set *quickRegister* to 5.

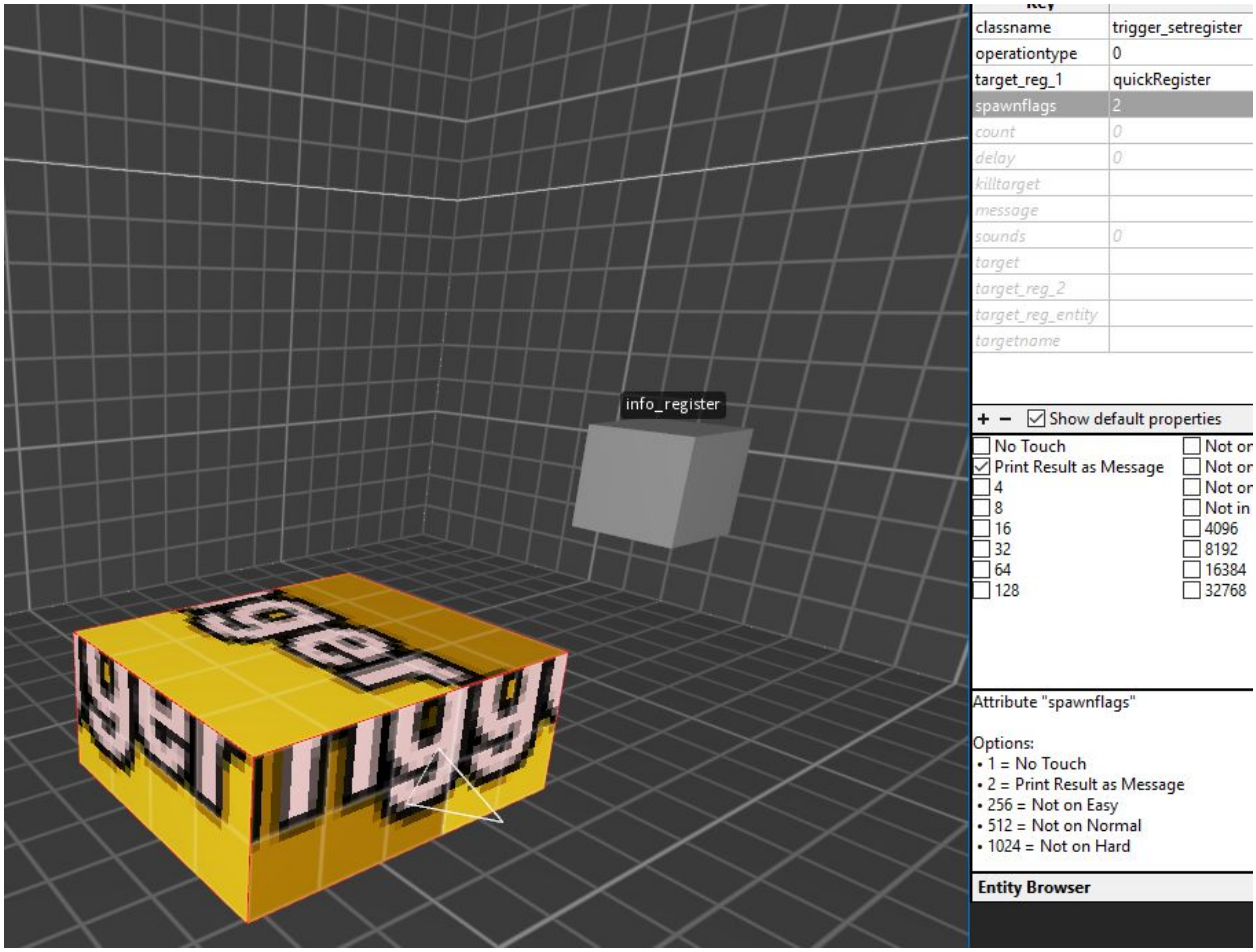
We want things to be a bit more dynamic, however, so let's instead ignore **count** and instead use the **operationtype** field. All triggers in CPQ utilize this new field to determine how they will behave when triggered[1]. Let's use the default **operationtype** 0, which will set the given register to the value of the activator's health. In our test scenario, this will be the player.

For debugging purposes, let's go ahead and enable the second flag in the **spawnflags**, "Print Result as Message" on our **trigger_setregister**. This will pop up the result of setting the register as a message in the game.

---

[1] We can also modify the **operationtype** field itself with *other* CPQ triggers to dynamically change how CPQ triggers behave, but more on that will be found in the included entity reference.

At this point, your map should have entities like this:

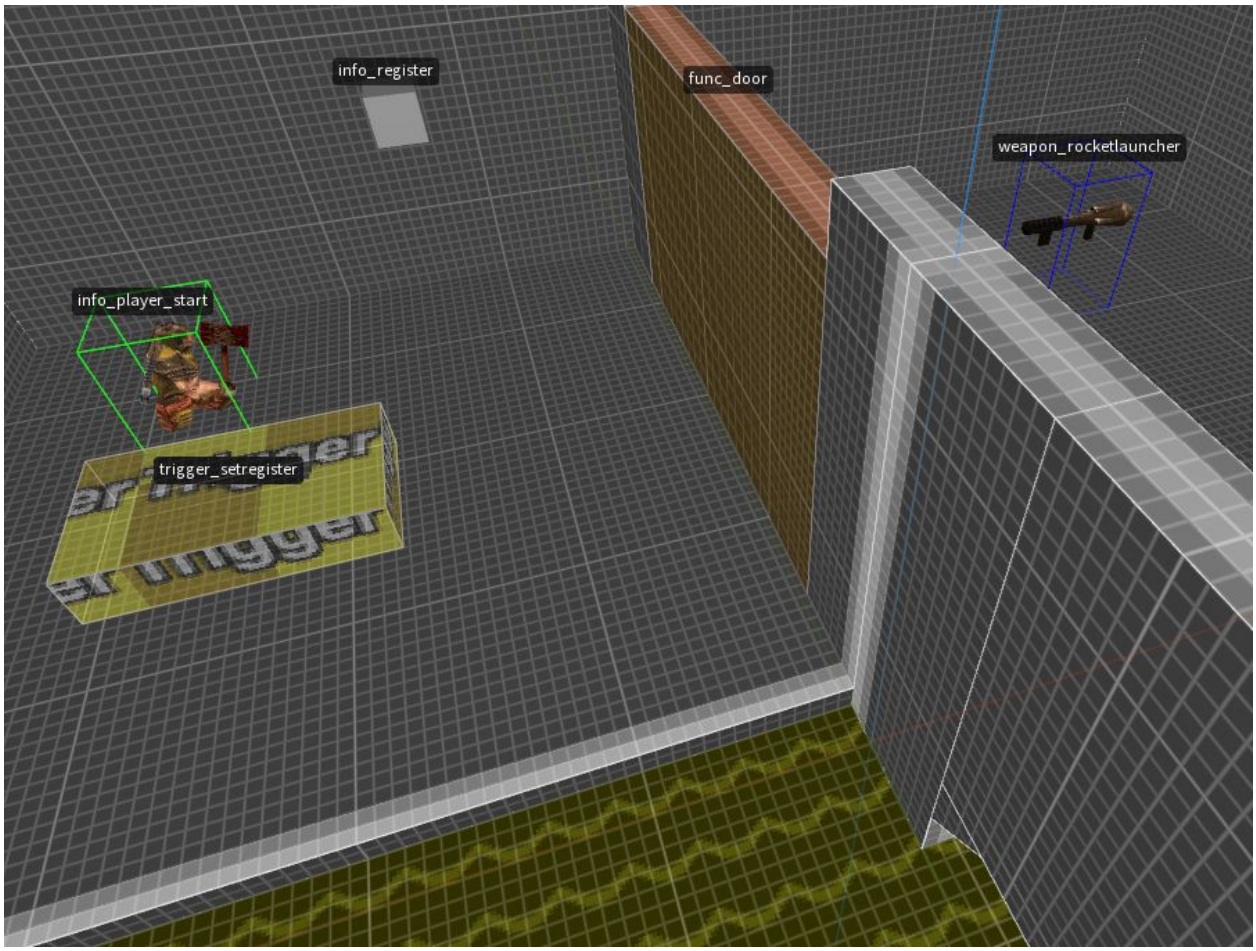


Feel free to test your map now. If everything is working, you should see "100" printed in the center of the screen when you walk over the trigger[2].

[2] There is a known issue right now where a CPQ register-related trigger may take a moment to respond the first time a map is loaded. This also can result in an error being displayed if a message is involved. I'm hoping to find a solution for this soon!

Evaluating Registers

Now, let's do something with that value. To do so, however, we'll need to setup our map to have a bit more functionality. Create a brush that will damage the player (slime is best as it doesn't hurt them too fast) and add a **func_door** that we'll want to get into. Make sure to give the door a **targetname**, let's do *quickDoor*. Here is my arrangement:



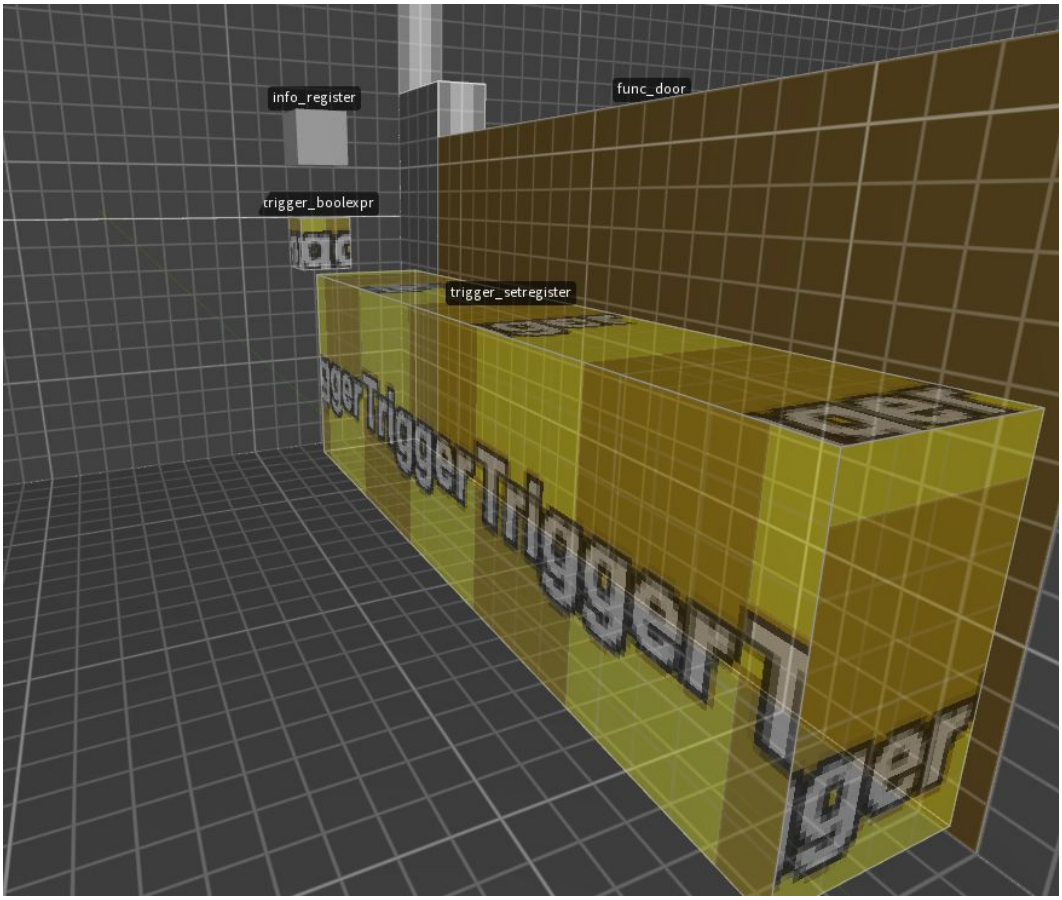
Now, let's add a new CPQ trigger brush, **trigger_boolexpr**, which is short for "boolean expression" ie: whether something is **true** or **false**. We're going to use this trigger to evaluate the value of our *quickRegister* register, and depending on what it is, open the door. Now, for this setup we don't need the **trigger_boolexpr** to actually be physically triggered, so make it small (16^3 perhaps), enable the first **spawnflag** ("No Touch") and put it somewhere tidy. Let's go through configuring our new trigger now. The first field we need is **operationtype**, the options of which are different boolean evaluations. Let's

choose 4 (“Less than”), set **count** to 80, **target_reg_1** to *quickRegister* and **targetname** to *checkDoor*. What we’re doing here is configuring the trigger to look at the value in *quickRegister*, and see whether or not it’s less than 80. Now, we need to set the behavior of the evaluation, what will happen when it’s true or false. We do this by setting **target_expr_true** and/or **target_expr_false**. These values will be the entities we wish to trigger based on the boolean result. For gameplay purposes, we also have access to **message_expr_true** and **message_expr_false**, each of which will display a message to the player based on the result. All of these fields are optional, but let’s use a couple of them. Set **target_expr_true** to *quickDoor*, and **message_expr_false** to “Come back with less health...”.

Your **trigger_boolexpr** should now look something like this:

Key	Value
classname	trigger_boolexpr
operationtype	4
targetname	checkDoor
spawnflags	1
target_expr_true	quickDoor
target_reg_1	quickRegister
count	80
message_expr_fals	Come back with less health...
delay	*0*
killtarget	

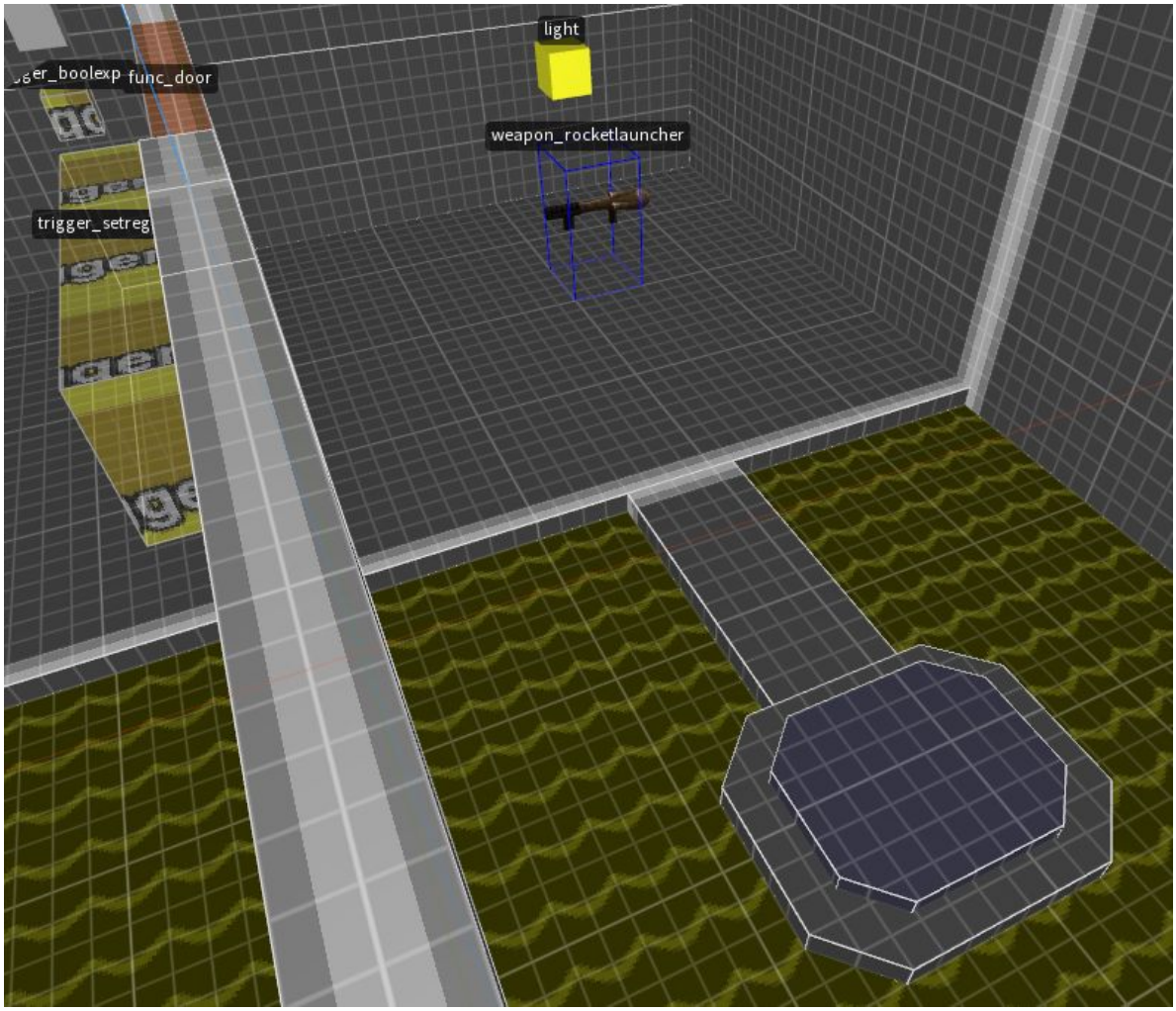
We’ve almost finished! All that’s left to do is connect our two triggers, and modify the brush a bit. On the **trigger_setregister**, set the **target** to *checkDoor*. CPQ can take advantage of the standard method of activating triggers by using **target** in this way. Finally, adjust the brush of the **trigger_setregister**, so that it covers an area in front of the door where the player would attempt to stand in. Our tutorial map should now look something like this:



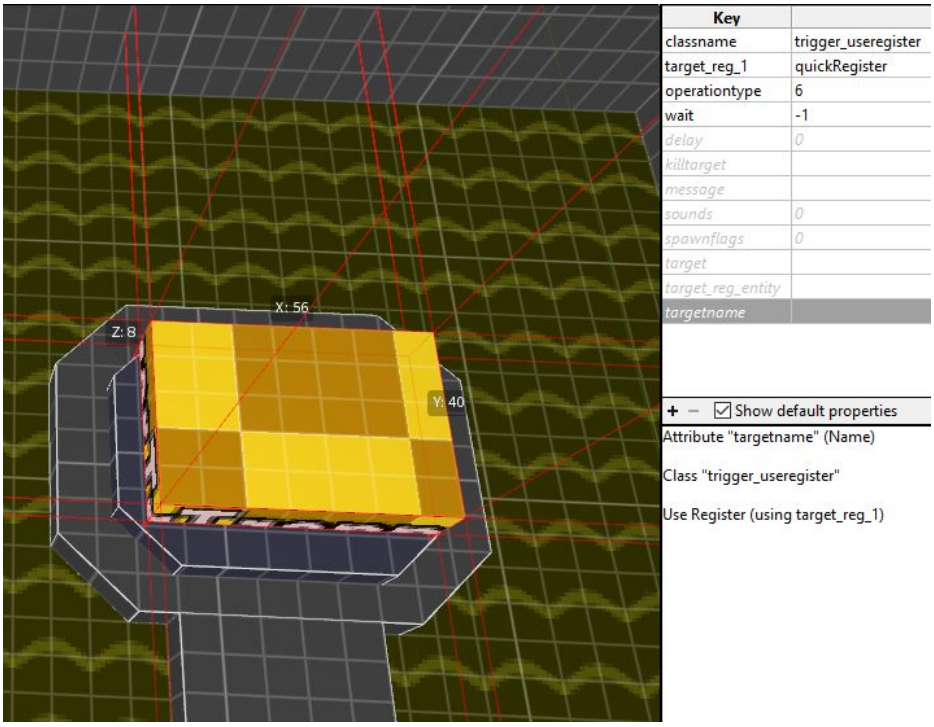
Give it a test and make sure it works! With everything functioning, the trigger in front of the door should check the player's current health, and if it's below 80 (achieved through a small dip in the slime), open the door for them.

Using Registers

Our last part of the tutorial will cover the other way of CPQ can leverage the value of a register, **trigger_useregister**. This trigger reads the value of a register it's connected to, and uses it to modify something else, or give the player something. Somewhere in your test map, add a small area of interest that is distinct from the rest. Here is my version:



What we want to do is make this an area where the player will be granted a one-time benefit. Create a new brush entity of type **trigger_useregister** and place it where you want your powerup to be activated. Now, let's setup the trigger. As before, let's use the same *quickRegister* for **target_reg_1**, and for **operationtype**, let's select 6 ("Add Player Armor"), and to make this a one-time benefit set **wait** to -1. The result of this will be a single trigger that, when the player walks onto it will give them an amount of armor equal to their health when they last passed through the door (as that's when **trigger_setregister** is triggered). Obviously in a full map, we'd want to use lots of different registers and different ways of triggering them, but this tutorial map gave us a nice and straightforward way of examining the important entities in CPQ. My version of the **trigger_useregister** looks like this:



The End…?

There's one more CPQ entity that we didn't cover here, and that's **trigger_operator**, which allows you do execute mathematical operations with register values. I may add that to the tutorial at a later point but to learn more about it, check out the included Entity Reference Document.

And that does it for the CPQ Quick Tutorial! I will include my version of the tutorial map (cpq_quickstart.map) in the dev kit package, so feel free to open it up and use it as a start for your own CPQ-enabled map.

Please reach out to me on Discord if you have any questions or feedback, and **especially** if you make a map using CPQ! It is my intent to keep a running database of all CPQ-enabled maps so players and mappers alike can learn from each other as more features are added to the dev kit.

Thanks for checking out CPQ!

(I used the very handy Prototype WAD for my initial texturing, download it here!)