Map Scripting 4 Newbies

Index

1. Overview

Why Script?

Scripting brings your map to life. It's the part of mapping that allows players to interact with the map environment, build things, destroy objects, move vehicles, cap objectives, and win the game. To use a computer analogy, your map is the nice shiny box sitting on your desktop. It's great to look at, but without the applications and programs you load into it (the script), it's incapable of really doing anything.

Unfortunately, it seems like scripting is the redheaded stepchild of mapping. It is not uncommon to see mappers spend an inordinate amount of time creating fantastic map architecture only to realize, at the end, that they have absolutely no clue about scripting or where to start. Common sense would seem to promote learning the basics about scripting (e.g., what you want players to do in your map) well ahead of finalizing your architecture and texturing.

One can probably attribute this to a health fear of scripting. I know when I opened my first script, one of the .SCRIPT files from Splash Damage, I had no idea what I was looking at. As a result, a lot of people end up copying and pasting sections of code, without really understanding what it does for/to them. In this set of tutorials, I will try to demystify scripting a bit and provide a basic understanding of how to write simple scripts that make your map playable.

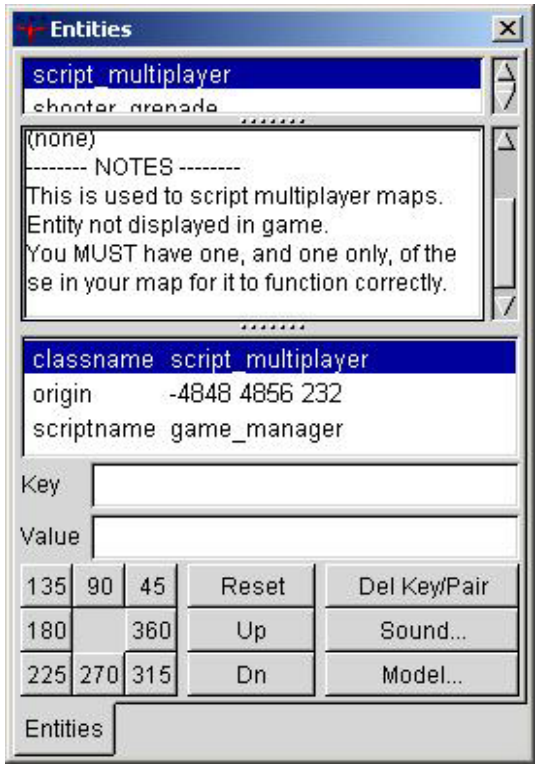
What This Tutorial Is NOT!

- This tutorial will not cover how to make entities (func_destructible, func_constructible, Team_CTF_red/blueflag, etc) or how to incorporate them into your .MAP file
- This tutorial will not cover anything having to do with vehicles or moving objects, since I have no clue how they work
- This tutorial does not address the coding for VOs nor adding and deleting them from the queue.

- This tutorial is by no stretch of the imagination an expert treatise on scripting ... just my notes on what I've learned in my first month of mapping

2. On To The Basics

What Do I Need To Get My Map Script Working?



- Script_Multiplayer entity. Think of this almost like the autoexec.cfg in your game script. The script_multiplayer automatically launches the game_manager script every time you launch your map
- Text Editor (Notepad works fine)
- Examples (can be found in the /maps folder of any .PK3 file and will be named [mapname].SCRIPT
- Game_manager routine, victory script (typically found inside the game_manager routine), and your objective scripts, which you can find in lot of tutorials

How Do Scripts Get Initiated?

Scripts can be initiated in one of three ways:

1. Due to the occurrence of a game event (e.g., spawn, time expiration)
2. Due to a change in the status of an entity (e.g., construction completed, death of a func_destructible). This is typically caused by a player action via a trigger, but not necessarily so. An example of this would be an unfinished construction, which automatically reverts to the unbuilt state after 30 seconds of inactivity
3. Called by another script. You can chain scripts together, using the 'trigger' function, so that one script calls another and so forth. An example of this would be the *objective_counter* and *gamecheck* scripts, which are executed every time an objective is completed to see if the game is over.

We're going to see the term 'trigger' quite a bit, so I should stop and briefly explain what it means. There is a very good tutorial on www.nibsworld.com titled, "Move A Platform. In it, there is a very detailed explanation, by Boltyboy, of the scripting required to make the platform move. I've paraphrased part of that explanation here.

The keyword 'trigger' will be seen in two places, 1) in the name of a subroutine and 2) as an executable line of script. In the first case, the word 'trigger' is almost always used at the start of a new section of code or subroutine. It's just telling the game engine, "this is a point where a new section of code will be triggered". In the second case, it's used to call another subroutine. It has the syntax trigger [name of routine][name of subroutine], which is

basically saying, “go and look for the following trigger, which is called [name of subroutine] and can be found in the routine called [name of routine]”.

What Are The Basic Script Elements?

You can think of each script having 3 distinct elements

1. Script Name
2. Script Contents
    - Spawn
    - Additional entity states
    - Additional instructions added by mapper
3. Comments

If you’re familiar with writing scripts for your player config, then understanding how map scripts are organized should be a snap. In your player autoexec, you might have the following line:

Set gamename2 “name ^4Hot Cross Bunny” //changes to pub name

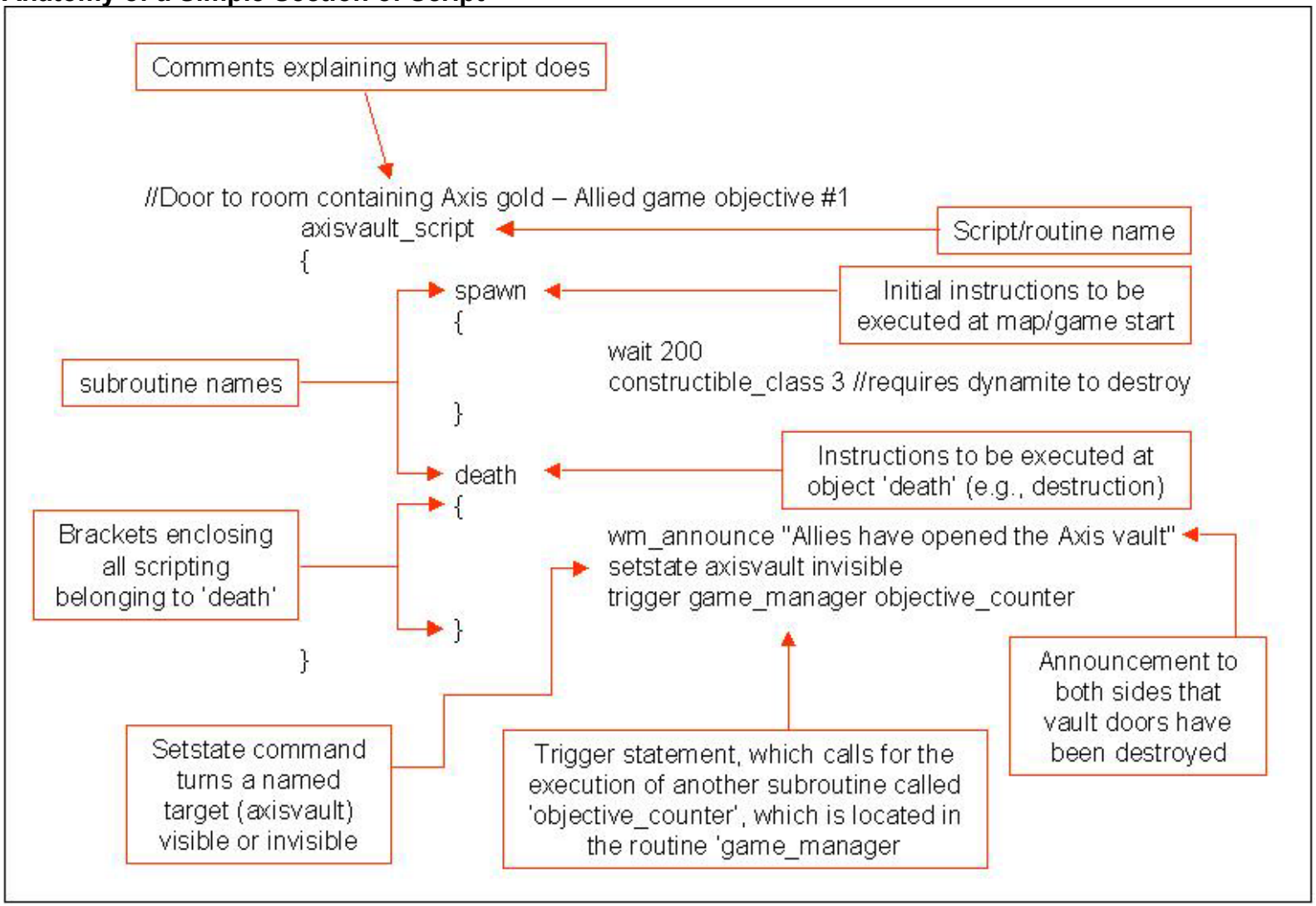
That line says, name the script ‘gamename2’ and when it’s executed, change my name to Hot Cross Bunny, in blue, then has a comment (everything after //) to remind me, or tell others what that line does. Everything except for the script name and the comments are encased in quotations (“), to show the game engine where the script starts and stops.

A key difference with map scripting is the concept of script hierarchy. I like to think of scripts in terms of routines and subroutines. In fact, after this point, I’ll use the terms ‘routine’ and ‘subroutine’ interchangeably with the term ‘script’. Just like a book is organized into chapters and paragraphs, a script has higher-level routines (chapters), which reflect a single overarching object (e.g., a destructible objective) and lower-level sub-routines (paragraphs), which reflect distinct states or stages of the routine’s subject. For example, one routine might be the script for a destroyable wall, called ‘*old_city_wall’* with subroutines called ‘*spawn*’ and ‘*death*’. That’s simple enough. Another, more complex, routine might be the script for a constructible fence, called ‘*fuel_dump_gate*’ with subroutines (states) of *spawn*, *buildstart*, *built*, *decayed*, and *death*.

Each standalone portion of script starts with a ‘{‘ and ends with a ‘}’, much like quotes are used to start and end a script in your player config.

Here is the script for a simple func_destructible, which calls a victory script upon destruction, to see if the game should end:

Anatomy of a Simple Section of Script



Again, we use comments to remind people what the script does. In this case, reminding us that this script controls the door protecting the axis gold, which the allied have to blow up, and that the door requires destruction by dynamite. Simple enough.

The routine has a name, '*axisvault_script*', that makes it easy to figure out what it's all about. You can really call your routine anything you want, but it's much easier when the name is short, simple, and related to its function. Each of the

subroutines has a name too. The specific required names of your subroutines is usually determined by the supported entity. In the case of a func_destructible, there are two: *spawn* and *death*.

In all cases, each standalone portion of the script, after the script name, starts with a { and ends with } to signify what is associated under the scripts named '*axisvault_script*', '*spawn*', and '*death*'. Also note, that each hierarchical level is indented to more easily show script relationships. You don't have to do this, but it sure makes editing, proofreading, and debugging scripts much easier.

Every script needs to have a subroutine called spawn. It doesn't have to contain any coding or instructions, but needs to be there. Anything in the '*spawn*' subroutine automatically gets executed at the start of the game. If you do have instructions in the spawn subroutine, then added a short 'wait xxxx' or you may end up with annoying little errors here and there.

How Are Scripts Organized?

I don't think this are any hard and fast rules governing how the overall [mapname].SCRIPT file needs to be organized, but it seems that a lot of map scripts are organized with the game_manager at the start and then major sections, organized by entity class and delineated by comments follow. This makes sense because the game manager is an overall script that outlines the initial state of every objective, sets game length, victory conditions, and spawn times, where as all subsequent sections are specific to a particular piece of the map or entity.

OK, enough of the basics, let's get into a real script. The first thing you will need is a *game_manager* script. Once you have your basic *game_manager* script in place, you can append any entity scripts you write/download/copy/steal etc, as new routines. We'll start with the script from Oasis and take a look at the game_manager routine there. Realistically, you can probably copy any game_manager script and edit it for your map, once you understand all the components you'll find inside. Here we go.

3. Holy Crap! I Don't Understand Anything I See Here!!!

Let's open up a .SCRIPT file and take a look at the first part you'll see, the game_manager routine. The rest of the routines are going to pretty specific to particular entities (func_constructibles, func_destructibles, forward spawn flags, command posts, movers, etc). We'll take a look at some of those later on. Use Pakscape or Winzip, go into etmain/pak0.PK3 then look for oasis.SCRIPT in the /maps folder. Oasis is a pretty straightforward map with 2 destructible objectives and no moving vehicles. Open it with Notepad. I've copied the *game_manager* portion here and deleted the portions related to VOs, pumps, and the forward spawn. Luckily, you'll notice that Splash Damage already used a lot of comments to explain what they did. My added comments are in red boxes so that they are distinctive from the comments already made by Splash Damage.

```
game_manager
{
        spawn
```

Every map script needs to have a game_manager section and every routine needs to have a spawn subroutine, even if it contains no instructions.

```
        {
                accum 1 set 0        // State of objective number one
                accum 5 set 0        // Current number of Pak 75mm guns destroyed
                accum 6 set 0        // Current number of water pumps built
                accum 7 set 0        // Value used in checking whether or not to announce "Axis have damaged both
water pumps!"

                globalaccum 5 set 0
                globalaccum 6 set 0
```

Setting all accum and global accum values to 0 at the beginning of the game is a good way of ensuring that all objectives are reset to their intended starting positions. It's just like starting your custom player config with 'unbindall'

```
                // Game rules
                wm_axis_respawntime      30
                wm_allied_respawntime    20
                wm_number_of_objectives          8
                wm_set_round_timelimit   30
```

This section sets the basic game rules for match length and respawn times. Wm_number_of_objectives should match the number of objectives in your .OBJDATA file

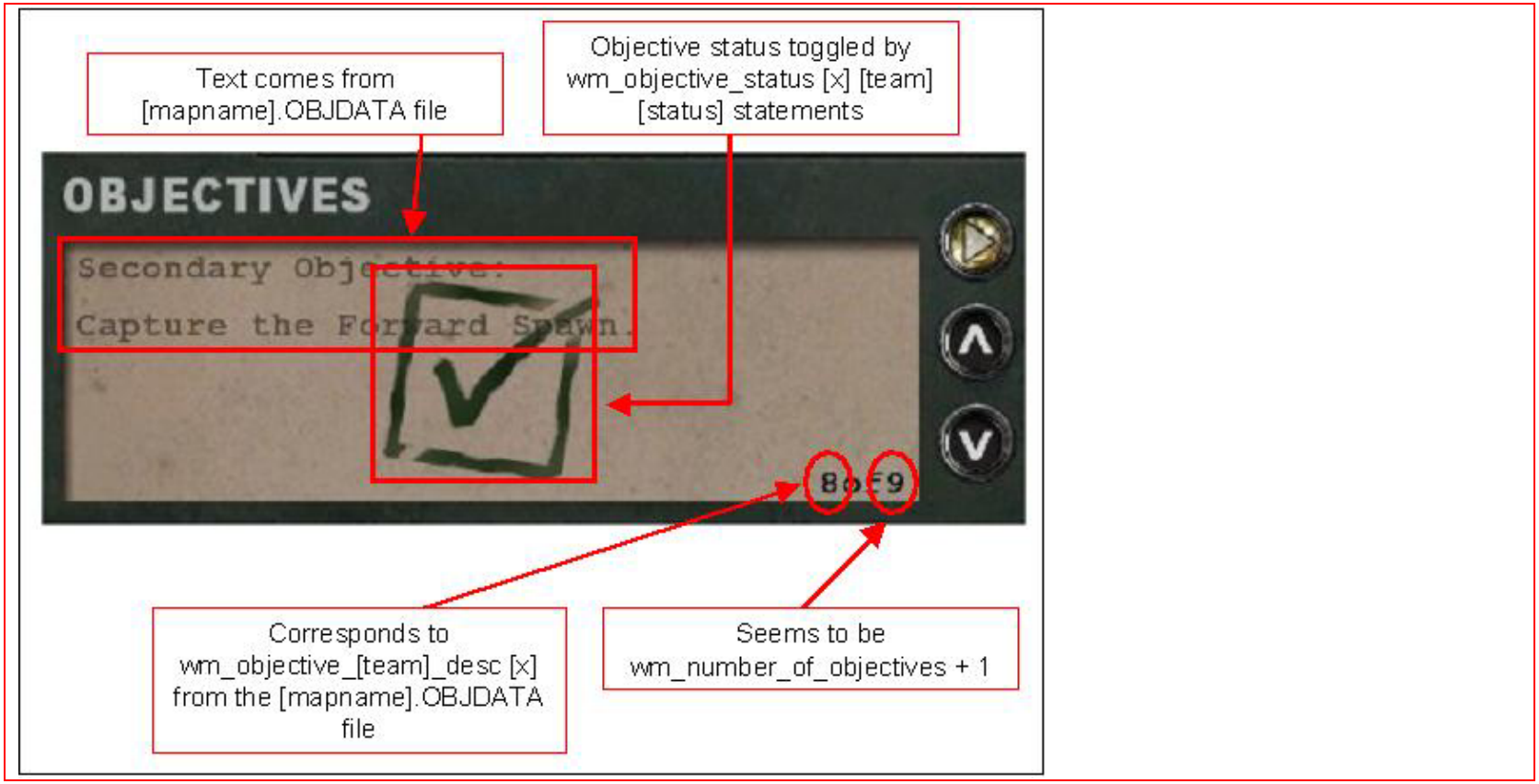
```
                // Objectives
                // 1: Primary1 : Destroy the North gun
                // 2: Primary2 : Destroy the South gun
                // 3: Primary3 : Breach Old City wall
                // 4: Secondary1 : Capture forward spawn point
                // 5: Secondary2 : Drain/flood cave system by repairing/damaging the Oasis pump
                // 6: Secondary3 : Drain/flood cave system by repairing/damaging the Old City pump
                // 7: Allied command post
```

```
// 8: Axis command post
```

Many people add a comment section here to remind them what are the map's objectives

```
// Current main objectives for each team (0=Axis, 1=Allies)
wm_set_main_objective		3	0
wm_set_main_objective		3	1

// Objective overview status indicators
//wm_objective_status <objective> <team (0=Axis, 1=Allies)> <status (0=neutral 1=complete
2=failed)>
wm_objective_status 1 1 0
wm_objective_status 1 0 0
wm_objective_status 2 1 0
wm_objective_status 2 0 0
wm_objective_status 3 1 0
wm_objective_status 3 0 0
wm_objective_status 4 1 0
wm_objective_status 4 0 0
wm_objective_status 5 1 0
wm_objective_status 5 0 0
wm_objective_status 6 1 0
wm_objective_status 6 0 0
wm_objective_status 7 1 0
wm_objective_status 7 0 0
wm_objective_status 8 1 0
wm_objective_status 8 0 0
```

You have to manually tell the game which team controls each objective. This is done here, to show all objectives as uncompleted, as well as later in the script as various scripts get triggered and objectives are completed. As the comment above notes, the format is wm_objective_status [obj number (1 thru n)] [team (0 or 1)] [status (0 thru 2]. In-game, as the game progresses, you will see checkmarks in a box or x's in a box for each objective listed in the Limbo menu. The wm_objective_status commands make that happen. Here's a quick explanation that links what you see in the Limbo menu with commands in your .SCRIPT and .OBJDATA files:




```
// Stopwatch mode defending team (0=Axis, 1=Allies)
wm_set_defending_team   0

// If the round timer expires, the Axis have won, so set the current winning team
// Set the round winner:  0 == AXIS, 1 == ALLIED
wm_setwinner 0

// Set autospawn markers <team (0 = axis, 1 = allies)> <message key of marker>
// Spawns on siwa:
// Axis Garrison
// Allied Camp
// Old City
wait 150

setautospawn "Old City"                    1
```

```
        setautospawn "Old City"                 0
```

The name of the spawnpoint used comes from the description key of the Team_Wolf_Objective entity. Both teams have their spawnpoints autoset for the Old City which makes people spawn as close to the old city flag as possible. Effectively, this means that as soon as a team captures the forward flag, you'll automatically spawn there.

```
        wait 2000

}

trigger 75mm_gun_counter
{
        accum 5 inc 1                           // Increase game counter
        accum 5 abort_if_not_equal 2            // All guns destroyed ?

        wm_announce     "Allies have destroyed both Anti-Tank Guns!"

        wait 8000

        accum 1 set 1                           // Both pak guns destroyed

        // Call function to check if the round has been won
        trigger game_manager checkgame
}
```

This is a simple objective counter script that counts how many objectives (Pak 75mm guns) have been destroyed. Whenever the North or South gun is destroyed, it calls the gun_counter script

```
trigger checkgame
{
        accum 1 abort_if_not_equal 1

        // Set the round winner:  0 == AXIS, 1 == ALLIED
        wm_setwinner 1

        // End the round
        wm_endround
```

```
}
```

Each time a gun is destroyed, we check to see if the game has been won by checking if accum 1 is equal to 1. Accum 1 is set to 1 in the gun_counter routine only when both guns have been destroyed. If so, the Allied team is declared the winner. If not, the script stops and the game continues.

```
}
```

4. OK, I've Read Enough! Show Me How To Write A Script To Win The Game

Here are a couple simple scripts, three of which, we will combine to determine if the game is over after an objective has been completed.

Objective Scripts

For the sake argument, let's limit our game objectives to one of three types. The attacking team either has to build something, destroy something, or return an objective. I'm going to make a simplifying assumption that you have already added objectives to your map and you have the relevant scripting handy. You can find tutorials that tell you how to set up the map entities and the relevant scripting at (these are the ones I used anyhow).

- Splash Damage Forums
- WolfensteinX/Surface
- Level Designers Resource (found in [gamedir]/Radiant-1.3/docs)
- And many other sites

So, before we go on, first read thru the following 5 scripts and at least be comfortable with reading thru them. Scripts 1a – 1c are entity scripts that I won't cover here. Scripts 2 and 3 are similar to the *75mm_gun_counter* and *checkgame* scripts in the example *game_manager* from Oasis that we looked at previously

1a. Func_constructible

```
//Axis safehouse.  Axis team has to build a gate in front of a safehouse

safehouse_script
{
        spawn
        {
                wait 200
                constructible_class 2
```

```
                trigger self startup
        }

        buildstart final
        {
        }

        built final
        {
                setstate safehouse default
                setstate safehouse_materials invisible
                wm_announce "The Axis safehouse door has been constructed"
        }

        decayed final
        {
                trigger self startup
        }

        death
        {
                trigger self startup
                wm_announce "The Axis safehouse has been breached"
                setstate safehouse_materials default
        }

        trigger startup
        {
                setstate safehouse invisible
                setstate safehouse_materials default
        }
}
```

1b. Func_destructible

```
//Door to room containing Axis gold – Allied have to dynamite a door protecting a box of Axis gold
        axisvault_script
        {
                spawn
                {
```

```
                wait 200
                constructible_class 3
        }

        death
        {
                wm_announce "Allies breached the Axis gold vault"
                setstate axisvault invisible
        }
}
```

1c. Team_CTF_redflag

```
//Radar.  Axis team must capture and secure an Allied  radar module containing critical codes
        allied_radar //enter this as the scriptname value for the team_CTF_red/blueflag entity
        {
                spawn
                {
                        wait 200
                        setstate allied_radar_captured invisible
                }

                trigger stolen
                {
                        wm_announce 0 "Return the Allied radar set to the getaway truck"
                        wm_announce 1 "The Axis have stolen the Allied radar set"
                        setstate allied_radar_cm_marker invisible
                }

                trigger returned
                {
                        wm_announce 0 "The Allies have retrieved the radar set"
                        wm_announce 1 "Radar set returned!  Protect the radar set"
                        setstate allied_radar_cm_marker default
                }

                trigger captured
                {
                        wm_announce "The Axis have secured the Allied radar components"
                        setstate allied_radar_red invisible
```

```
                setstate allied_radar_captured default
        }
}
```

2. Objective Counter

```
trigger objective_counter  //Counts allied objectives completed
        {
                accum 1 inc 1
                trigger game_manager checkgame
        }
```

3. Game Win Check Script

```
trigger checkgame
        {
                accum 1 abort_if_not_equal 1
                wm_setwinner 0
                wait 1500

                wm_endround
        }
```

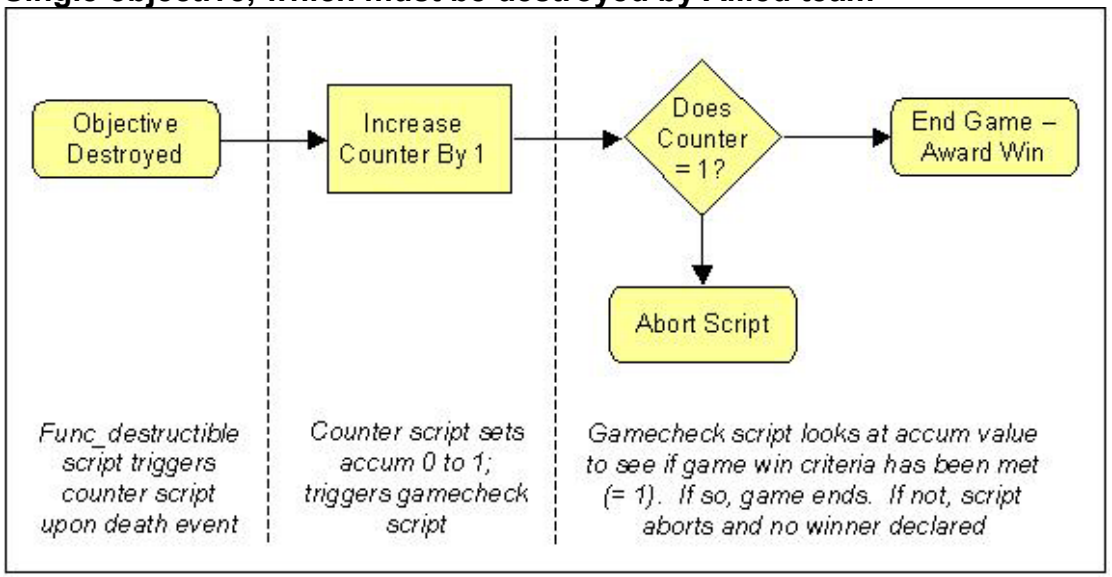
So How Do I Combine These To Determine If The Game Is Over?

Here are 4 examples in order of increasing complexity.

1. Single objective, which must be destroyed by Allied team
2. Two objectives, which must be destroyed by Allied team
3. Dual objective, both teams have two destructible objectives
4. Dual objective, both teams have two objectives they must capture (not yet written)

Notice that all four of them really use the same scripts, just with more objectives, different variables, and slightly different organizations. For each, I've included a flow chart, the script, and an explanation of how it's changed from previous scripts.

Single objective, which must be destroyed by Allied team



In this example, we used scripts 1b, 2, and 3. We have a standard func_destructible script, which has an additional line, "trigger game_manager *objective_counter*", in the *death* subroutine. This is the only addition to the standard func_destructible script you find in most tutorials and means that whenever the target is destroyed, it automatically executes the script, '*objective_counter*'. When call another script via a trigger statement, I have to specify in which routine it is contained (*game_manager*) and the name of the script itself (*objective_counter*). It's just like me tell you to go to Chapter 5, paragraph 4

'*Objective_counter*' increases the value of Accum 0 by 1, and then executes the script named '*gamecheck*' to see if the game is over.

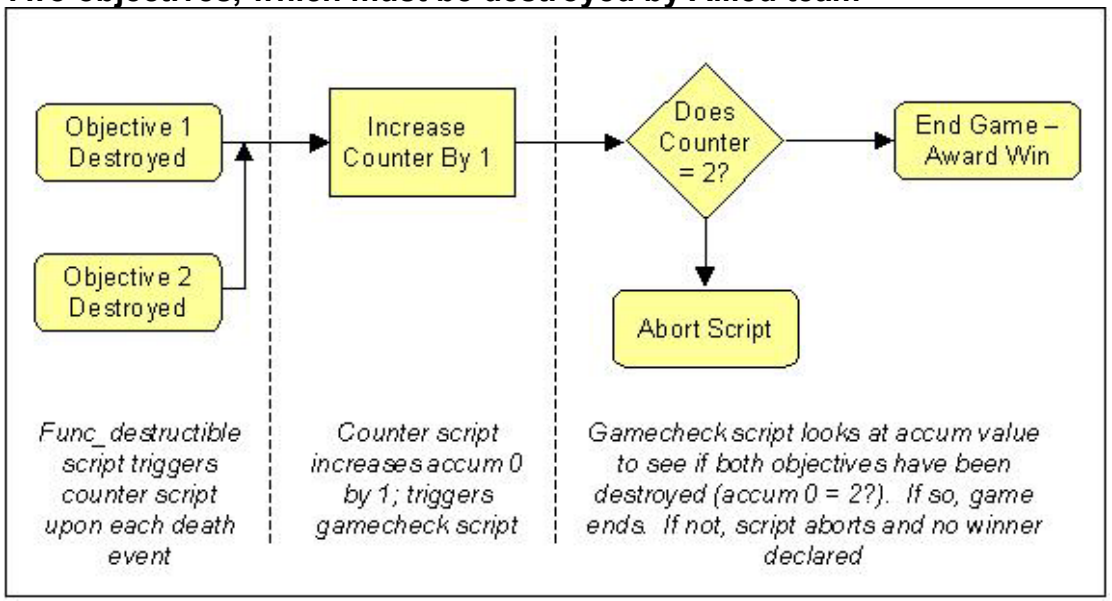
'Gamecheck' looks at the value of Accum 0 and if it is 1, it says, "yep, victory conditions are met, Allies win, and the game is over." If the value of Accum 0 is anything other than 1, the script just stops and does nothing.

Here are the scripts:

```
Game_manager
{
        spawn
        {
        }
```

```
	trigger objective_counter  //Counts allied objectives completed
		{
			accum 1 inc 1
			trigger game_manager checkgame
		}

	trigger checkgame
		{
			accum 1 abort_if_not_equal 1
			wm_setwinner 0
			wait 1500

			wm_endround
		}

}

//Door to room containing Axis gold
	axisvault_script
	{
		spawn
		{
			wait 200
			constructible_class 3
		}

		death
		{
			wm_announce "Allies breached the Axis gold vault"
			setstate axisvault invisible
			trigger game_manager objective_counter
		}
	}
```

Two objectives, which must be destroyed by Allied team



This uses the exact same script as in case #1, except that the *gamecheck* script now looks for an Accum 0 value of 2. This makes sense, because each time one of the two objectives is blown up, the *objective_counter* script increases the value of Accum 0 by 1 until it has a value of 2, which signifies that both objectives have been blown up.

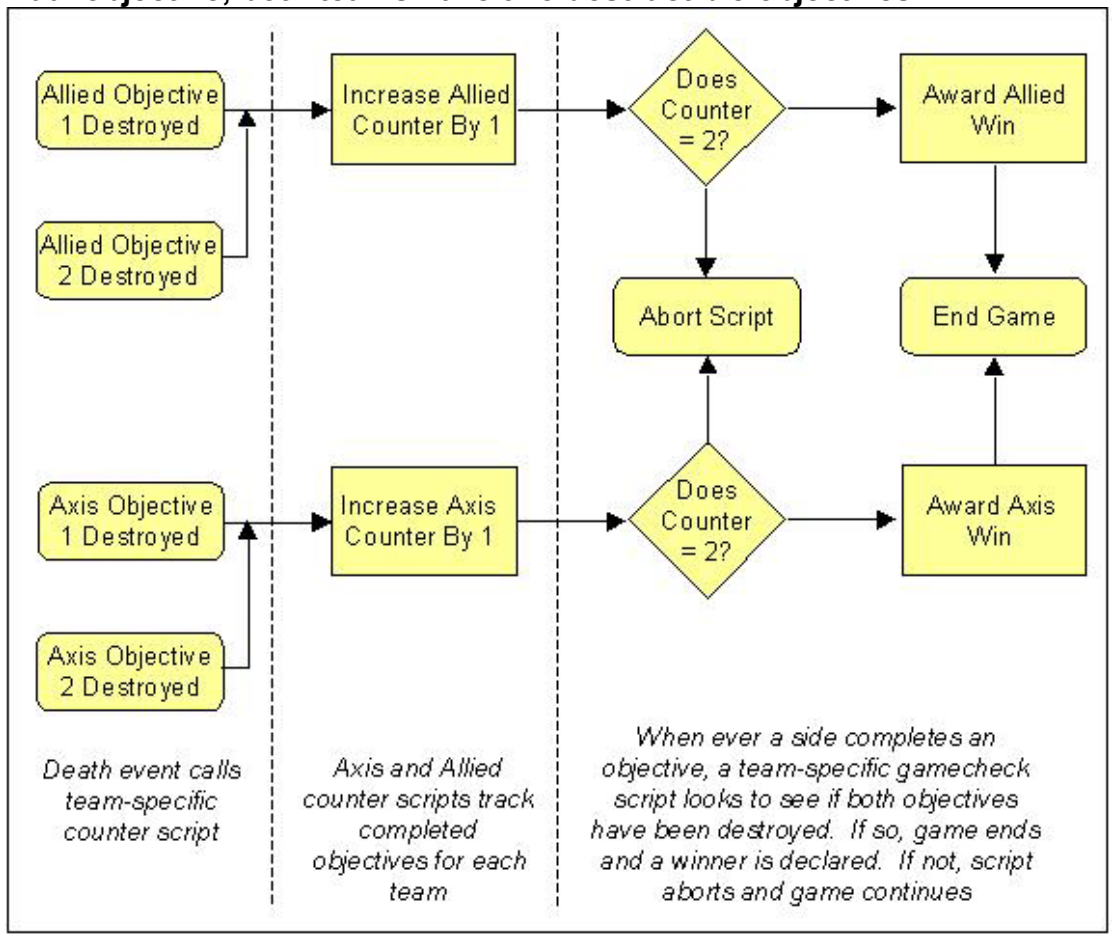
Here are the scripts:

```
Game_manager
{
	spawn
	{
	}

	trigger objective_counter  //Counts allied objectives completed
		{
			accum 1 inc 1
			trigger game_manager checkgame
		}

	trigger checkgame
		{
			accum 1 abort_if_not_equal 2
```

```
                wm_setwinner 0
            wait 1500

                wm_endround
            }

}

//Door to room containing Axis gold - Allied objective 1
    axisvault_script
    {
        spawn
        {
            wait 200
            constructible_class 3
        }

        death
        {
            wm_announce "Allies have breached the gold vault"
            setstate axisvault invisible
            trigger game_manager objective_counter
        }
    }

//Axis radar unit – Allied objective 2
    axisradar_script
    {
        spawn
        {
            wait 200
            constructible_class 3
        }

        death
        {
            wm_announce "Allies have destroyed the Radar unit"
            setstate axisradar invisible
            trigger game_manager objective_counter
```

```
        }
    }
```

Dual objective, both teams have two destructible objectives



Again, almost no fundamental change from the previous script. The only change is that we've totally duplicated *objective_counter* and *gamecheck* and made them specific to each side. That is, there is one copy of the two scripts (*allied_objective_counter* and *allied_gamecheck*) which checks to see if the Allies have won the game whenever they blow up an objective and another copy of the two scripts (*axis_objective_counter* and *axis_gamecheck*) which checks to see if the Axis have won the game whenever they blow up an objective. Now, it's just a race to see which team can blow up their two objectives first.

5. So, Scripting Isn't That Scary … What About More Advanced Objective Scripting

Sorry, I'm not going to cover that here. There are simply too many potential permutations of what people might want to do to be able to effectively cover it in a basic scripting tutorial. However, the basics are the same and if you look around a little bit, I'll be you can find all the pieces you might need in various examples. What do I mean? Let's imagine a dual objective map where each team is trying to capture a set of documents and transmit them. The twist is that there is only one transmitter, which can be built by either side, and the ability to transmit the documents is contingent on your side having built the transmitter. What would this script look like? Hell if I know, but if we diagramed the victory logic, we'd probably see that we need:

- 1 Team_CTF_Redflag entity and script
- 1 Team_CTF_Blueflag entity and script
- 1 Neutral constructible and script, which toggles the status of the trigger_flagonly_multiple

Well, we gave an example of a Team_CTF_red/blue flag script earlier; we just learned how to write single objective victory script for a dual objective map; all we're missing is the neutral constructible and script … hmmm … why don't we just go to fueldump.SCRIPT and see if I can just adapt the scripting from the axis/allied MG constructs by the tank bays. Point is, all you have to do is look around a little, and the answers are there.

6. What The Hell Is An Accum Value?

Paraphrased from posts by DJBob and SCSD_Reyalp

Accum values are the heart of your map's game logic. They are like a register or variable that are assigned or change based on the occurrence of selected game events. We used this earlier to count the number of objectives completed and based on that, determine whether or not the game was over.

They allow you to program events into your map that happen or don't happen based on the accum values and some limited set of accum math. For example, you can trigger an event if your accum value is:

- equal to a stated target (e.g., IF number of completed objectives = 2, THEN execute win routines)
- not equal to a stated target (e.g., IF number of completed objectives not = 2, THEN abort script)
- less than a stated target
- greater than a stated target
- and many more

If you've ever used MS-Excel, this is very much like writing a function with IF statements. They can also be nested to allow for multiple paths, though the scripting required is a little more complex.

Accum vs. GlobalAccum

Accum values are specific to each entity and routine. So, I could use accum 1 in my *objective_counter* script as well as in my *forward_spawn* and *command_post* scripts. This could also lead to some confusion if you use the same scriptname with multiple entities. While they share the same scriptname, each will have its own, unique set of accum values. As a result, you might end up with unexpected results.

A globalaccum can be used by multiple scripts and entities and therefore can be very useful for sharing information across scripts and entities. An example of this can be found in [insert example]

Set vs. Bitset

You can use accum values in one of two ways: 1) as a single discreet value, which ranges from x to y, or 2) as a 32-character bitset, which has a total of 32 separate on/off states.

'Set' allows you to set the value of accum x to any number. If you use set, you are limited to 8 discrete values per entity or script. An example of set might be used to count the number of completed objectives or flags captured.

'Bitset', allows you to set the value of bit (I won't explain binary math here) X to 1, and 'bitreset' allows you to set it to 0, this allows you to store 32 different on or off states. So, if all you need to know about parts something is whether they are on/off, dead/alive, etc, then you can use bitset and bitreset + the various check functions to your advantage, rather than using a single accum value for each one. An example of this might be a tank with 32 checkpoints that it needs to pass. Each can be thought of, in black and white, as passed/not passed.

I guess you can think of it this way, if you can need a range of values (up to 8) in your script, use 'set'. If your values can be expressed as yes/no or on/off states (up to 32) then you should use 'bitset'. I believe that there are lots of situations where you can really use either 'set' or 'bitset' to achieve the same results. Confused? I was.

If you're still confused, then you'll just have to experiment around.

7. Commonly Seen Commands

A lot of the commands you will see in a script are pretty self explanatory, but I thought I'd include them for the sake of completeness:

- accum x set y – Set the value of accum #x to y
- accum x inc y – Increase the value of accum #x by y
- accum x bitset y – Turn bit #y of accum #x to the ON position (e.g., 1)

- accum x bitreset y – Turn bit #y of accum #x to the OFF position (e.g., 0)
- alertentity - triggers the entity with the given targetname
- constructible class n (1-3) – Determines how much chargebar is required to finish construction and by what weapons it can be destroyed
- disablespeaker – Disables a speaker from playing
- enablespeaker – Enables a speaker to start playing
- gotomarker – Tells a mover to go to a specified point
- remove – Removes an entity from the game (e.g., the Axis Old City Spawn on Oasis and the Axis Forward Bunker Spawn on Radar)
- setautospawn "Team_Wolf_Objective description" [team] – Make players from Axis (team=0) or Allies (team=1) spawn as close to the selected spawn point as possible.
- setchargetimefactor [team] [class] [% of default time required for a full charge] – Tells game to set time required for a full recharge by [% of default time required for a full charge] for everyone who is a [class] on Axis (team=0) or Allies (team=1)
- sethqstatus [team] [status] – Identifies command post as built (status = 1) or unbuilt (status = 0) for Axis (team=0) or Allies (team=1)
- setstate [entity name] default – Make [entity name] visible
- setstate [entity name] invisible - Make [entity name] invisible
- startanimation – Starts a model animation
- trigger [routine name] [subroutine name] – Execute the script named [subroutine name], which can be found in the routine called [routine name]
- wait x – wait for x milliseconds before continuing to the next step
- wm_allied_respawntime x – Allow Allies to respawn every x seconds
- wm_announce [team] “message” – Send the text “message” to Axis only (team=0, Allies only (team=1), or everyone (team=blank)
- wm_objective_status [objective number] [team] [status] – Set the status of objective #[objective] to not completed (status=0) completed (status=1), or failed (status=2) for Axis (team=0) or Allies (team=1)
- wm_axis_respawntime x – Allow Axis to respawn every x seconds
- wm_endround – Ends the game
- wm_number_of_objectives x – Set number of objectives to X
- wm_removeteamvoiceannounce [team] “[name of voice over]”
- wm_set_defending_team [team] – Set Axis (team=0) or Allies (team=1) as the defending team
- wm_set_main_objective [objective number] [team]
- wm_set_round_timelimit x – Let the game play for X minutes
- wm_setwinner [team] – if the time limit expires, then award the win to Axis (team=0) or Allies (team=1)

- wm_teamvoiceannounce [team] “[name of voice over]” – Play the .wav file associated with [name of voice over] to Axis only (team=0, Allies only (team=1), or everyone (team=blank). File associations can be found in the .SOUND file

8. Contributors

- Hummer
- Mean Mr. Mustard
- The budding mappers from team Wolfjaeger
- Tons of people over at the Splash Damage forums