**Looking to get into modding and map making? Check out our editing documentation.**

**Contents**

* Introduction
* Player Spawnpoints
* Construction System
o Construction Classes
o New Entities
o Scripting
o Neutral Team
o Multiple Stages
* Command Posts
o Entity Requirements
o Scripting for Command Posts
* Health & Ammo Cabinets
o Entity Requirements
* Command Maps
* Dual Objectives

* Objective Cameras
* Spline Path System
* Moving Object Scripting
o Miscellaneous Entities
* Environmental Effects
* Foliage Technology
* Surfaceparms
* Implicit Shaders
* Compiling Maps
o Sample Compile Batch File
* Useful Console Commands
* Trace Maps
* Arena File
* Campaign File
* Speaker Editor
* Appendix A: Scripting Commands
* Appendix B: Entities
* Appendix C: Common Textures
* Glossary

## Introduction

This document is a basic primer for creating maps for Wolfenstein: Enemy Territory. It assumes that you have downloaded and installed the editing tools correctly and know how to use the GtkRadiant editor. So that all future upgrades of the GtkRadiant editor do not affect the Enemy Territory editing tools the editor executable has been renamed to SDRadiant.

There are plenty of tutorials on the Internet on how to use the GtkRadiant editor. A good place to start would be the www.qeradiant.com site. The editor also has an extensive collection of links in the help documentation. (Press F1 inside of the editor)

At the back of this documentation is a glossary which defines certain technical terms used through out this manual. At relevant points certain words are hyperlinked to the glossary so that it is easier to understand what is being said.

This manual is not a beginner's guide to mapping.

Enemy Territory is based on existing RtCW technology and assumes the reader is familiar with the previous system. This manual only describes the new features of Enemy Territory. There are plenty of forums and tutorials which can help with the learning of the basics of mapping.

## Key Differences between Enemy Territory and RtCW

Enemy Territory introduces a lot of new features and gameplay strategies. If you are planning on converting an existing RtCW map to Enemy Territory then consider the following points:

* The spawning system is simpler and uses fewer entities.
* Command posts. (either neutral or individual)
* Health/ammo cabinets. (Can be linked to constructions if necessary)
* Map objectives should be set up to use the limbo camera system.
* The construction system is a critical component in Enemy Territory and all maps should be designed with this feature in mind. (This a feature of the Engineer class only)
* Command map - all maps need to define the co-ordinates of the command map so that the map will load correctly and work with the limbo menu.
* Maps loaded from the in-game menu system need an arena file.
* Enemy Territory supports script moveable objects. (vehicles)
* Environmental effects like rain/snow.
* New fog system which works with multiple pass shaders and lightmapped terrain.
* Support for model foliage via a special shader. (Compiler option)
* A new class has been added to Enemy Territory called CovertOps. Team doors should be added to critical choke points in maps so some objectives can be bypassed.
* The terrain system can have external lightmaps and decals.
* All maps need a special "trace map" for the mortar and environmental effects to work.
* The shader language supports implicit shaders. This can affect how models and surfaces are lit in the game.
* New scripting commands to enhance the existing RtCW system.
* New compiler and Gtk plug-ins and features.
* Land Mines and new surface parameters.

## Player Spawnpoints

The player spawning system in Enemy Territory has one spawn entity per team with 2 spawnflag values as follows:
Classname: team_CTF_redspawn, team_CTF_bluespawn
Spawnflag: invulnerable, startactive

Generally an Enemy Territory map will have an initial spawn area for both teams and possibly one or more capture/autospawn areas.

The initial spawn areas are made up of team-specific spawn entities with the spawnflag values both set so that players can start the map without being killed immediately.

The capture / autospawn areas will have Axis / Allied player spawn entities and both sets should be given a targetname key. The targetname key is used so that the game can switch the spawn entities from

one state to another. Initially one set will be active and the other set will be dormant until triggered via the targetname key.

Located near all spawnpoints in the game is a team_WOLF_objective entity which appears on the command map, determines who owns the spawnpoint initially (If you select both spawnflag values then it WILL produce erratic results) and has a description key used by the autospawn feature.

If the spawnpoints are spread out across a wide area or in several rooms then the team_WOLF_objective entity should be located somewhere in the middle of the spawnpoints. When the game spawns a player it will choose the next available spawnpoint closest to the team_WOLF_objective entity. Some spawnpoints will not be used unless you have a lot of players spawning at once. All spawn areas should be designed to cope with 16 players spawning at once on each side.

The autospawn feature is designed to start players at the most forward spawn point on the map for the relevant team. As spawn areas are captured temporarily or permanently the script system needs to keep the autospawn feature up to date.

The description key is used by the autospawn system to switch the state of the team_WOLF_objective entity from one team to the next. The autospawn feature is totally controlled via the script and does not switch spawnpoints around either. (You still need to triggered the spawnpoint entities as before)

The setautospawn command (used in the script) uses the description key of the team_WOLF_objective entity to determine which spawn area to use, followed by a 0/1 value for which team. 0 represents Axis and 1 is for the Allied. There is no default for the setautospawn command so it needs to be setup at the beginning of the game, usually located in the game_manager routine of the script.

Please note that Enemy Territory maps still require the game_manager entity to be present in all maps. The map script usually starts with the game_manager routine because it always exists in every map.

## Spawn area considerations

Unfortunately there is a certain type of player who insists on disrupting online games as much as possible. These tactics range from spawn camping to door blocking and sadly these things can ruin a game for all people involved.

There is currently no way of "bumping" players out of the way in RtCW / Enemy Territory and some players on public servers insist on disrupting games by blocking doorways to / from spawn areas.

When designing spawn areas we considered the following points:

* If you can spare the space then create all spawn areas with at least two entrance/exits so one player cannot block a whole spawn area.
* If you have only one entrance / exit to a spawn area then make it at least 3 players wide in order to prevent 1 player from blocking the doorway.
* Make sure that neither team can place landmines on the floor of the spawn area.
* Place all spawn entities as close as possible against a wall so that enemy players cannot stand behind the spawn entities and spawn camp.
* If possible place spawn areas under cover so that mortar / artillery strikes cannot affect new players joining the game.
* Remember to make sure all spawn entities have the spawnflag set to invulnerable so that new players are not killed instantly upon spawning into the game.
* Try creating spawn areas high up so that the new players have a height advantage over the opposing forces.
* Create multiple spawn locations for critical areas to minimize spawn camping and door blocking.

No spawnpoint is entirely spam-proof. Regardless of how many features you build into a spawn area to prevent disruptive game play, some players will always find a way to spam it. All you can do is to try and prevent as many of the most obvious problems associated with spawn areas as possible.

## Construction System

### Intro

The design goal for the construction system was to introduce basic RTS style game-play to the multiplayer Wolfenstein universe. The construction system was designed to be as flexible as possible by

having support for multiple stages which can be created from either game models or brushwork.

The ability to build constructions is an enhanced feature of the engineer class. Engineers are able to construct objects within maps such as bridges, guard towers and gun emplacements in predefined locations. The construction locations should be marked by the crates model so that everyone can recognise a construction point in the map.

Each construction will have a construction class which defines how much 'charge bar' time is required to complete each stage of the construction. The speed at which an item can be constructed can be increased with additional Engineers.

While the construction is being built a translucent representation of the object will appear and then become solid when built. If a player is in the construction area while being built they will be warned to leave the area before the construction is finished. Players failing to leave the construction zone will be killed the instant the construction is complete.

A construction object which has not finished or been touched by an engineer will decay after 30 seconds (current default) to its previous deconstructed 'crates' state. Multiple stage constructions will decay back to their previous built stage.

Once the construction has been built by one team, only the opposing team can damage it. The construction class of the construction object dictates how it can be destroyed. Friendly fire will not apply to constructed objects.

## Construction Classes

Most of the construction system's functionality is located in the scripting system and only the basic parameters are present in the maps. This could present a problem for Level Designers who have had very little experience of scripting but the existing maps do offer a lot of workable examples to study if you're unsure.

The construction system has 6 variables which can affect the way a constructible object will work in-game. These variables are also organized into 3 presets so that construction objects can be consistent across all maps. The main difference between each of the construction presets is the 'charge bar' times and the type of explosives that can damage the object.

The largest construction objects in the game were given the highest 'charge bar' value so that it would encourage the engineers to work together on completing objectives. The preset value of a construction should reflect its importance to the overall map objectives. The only exception to the rule is the escorted tanks which have a high health value and can be damaged by all explosives.

All constructible items must have the construction class (preset) defined upon spawning in the map. Further construction parameters can be changed after the construction class has been defined. The following table highlights the presets and what values they use.

| Construction Class (preset) | Charge Bar Req. | Construct XP Bonus | Destruct XP Bonus | Health | Weapon Class | Duration (msec) | Icon |
|---|---|---|---|---|---|---|---|
| 1 | 0.5 | 5 | 5 | 350 | 1* | 2500 | Grenade |
| 2 | 1 | 7.5 | 7.5 | N/A | 2* | 5000 | Satchel Charge |
| 3 | 1.5 | 10 | 10 | N/A | 3* | 2500 | Dynamite |

1* = All Explosives (Dynamite, Satchel Charge, Grenades, Panzerfaust, Mortars, Air Strike, etc.)
2* = Dynamite, Satchel Charge
3* = Dynamite Only

Please note that constructible_health only applies to construction class 1 because the other values are instantly destroyed by satchel or dynamite explosions.

The script commands below match the columns of the table above.

constructible_chargebarreq
constructible_constructxpbonus
constructible_destructxpbonus
constructible_health
constructible_wea ponclass
constructible_duration

For example, if a tank takes a lot of damage and can be damaged by anything from grenades upwards, the following spawn settings would be used:

```
example_tank
{
        spawn {
            constructible_class 1
            constructible_health 1200
        }
}
```

## New Entities

All constructible objectives have a core set of entities which are used everytime regardless of construction configuration. Each construction objective must contain at least one func_constructible and one trigger_objective_info entity both linked together.

Neutral team construction objectives contain two func_constructible entities because both teams cannot own the same construction entity. The trigger_objective_info entity targets both func_constructible entities.

### func_constructible

The func_constructible is the primary entity which specifies which team can build or destroy the construction. The func_constructible is a brushwork entity and must contain an origin brush; otherwise it will cause problems with the planting of dynamite.

When a team plants explosives next to a construction objective the func_constructible defines where the explosives can be planted. Because some explosives have a wide damage radius be careful not to create func_constructible entities too close to each other.

The func_constructible entity has the following keys:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
track: Construction group function.
wait, score, health: Not used anymore (replaced with script commands).
constages: List of construction func_brushmodel entities.
constages: List of destruction func_brushmodel entities.

The track key functions like a group name. All entities with the same track key as the func_constructible will be constructed at the same time. This is important if you want players to be warned to leave construction areas during the transparent stage of construction. If you set entities via the script to the transparent construction state then players will not get the warning message for construction.

The func_constructible entity has the following spawnflag options:
start_built: Starts built.
invulnerable: Cannot be destroyed by explosives.
axis_constructible: Can be built by the Axis.
allied_constructible: Can be built by the Allied.

If the func_constructible entity is not assigned to a team then the map will fail to load. If both teams are selected on the spawnflag then the default team will be Axis.

Hint: Keep in mind that some explosives have a very wide destructive radius and can be placed some distance away from the func_constructible and still be effective at destroying the construction objective.

## trigger_objective_info (TOI)

This entity is used to represent other entities (func_constructible, func_explosive, misc_commandmap_marker ) on the command map. The state of the construction objective determines what icons are displayed on the command map.

The trigger_objective_info is a brushwork entity and must contain an origin brush; otherwise it will appear half way between its current map position and "0 0 0" map position on the command map. The brushwork area of the trigger_objective_info entity specifies where the player can build, a 'pliers' hint icon and an on screen message.

Each map is limited to a maximum of 18 trigger_objective_info entities. This total usually includes command posts, health / ammo cabinets, onscreen hint areas and all constructible objectives. Some constructible objectives may only exist in certain game types. (The map 'Fueldump' hits the TOI limit)

The trigger_objective_info entity has the following keys:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
shortname: Name on the command map.
objflags: Used for pulsating icons on the command map.
track: Onscreen text message for what you are near.
customimage: Replaces the default icon for both teams.
customalliedimage:: Replaces the default icon for the allied team only.
customaxisimage: Replaces the default icon for the axis team only.
infoAxis/infoAllied: Not used anymore, do not use.
score: Replaced with script commands.

The trigger_objective_info entity has the following spawnflag options:
axis_objective: Only works with 'func_explosive' entities.
allied_objective: Only works with 'func_explosive' entities.
tank: Will use a tank icon on the command map.
is_objective: Will use a standard objective icon on the command map.
is_healthammocabinet: Will use a health / ammo cabinet icon on the command map.
is_commandpost: Will use a command post icon on the command map.

If the trigger_objective_info entity does not target any other entities it will still exist in the map. The entity will still produce "You are near ." messages and if the axis/allied objective spawnflags are set, it will still interact with explosives.

Using the script command setstate the trigger_objective_info entity can be switched on and off in the map for different game types. Once the entity has been switched off all relevant command map icons disappear as well.

Hint: Try and keep the trigger_objective_info entity some distance away from the func_constructible entity, so that any Engineers building the construction cannot stand in the construction area and get killed when it's complete.

### Single Team

The flag and construction crates mark the location of where the Engineers can build a constructible objective. Each team has specific skins for the flag and construction crates so that they are easily recognisable by all players.

Construction crates and skins: ( misc_gamemodel )

models/mapobjects/cmarker/cmarker_crates.md3
models/mapobjects/cmarker/axis_crates.skin
models/mapobjects/cmarker/allied_crates.skin

Flag and skins: ( misc_gamemodel )

models/mapobjects/cmarker/cmarker_flag.md3
models/mapobjects/cmarker/axis_cflag.skin
models/mapobjects/cmarker/allied_cflag.skin

Additional key/values for the flag model:

modelscale: 0.4
frame: 0.4 (Total amount of animation frames)
spawnflags: 2 (Start entity animated)

The flag and construction crate models are made from misc_gamemodel entities with targetname and scriptname keys so that they can be referred to by the script.

Create a rough set of clip brushes which mirror the shape of the construction crates model. Then create an origin brush, select all the previously created clip brushes and converted them into a func_static entity. Select the func_static entity and set the targetname and scriptname keys so that it can be referred to by the script.

Create the constructible objective from brushwork and position it a reasonable distance from the construction crates model. Create an origin brush, select the constructible objective brushwork and convert into a func_constructible entity.

Note: If you want to create the constructible objective from a model then use clip brushes for the func_constructible instead. It's important that the func_constructible entity is made from brushes because the explosive's radius has to hit something.

Select the func_constructible entity and set the targetname and scriptname keys so that it can be referred to by the script and targeted by the trigger_objective_info entity. Set the spawnflags to the relevant team which can construct the objective. (Only one team should be selected.)

Finally create a trigger_objective_info brushwork entity around the construction crates model and target it at the func_constructible entity. Allow enough room for the Engineers to be able to move around the construction while building. Setup all the relevant keys and spawnflags for the trigger_objective_info entity as [#toi documented above] .

## Scripting

All constructive objectives require scripting to work. In order for Level Designers to create maps for Enemy Territory they will need to learn how to write scripts. This may be a simple case of cut and paste from the original scripts or may involve creating new script content.

In order to explain scripting this documentation defines certain words with regards to scripting. Wherever possible certain words/phrases have been hyperlinked to a glossary.

Here follows a quick primer on the how, where and what of scripting. This is by no means a complete guide to scripting but all of the maps that come with the original game do have script files which can be used as examples. Open them up and have a look .

===Back to Basics=== The script file is located in the maps directory with a ".script" file extension. The script filename has to be the same as the map filename. (Eg. 'oasis.bsp', 'oasis.script')

The script file is made up of routines which define what various entities do and how they react to game events. Each entity in the map has the possibility to run a routine in the script file. To enable an entity to run a routine in the script file it needs a targetname and scriptname key with relevant values.

For example: targetname, "alliedmgnest" and scriptname , "alliedmgnest".

It's recommended that the targetname and scriptname values are the same so it's easier to debug the script and map for possible errors. (Technically not all of the entities require the targetname and scriptname keys to be defined but it's recommended)

Each entity routine is broken down into functions which are triggered by various game events. For example: The function spawn is triggered when the entity is spawned into the map at the beginning of the game.

Certain functions exist for all entities regardless of what type they are. The standard functions are spawn, trigger, pain and death. Some entities have further functions which are specific to what they can do in game.

The script can run several entity routines at once, but each routine can only run one function at once. For example if any function uses a wait command it will stop all other functions for that entity routine. Because of this limitation several entities maybe required in order to perform several tasks at once.

Each entity routine can use / define up to 8 local variables. These local variables cannot be referenced outside of the defining routine. The script system also supports up to 10 global variables which can be used / referenced by any routine / function in the same script.

Script variables can be used to store values which can determine if game events have been completed or what state they are currently in. For example in the map 'Radar' a variable is used to count the total amount of radar parts taken from the axis base.

### Construction Script Functions

The func_constructible entity has 5 basic functions as defined below:
spawn:

This function is run the first time upon starting the map. All spawn functions should have a wait command before processing any other commands because not all entities are initially ready when the map is first loaded.

All constructible items in Enemy Territory must setup their construction class when they spawn. This determines how much 'charge bar' time to be used while constructing. (There are currently 3 types of [#conclasses construction classes] defined.)

The construction materials should be made visible (default) and the main construction elements made invisible.
buildstart:

This function is called once the construction has been started. The various construction elements are set up to be displayed in a translucent state. This shows all players where the construction is going to be built.
built:

This function is called once the construction has been finished. All construction materials are made invisible and the main construction elements are visible.
decayed:

If the construction has not been touched by an engineer for 30 seconds (current default) then this function is called. The construction elements are made invisible. If this function is not defined then the construction will remain in a translucent state.
death:

This function is called once the construction has been destroyed. All construction materials are made visible and the main construction elements are made invisible.

If the construction involves a MG entity then it must be repaired by the script at this point. If the MG entity was damaged during the destruction of the construction object then the MG will be built in a damaged state.

The backbone of the construction system is one script command called setstate . This command allows entities to be switched into 3 different states of 'default / invisible / underconstruction'. Combined with the various functions defined above the construction system is not overly complex to setup.

### Neutral Team

A neutral construction is no different to a single stage construction except there is one for each team, both linked to one trigger_objective_info entity. The func_constructible entity can only be associated to one team so neutral constructions need two constructible entities.

It is recommended that both team constructions are different so that the players can easily tell from a distance who built the construction in the first place. The trigger_objective_info entity controls which func_constructible is built and which script routine is active.

### Multiple Stages

The construction system can support up to 3 different stages for construction (stage1, stage2, final) and 2 different stages for destruction. Depending on which happens to the func_constructible depends on which stage will be active as the diagram below illustrates:

The best way to make a multiple staged construction is to create the final object first and set it up like a single staged construction. Once you have a single stage construction working then add more stages to it.

Multiple stage constructions use two new keys for the func_constructible entity.
constages: List of construction func_brushmodel entities.
constages: List of destruction func_brushmodel entities.

All additional construction/destruction stages need to be setup as func_brushmodel entities with targetname key. The value of the targetname key will appear in the constages or desstages lists.

For example if the construction stage 1 and 2 entities have targetname key values of 'constage1' and 'constage2', then the constages key for the func_constructible will have a value of 'constage1;constage2;'

Note: The constages, desstages lists must be separated by semicolons and have no spaces. Only need to use these keys if creating multiple stage constructions.

Wherever the func_constructible is in the map, all the construction/deconstruction stages will appear as well. It does not matter if you create the stages elsewhere in the map or at the same space, all stages will be displayed at the same location.

The last stage in a multiple construction is always referred to as 'final' in the script. If you are only creating a 2 stage construction then the script will use 'stage1' and 'final' only.

An Example Multiple Stage Construction Script

```
//===========================================
// Test multiple stage construction
//===========================================
constructfinal
{
	spawn
	{
	   wait 200
	   constructible_class 1

	   setstate constage1 invisible
	   setstate constage2 invisible
	   setstate constructfinal invisible

	   setstate desstage1 invisible
	   setstate desstage2 invisible

	   setstate construct_stage1 default
	   setstate construct_stage2 default
	   setstate construct_stage3 default
	}

	//===========================================
	// Stage 1
	//===========================================
	buildstart stage1
	{
	   wm_announce "Stage 1 started ..."
	}
	built stage1
	{
	   setstate construct_stage1 invisible
	   wm_announce "Stage 1 BUILT!"
	}
	decayed stage1
```

```
{
    setstate construct_stage1 default
    wm_announce "Stage 1 DECAYED!"
}
death
{
    setstate construct_stage1 default
    wm_announce "Stage 1 DEATH!"
}
```

Note: The 'constructfinal' script routine is the func_constructible entity. The 'constage1', 'constage2', 'desstage1' and 'desstage2' are the targetname key values for the func_brushmodel entities.

The 'construct_stage1', 'construct_stage2' and 'construct_stage3' are the construction materials. (The crates model and clip brushes)

```
//===========================================
    // Stage 2
    //===========================================
    buildstart stage2
    {
        wm_announce "Stage 2 started ..."
    }
    built stage2
    {
        setstate construct_stage2 invisible
        wm_announce "Stage 2 BUILT!"
    }
    decayed stage2
    {
        setstate construct_stage2 default
        wm_announce "Stage 2 DECAYED!"
    }
    destroyed stage2
    {
        setstate construct_stage2 default
        wm_announce "Stage 2 DESTROYED!"
    }

    //===========================================
    // Stage 3
    //===========================================
    buildstart final
    {
        wm_announce "Stage 3 started ..."
    }
    built final
    {
```

```
        setstate construct_stage3 invisible
        wm_announce "Stage 3 BUILT!"
    }
    decayed final
    {
        setstate construct_stage3 default
        wm_announce "Stage 3 DECAYED!"
    }
    destroyed final
    {
        setstate construct_stage3 default
        wm_announce "Stage 3 DESTROYED!"
    }
}
```

Note: Each stage has four functions which are run depending on the state of the func_constructible entity. Some of the functions in this example script don't actually do anything but are included so that you can use them if you want.

The final stage of construction is always called 'final', so if you require only 2 stages of construction then only use 'stage1' and 'final' sections of the above sample script.

## Command Posts

The command post is an essential object on the map which should be constructed and carefully defended because of its tremendous benefits to 'charge bar' times. It should be placed with care on a map because it can tip the scales of balance for one team over another.

The command post comes in two different variants: individual team or neutral team. The neutral team version is the most complex to set up because it involves so many different components. The command post totally relies on scripting to work, so using the existing scripts as a template will save you loads of time.

Closed model

Axis
models/mapobjects/radios_sd/compostaxisclosed.md3
models/mapobjects/radios_sd/compostaxisclosed.skin
Allied
models/mapobjects/radios_sd/compostalliedclosed.md3
models/mapobjects/radios_sd/compostalliedclosed.skin
Modelscale 1.5

Built model

Axis
models/mapobjects/radios_sd/compostaxisopened.md3
models/mapobjects/radios_sd/compostaxisopened.skin
Allied
models/mapobjects/radios_sd/compostalliedopened.md3
models/mapobjects/radios_sd/compostalliedopened.skin
Modelscale 1.5

Damaged model

Axis
models/mapobjects/radios_sd/compostaxisdamaged.md3
models/mapobjects/radios_sd/compostaxisdamaged.skin
Allied
models/mapobjects/radios_sd/compostallieddamaged.md3
models/mapobjects/radios_sd/compostallieddamaged.skin
Modelscale 1.5

## Entity Requirements

The command post is essential a func_constructible entity with three different models for visual representation. The command post models are setup as game_models so that they can be swapped in and out of the game easily. Each separate section of the command post has targetname and scriptname keys so that they can be referred to by the script system.

All constructible items in Enemy Territory must have a trigger_objective_info entity. When the player is near the trigger_objective_info entity a message is displayed on screen. Whatever is defined in the track key is used as the onscreen text message.

Important note: The trigger_objective_info is a brushwork entity and must contain an origin brush; otherwise it will appear half way between its current map position and '0 0 0' map position on the command map.

In order for the command post to appear on the command map as a constructible item it must be linked to a certain entity type. The func_constructible entity tells Enemy Territory who owns the construction (Allies/Axis), what state it starts in and if it can be destroyed or not.

The func_constructible entity defines where the explosives can be put by the player so that the construction is destroyed. Because of this restriction the func_constructible entity is usually the same physical size as the command post when built. The clip brush used for the 'built' command post game model is setup as a func_constructible entity.

For each command post game model a rough set of clip brushes will need to be created to mirror the shape of the model. All the command posts in the current Enemy Territory maps were setup with roughly the correct metal/wood clip brushes.

Once all the clip brushes have been created an origin brush will need to be added and then everything converted into a script_mover entity. Because the clip brushes for the 'built' game model were used as a func_constructible entity, only the 'closed' and 'damaged' clip brushes will need to be converted to script_mover entities.

## Entity summary

command post game models (Allied, Axis and Neutral)
1 trigger_objective_info entity
1 clipbrush func_constructible entity
2 clipbrush script_mover entities
1 info_limbo_camera (all command posts are secondary objectives)

All the above entities need targetname and scriptname keys and with that many inter-linked entities to keep track of it can get confusing. The command posts behave in a similar way across most of the maps and if the key/values are standard then it makes referring to them in the script easier. Here is a list of standard names that were used with the command posts.
Command post models: _radio__model
Clipbrush entities: _radio_
Trigger_objective_info entity: hqradio_toi

## Scripting for Command Posts

The primary entity for controlling the construction/deconstruction of the command post is the func_constructible entity. This entity has many functions as summarised below:
spawn:

This function is run the first time upon starting the map. All spawn functions should have a wait command before processing any other commands because not all entities are initially ready when the map is first loaded.

All constructible items in Enemy Territory must setup their construction class when they spawn. This determines how much 'charge bar' time to be used while constructing. (There are currently 3 types of construction classes defined.)
setup: This is a custom trigger function which sets the initial speed at which the 'charge bar' is replenished. (Called by the spawn function)
buildstart: This function is called once the construction has been started. The 'destroyed' and "closed" game models are hidden and the 'built' game model is displayed in a translucent state.
built: This function is called once the construction has been finished. The 'built' game model is displayed and the command post features are enabled. The 'charge bar' times are adjusted, team VO updated, command post status changed and the secondary objective completed/failed for the relevant team.
decayed: If the construction is not completed within 30 seconds (current default) then this function is called. The construction will be reset back to a 'closed' or 'damaged' game model state.
death: This function is called once the construction has been destroyed. The 'destroyed' game model is displayed and the command post features are disabled. The 'charge bar' times are adjusted, team VO updated, command post status changed and the secondary objective reset. (The tick/cross overlay is removed)

The clip brush script_mover entities are mainly controlled from the func_constructible entity. They control the relevant game_models entities and switch states depending on what is happening with the command post.

Finally the 'built' game model is used for enabling/disabling the command post features. The 'charge bar' times are adjusted for all classes and the VO is updated for the relevant team. The sethqstatus script command will inform all team members about landmines being close by. (Command post feature)

When the command post is constructed / damaged a sound speaker entity (positioned infront of the command post) which is enabled / disabled via the script using new commands. All the speaker sounds in the maps where placed by the new speaker sound system built into the game executable.

## Health & Ammo Cabinets

The health and ammo cabinets are useful for creating fall back points in maps. These orchards of health and ammo are very important for both teams. The cabinets should be placed in areas that cannot be defended too easily.

The health and ammo cabinets have two model states: broken and operational. They could be turned into a construction item if necessary because all states exist (empty, built and damaged). The cabinets do not rely on scripting because most of their functionality is provided via the code.

Functional model

Entities: misc_cabinet_supply, misc_cabinet_health

The above entities will create the correct models in game with all the ammo and health boxes attached. The empty stands exist as models as follows:

Folder: models/mapobjects/supplystands/
Models: stand_health.md3, stand_ammo.md3

Broken model

The broken health and ammo cabinets are created in game with a misc_gamemodel, using a model key pointing at the md3. A targetname key will also allow the models to be added or removed from the

game.

Folder: models/mapobjects/supplystands/
Models: stand_health.md3, stand_ammo.md3

### Entity Requirements

The ammo cabinet is defined by a point entity called misc_cabinet_supply . This entity will generate the correct game model for the ammo boxes to be stored on. The game will automatically keep re-stocking the ammo over time.

The ammo cabinet is surrounded by a common/trigger brush entity called trigger_ammo. This entity is the area in which the player has to stand in order to receive ammo. The rate and total amount of ammo the cabinet will give out is defined by the ammototal and ammorate keys. This entity is targeted at the misc_cabinet_supply entity.

The health cabinet is defined by a point entity called misc_cabinet_health . This entity will generate the correct game model for the health boxes to be stored on. The game will automatically keep re-stocking the health over time.

The health cabinet is surrounded by a common/trigger brush entity called trigger_heal. This entity is the area in which the player has to stand in order to receive health. The rate and total amount of health the cabinet will give out is defined by the healtotal and healrate keys. This entity is targeted at the misc_cabinet_health entity.

In order for the health and ammo cabinet to appear on the map a trigger_objective_info entity is required. When the player is near the trigger_objective_info entity a message is displayed on screen. Whatever is defined in the track key is used as the onscreen text message.

Important note: The trigger_objective_info is a brushwork entity and must contain an origin brush; otherwise it will appear half way between its current map position and "0 0 0" map position on the command map.

The trigger_objective_info entity is targeted at a misc_commandmap_marker entity. This entity allows the trigger_objective_info entity to appear on the command map. The trigger_objective_info needs to have the correct spawnflag set 'is_healthammocabinet' so that the correct icon is displayed on the command map.

For each cabinet a rough set of clip brushes will need to be created to mirror the shape of the model. Once all the clip brushes have been created an origin brush will need to be added and then everything converted into a script_mover entity.

#### Entity summary

1 misc_cabinet_supply entity (Will create the correct game model as well)
1 trigger_ammo brushwork entity (The trigger brush for ammo)
1 misc_cabinet_health entity (Will create the correct game model as well)
1 trigger_heal brushwork entity (The trigger brush for health)
1 trigger_objective_info entity
1 misc_commandmap_marker entity
2 clipbrush script_mover entities

## Command Maps

The command map is a critical element of the game offering players the chance to keep track of objectives, other team players and where health/ammo requests are coming from. The command map is usually a 512 x 512 pixel image on which all other information is displayed.

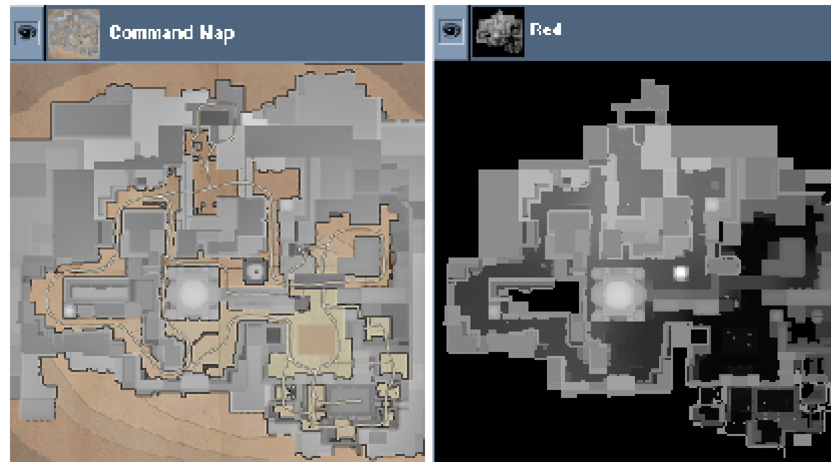
The command map can be 256 pixels square up to 1024 in size, the game will resize the image regardless. The command map picture (stored as a .tga) has to be created externally by a paint program of your

choice. The grid size and co-ordinates are worked out from the min/max keys in the worldspawn entity for the map.

Key mapcoordsmins represent the top left corner (x / y map editor values).
Key mapcoordsmaxs represent the bottom right corner (x / y map editor values).

One of the easiest ways to do that is to create a square brush under your map. (Make sure it's square!) Also make sure that everything that has to be on the command map is within the bounds of the square brush. Then move the cursor over the top left corner of the brush and you will get the x / y mapcoordsmins (first two values) from the little coordinate line in Radiant (bottom right). Repeat for the x / y mapcoordsmaxs with the bottom right corner of the brush and you're done.

A quick way to create a temporary command map is to use the tracemap command and take the RED channel image and re-scale it to a 512x512 image. Here is an example of the final command map image and the RED channel image for the map 'Goldrush'. (As a temporary image for the command map the red channel has basic similarities)




## Adding the command map to the game

For this example of how to add a command post to the game I will assume your new map is called 'mytest'. So that your map does not conflict with others maps all the files for the command should be named after your map name.

First create a square .tga image and save it as 'mytest_cc.tga' in the 'levelshots'directory. (512x512 pixels should be enough.)
Create a new shader called 'mytest_levelshots.shader' in the 'scripts' directory.
Open the new shader file and create two new entries as follows:

```
levelshots/mytest_cc_automap
{
	nopicmip
	nocompress
	nomipmaps
	{
	   clampmap levelshots/mytest_cc.tga
	   depthFunc equal
	   rgbGen identity
	}
}
```

```
levelshots/mytest_cc_trans
{
	nopicmip
	nocompress
```

```
        nomipmaps
        {
            clampmap levelshots/mytest_cc.tga
            blendfunc blend
            rgbGen identity
            alphaGen vertex
        }
}
```

The 'automap' shader is used inside of the compass on the HUD and the 'trans' shader is used for the pop-out version of the command map. The command map displayed on the Limbo menu does not require any shader features.

### Icons on the command map

At first glance the command map can look confusing but all of the relevant information is displayed in small icons which if necessary can be filtered out. The filter controls on the limbo menu also apply to the 'pop out' in game command menu.

#### Spawns

The spawnflag icons will appear on the map where the team_WOLF_objective entities are located. The text displayed on the command map for the spawn point is from the entity key description.

#### Destructible

If the Allied/Axis team have an objective which involves deconstructing something then a dynamite icon will appear on the map where the relevant trigger_objective_info entity is. Once the objective has been destroyed then the icon will disappear from the command map.

#### Constructible

If the Allied/Axis team have an objective which involves constructing something then a collection of crates icon will appear on the map where the relevant trigger_objective_info entity is. Once the objective has been built then the icon will disappear from the command map.

#### Command Posts

Initially the command post will appear as a construction objective and then turn into a command post icon. In order for the game to know it is dealing with a command post object the trigger_objective_info entity needs to have the is_commandpost spawnflag set.

#### Health and Ammo Cabinets

The health/ammo cabinets will always appear on the command map unless they are linked to another construction. (Eg. the tank bay in Fueldump has the health/ammo cabinet linked to the command post) In order for the game to know it is dealing with a health/ammo cabinet the trigger_objective_info entity needs to have the is_healthammocabinet spawnflag set.

Custom Icons

All custom icons are stored in the 'gfx/limbo' directory and prefixed with 'cm_'.

The custom icon will replace both construct/destruction states for the object. Normally the system will use crates for construct and dynamite for destruction. Because of this limitation the custom icons where mainly used for flag objects like god bars and radar parts or func_explosive entities like the wall in the old city for 'Oasis'.

The following keys are used on the trigger_objective_info entity to specify new custom icons for the command map:
customimage: replaces the icon for both teams
customalliedimage: replaces the icon for the allied team only
customaxisimage: replaces the icon for the axis team only

## Dual Objectives

The map 'Radar' has the Allies trying to steal 2 experimental radar parts from the Axis base and delivering them to a waiting truck. This is a dual objective map created with a few entities and some scripting.

The command map can be 256 pixels square up to 1024 in size, the game will resize the image regardless. The command map picture (stored as a .tga) has to be created externally by a paint program of your choice. The grid size and co-ordinates are worked out from the min/max keys in the worldspawn entity for the map.

The radar dish is a misc_gamemodel entity with special animations for the dish rotation and fold up/down. The dish is metal clipped as close as possible to the fold down animation and a big player clip is put around the whole dish.

The radar box is a team_CTF_redflag entity with a custom model. The radar dish is surrounded with a trigger_objective_info brushwork entity that targets a misc_commandmap_marker entity.

The truck is a misc_gamemodel entity. The radar boxes exist in two states in the back of the truck. One version is the standard model skin and the other one is a red coloured transparent shader.

The back of the truck has a brushwork trigger_flagonly_multiple entity which is setup to receive only red flags. (The stolen radar parts)

The whole of the truck is surrounded with a trigger_objective_info brushwork entity that targets a misc_commandmap_marker entity.

### Scripting

The script is controlled by the two radar boxes (objectives). The team_CTF_redflag entity has 3 special functions: 'trigger stolen', 'trigger returned' and 'trigger captured'.

When the player touches the team_CTF_redflag entity the game will call the function 'trigger stolen'. The script will make the radar box model by the radar dish invisible and fold down the radar dish model.

If the player is killed and the opposite team touches the fallen radar box the game will call the function 'trigger returned'. The script will make the radar box model by the radar dish solid again and fold up the radar dish model.

When the player touches the trigger_flagonly_multiple entity with the correct flag type the game calls the function 'death'. The script sets the map objective counter and checks if the game is finished or not. (It's certainly possible to have more than 2 objective flags)

The game also calls the function 'trigger captured' for the team_CTF_redflag entity. The script will switch the radar boxes in the back of the truck. (Each radar box in the back of truck has a red transparent and solid version as a visual clue to the players)

## Objective Cameras

The original RtCW used screenshot images to display to the player where they should be going to complete the map objectives. Generally Enemy Territory involves a lot more objectives and constantly maintaining these images during the development of the maps was time consuming and resource-hungry.

The solution was to create a virtual camera system which could be setup to point at objectives or spawn areas throughout the map. As dynamic objectives were used for the escort-a-tank missions, the system was also extended to allow chase cameras as well.

Below is a screenshot from the Limbo menu featuring the various command map icons, filters down the right and the objective description/camera viewpoint along the bottom.

When the player clicks on the spawnpoint icons on the command map the game will update the camera viewpoint at the bottom of the screen.

When the player cycles through the objectives with the up and down arrows the camera will change to reflect what they are looking at.

Critical objectives will make the icon pulsate slowly so players know they are important and they can find them on the command map easier.

The objective camera entity info_limbo_camera has to target another entity; otherwise Enemy Territory will not load the map. This is something to be careful of if you region compile maps. (You can add entities to compiled maps with the "-onlyents" option.)

To create a spawn room camera you will need to setup the info_limbo_camera entity to target the team_WOLF_objective entity. Also remember to give the info_limbo_camera entity a targetname so that it can be removed from the game if need be. Often forward spawn areas are removed in LMS.

For each objective in the map you will need to create an info_limbo_camera entity and target it at an info_null entity. The info_null is used as the direction the camera will be looking. The info_limbo_camera entity must also have an objective key which tells the game what camera is looking at what objective.

The final use for the info_limbo_camera entity is to follow a moving objective like for example the tank in Goldrush or Fueldump. In the map Fueldump the tank (script_mover entity) is made from various clip brushes which all the camera entities target.

At various points on the tanks journey are info_limbo_camera entities which are used to watch the tank. They all target the tank entity regardless of how far away they are from the tank initially.

As the tank (script_mover) goes past each info_limbo_camera entity, the camera viewpoint on the limbo menu focuses on the tank during its journey. The camera will pan around and watch the tank at all times.

Which ever info_limbo_camera entity the tank (script_mover) is closest to, will be used as the current camera location. With some clever placement of the info_limbo_camera entities you can create some really cool angles. In the map 'Goldrush' the tank is shown rolling over the bridge towards the market, but from the angle of someone standing in the courtyard in front of the bank.

## Description Text

All the objective/map text is stored in a separate file called '.objdata' in the 'maps' folder. This was done because some of the objective text needed was special international characters.

The first 3 lines of the file deal with the text for Axis/Allied/Spectator teams. These lines of text are referred to as objective 0 by the info_limbo_camera entities.

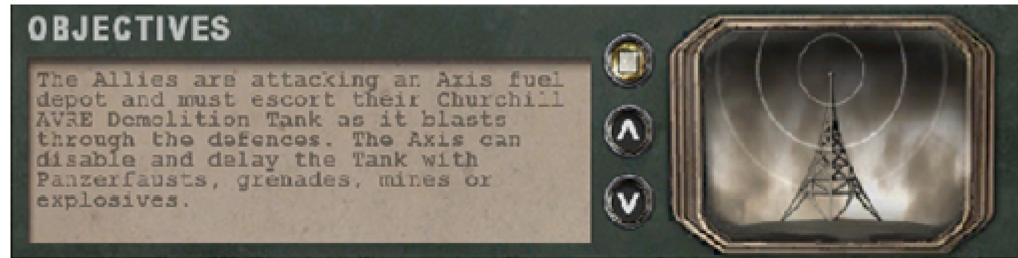
Each team must have an equal amount of text descriptions for each objective in the map. The total number of objectives is still set in the map script file.

Once the objectives are completed or failed they are updated with an alpha channel image stored in 'ui/assets2' called 'stamp_complete.tga' or 'stamp_failed.tga'.

## News Reel

Each map has a WAV file which is used to read a news bite introducing the map as if from an old style newsreel. The wav file is stored in the following area:

/sound/vo//news_.wav




All maps have a sound script file which also defines the news reel WAV file at the top. The scripts files are in the 'sound/scripts' folder and named after the map name.

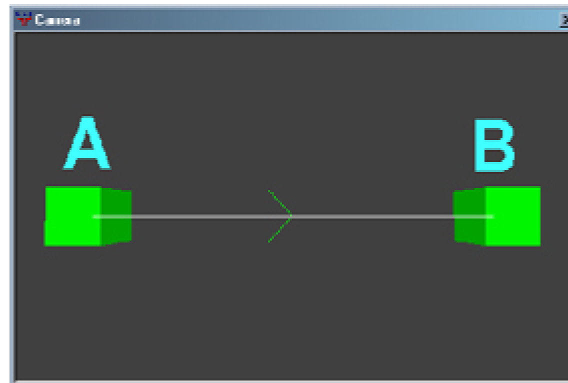
# Spline Path System

The new spline path system allows entities to move smoothly between predefined points on a map. The system is based on Bezier splines which are the same system used by patches. Each map is limited to 512 spline entities. The new entities are stored independently by server and client and don't add to the runtime entity count. The spline path is defined by two new entities called info_train_spline_main and info_train_spline_control . All spline paths are calculated on the 'fly' and will visually look smoother the higher the frame rate.

The spline path system cannot stop objects mid flow between spline points. For greater control of an objects movement use more spline points.

To create a simple path first place two info_train_spline_main entities and target one to the other. This will create a simple linear path from one point to another.

When an entity is moved to point A it will face on a line towards point B. As the entity moves forward on the spline path it will face towards point B.

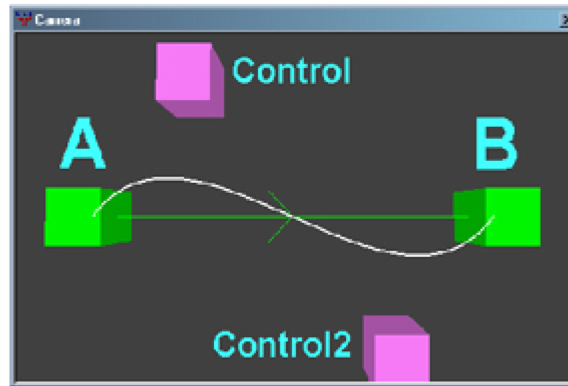



To create a curve between two points add an info_train_spline_control entity. Select the control entity and give it a targetname key with a unique value. Select point A and add a control key with the same value as the control entities targetname key.

When an entity is moved to point A it will face on a line towards the control point. As the entity moves forward it will adjust its face angle based on control and point B.

To have multiple control points for a spline, add each entity as before and create unique targetname keys for each control point. Select point A and add the relevant keys based on how many control points.

The key names must be: control, control2, control3, etc. There is an upper limit of 16 control points, and the more you use, the more CPU it requires, though unless there is excessive use there should not be any noticeable problem.




A new command has been added to BobToolz to enable plotting of the spline paths in SDRadiant, in order to visualize the path your entity will take.

# Moving Object Scripting

Making a moving object objective requires many interconnected entities and scripting, especially if it needs to be damageable.

### Triggers

To allow people to move the object, and see it displayed on the command map, requires a few triggers: a trigger_multiple entity for movement and a trigger_objective_info entity for the command map.

The keys you will need to set on the trigger_multiple are:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
target: This should be the targetname of the target_script_trigger entity used to control the movement, which will be described later.

You may also want to set some of the spawnflags , to allow only one team, or one class, the ability to move the object.

The keys you will need to set on the trigger_objective_info are:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
shortname: The name that will be displayed on the command map.
track: The name that will display when you are near it.
target: This should be the targetname of the func_constructible which will be used to handle damaging/repairing the object, and allow it to show on the command map.

You must set the tank spawnflag for this to show up correctly, and if the object is to be destroyed, the team that owns the object must be chosen from the objective spawnflags.

The trigger_multiple should be a fairly large area around the object, whilst the trigger_objective_info should just fit around the edges, thus only allowing people to construct it when very close to it.

### Building, Destroying and the Object Itself

The object itself is made from a script_mover entity. This is best made out of clip brushes, with a model2 key supplied to use an external md3 model.

The keys you will likely want to use for this are:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
health: The amount of damage the object can take.
model2: The path to an md3 that will be used to display the object.
description: The name that will show up when you hover your cursor over it.
tagent: (as in tag-ent) If using a gun, the targetname of the entity to which it will be attached. (Note: this can be the same entity)
gun: The type of gun. Default is mg42, other value is; browning.

There are various spawnflags which should be set, and some which must be set. You must have the resurrectable flag set and in most cases you will want solid to be set as well. If the object is to be damageable, the appropriate team button should be selected for who owns the object. The compass flag should be set if the object is to show on the compass. If the object is only to be destructible using explosive weapons, the explosivedamageonly flag must be set. Last of all, if you want to attach a gun to the object, the mounted_gun flag must be set.

For damage and repair of the object to work, you must have a func_constructible .

The keys you will need for this entity are:
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.

The start_built flag should be set if the object is to be 'alive' at the beginning of the map. The appropriate team should again be chosen for the team that is to own the object.

## Miscellaneous Entities

You will also need a handful of other point entities which I will now detail.

2 target_script_trigger entities
1 func_timer entity
1 target_kill entity

The target_script_trigger entities require the following keys :
targetname: The name used in the script for referencing this entity.
scriptname:

The routine name in the script file.
target: The name of the trigger event to fire within the routine. (Note: generally this was called 'run')

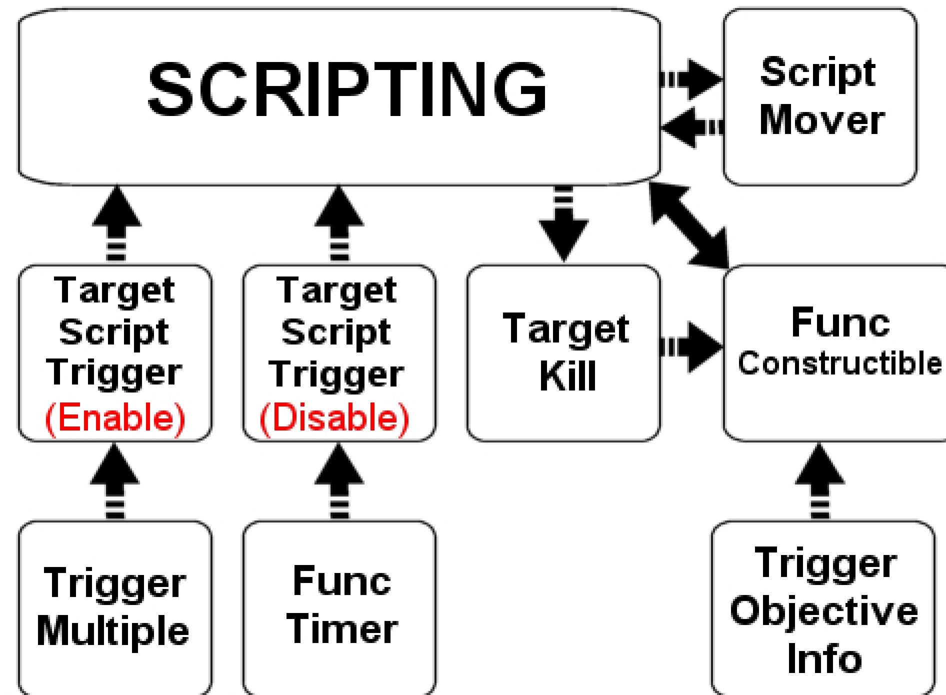
One of these will be known as the disabler, the other is the enabler (This is the one pointed to by the trigger_multiple entity)

The func_timer entity requires the following keys:
target: This should point at the other target_script_trigger entity. (Disabler)
wait: "1"

It should also have the start_on spawnflag set.

The target_kill entity requires the following keys:
targetname: The name used in the script for referencing this entity.
scriptname: The routine name in the script file.
target: This should point at the func_constructible entity.

Scripting




The control of the object via scripting may look complex at first, but it is in fact quite simple. The examples below are taken from the 'Fueldump' script if you wish to take a look over the whole script.

First off, you have the two script trigger routines, which tell the tank whether there is someone in the trigger or not.

tank_disabler

```
{
        trigger run
        {
            trigger tank tank_disable
        }
}

tank_enabler
{
        trigger run
        {
            trigger tank tank_enable
        }
}
```

The 'tank_enabler' routine will call the trigger run function while there are players standing within the trigger, and the 'tank_disabler' will call its trigger run function every time the func_timer activates, i.e. once per second. These two functions just keep the tank object informed about players being in the trigger or not, and it will make decisions based on this information.

Now we'll take a look at what the tank does with this information.

```
trigger tank_enable
{
        trigger self stuck_check
        accum 1 abort_if_bitset 3 // stuck check

        accum 4 set 0 // reset stop counter
        accum 1 bitreset 8 // reset stop check
        accum 1 abort_if_bitset 2 // already following spline
        accum 5 abort_if_not_equal 0// are we not in a script lockout?
        accum 1 abort_if_bitset 7 // death check

        // Any just started moving stuff goes here

        trigger self script_lockout
        trigger tank_sound start

        startanimation 55 10 15 nolerp norandom
        wait 666
        startanimation 5 40 15 nolerp norandom
        wait 500
        trigger self tracks_forward
        trigger self script_lockout_stop
        trigger self move
}
```

First off, it calls the trigger 'stuck_check' function (which will be explained later), which will set bit 3 of accum 1 if the object is stuck. If it is stuck, it aborts, as movement is not possible.

Next, it resets the counter (accum 4) which would stop the vehicle from moving if there was no-one there, and resets the counter that says we shouldn't be moving any more (accum 1 bit 8 ).

Next it aborts if we were already following a path, as you don't want the startup stuff to happen if already moving.

Next, it will abort if a script_lockout (accum 5, more information later) is in place, or if the tank has been damaged (accum 1 bit 7 ).

If it passes all of these checks, then the tank has just started moving, and so it calls all the sound startup and animation startup functions. As it has some waits to have the animation work properly, a script_lockout is put in place so that nothing can interrupt this sequence. After the startup is complete, it then calls the 'trigger move' function, to commence movement along a path.

The disabler routine below is fairly simple.

```
trigger tank_disable
{
	accum 4 inc 1 // up the stop counter
	accum 4 abort_if_less_than 4

	accum 1 bitset 8 // set stop check

	trigger self deathcheck
}
```

Every time this is called (once per second), accum 4 is incremented. If the value of accum 4 is below 4, then the function does nothing more. Accum 4 will be reset by the enabler function as long as the tank is not stuck, and someone is in the trigger. If the tank is stuck, or no-one is in the trigger, then the accum will be allowed to continue incrementing, up to the value of 4 and beyond. Once it reaches this stage, the rest of the function will be allowed to execute, and the stopped moving flag (accum 1 bit 8 ) will be set. This also then calls the 'trigger deathcheck' function, to be sure that something is checking whether the tank has been damaged enough to go into its broken state.

Outlined below are some more of the tank's main functions.

### Movement Loop

```
trigger move_check
{
	trigger self stuck_check
	accum 1 abort_if_bitset 3

	trigger self dispatch
}

trigger move
{
	trigger self move_check
	wait 500
	trigger self move
}

trigger dispatch
{
```

```
	accum 3 trigger_if_equal 0 tank run_0
	...
	accum 3 trigger_if_equal 133 tank run_133
}
```

The 'trigger move' function will be called from the 'tank_enabler' event upon startup, and never again, unless it shuts down, and then starts up again, or is continuing movement from a previous call. The 'trigger move' function calls the 'trigger move_check' function which will see if the tank is stuck, and if not, then call the 'trigger dispatch' function which will in turn select the appropriate 'run_XXX' function to move from its current position. If the 'trigger move_check' function fails because the tank is stuck, then execution will return to the 'trigger move' function. The function will wait 500ms and then call itself again.

### Run Events

```
trigger run_0
{
	accum 1 bitset 2
	trigger tank_turret2 run_0
	followspline 0 spln0 80 wait length -64
	accum 1 bitreset 2

	trigger self run_continue
}
```

Above is one of the tank's 'trigger run' functions. This first sets the moving status to in (accum 1 bit 2 ), then issues the command for the turret to move (as it isn't attached using a tag in this case), and then issue its own move command. After this has completed, it sets the move status to off again, then calls the 'trigger run_continue' function.

```
trigger run_incpos
{
	accum 3 inc 1
}

trigger run_continue
{
	trigger self run_incpos
	trigger self deathcheck
	trigger self stopcheck
	trigger self move
}
```

The 'trigger run_continue' function increases the position counter for the tank, checks it hasn't been damaged, if it hasn't, then checks it hasn't stopped because no one is escorting it, and then if there are escorts, trigger the move function, to continue movement.

### Stuck Checking

```
trigger stuck_check
{
	accum 1 bitreset 3
```

```
	trigger self stuck_check_bridge_built
	trigger self stuck_check_bridge_dyna
	trigger self stuck_check_scriptlockout
	trigger self stuck_check_finished
}

trigger stuck_check_finished
{
	accum 3 abort_if_less_than 134
	accum 1 bitset 3
}

trigger stuck_check_scriptlockout
{
	accum 5 abort_if_equal 0
	accum 1 bitset 3
}
```

These functions, plus several more not outlined here, form a set of functions that check whether the tank can move or not.

The 'trigger stuck_check' function first resets accum 1 bit 3 , which it uses as an indicator to say whether or not the tank is in fact stuck. Since we are resetting it every time, we are assuming that the tank is not stuck unless we are told otherwise, which is the easiest way to handle things. Next, it calls the functions for each individual reason for being stuck, such as the bridge not being built, or it having finished its entire run across the map.

These extra functions simply check to make sure they are valid, usually by checking the position indicator, and any other data they may need to know about. If those checks don't fail, then they set accum 1 bit 3, as an indicator that the tank is in fact stuck. Now, no matter what any of the other events do (assuming they don't reset accum 1 bit 3, which they shouldn't ever do), the flag will remain set when the 'trigger stuck_check' function exits. Other functions can use this to check whether the tank is stuck by simply calling the 'trigger stuck_check' function, then checking the value of accum 1 bit 3 .

### Stop Checking

```
trigger stopcheck_setup
{
	accum 1 bitset 6
	accum 1 abort_if_bitset 8
	trigger self stuck_check
	accum 1 abort_if_bitset 3
	accum 1 bitreset 6
}

trigger stopcheck
{
	trigger self stopcheck_setup
	accum 1 abort_if_not_bitset 6
	trigger self script_lockout

	// Any just stopped moving stuff goes here
	trigger tank_sound stop
```

```
	trigger self tracks_stop
	startanimation 45 10 15 nolerp norandom
	wait 666
	startanimation 0 1 15 nolerp norandom
	wait 900

	trigger self script_lockout_stop
	resetscript
}
```

These two functions, in combination, check whether the tank should stop its movement, as either it's stuck, or no-one is escorting it.

The 'trigger stopcheck_stuck' function does the initial checks to see whether either of those conditions are met, and stores the result as accum 1 bit 6 .

The 'trigger stopcheck' function will then check that value, and act accordingly, either shutting the tank down, or aborting. As before, since the shutdown animation requires some waits, the shutdown enforces a script lockout during that period to prevent anything from interfering.

### Script Lockouts

```
trigger script_lockout
{
	accum 5 inc 1
}

trigger script_lockout_stop
{
	accum 5 inc -1
}
```

The script lockout system acts as a reference counter, that is, it keeps track of how many times it has been told to lock itself using accum 5. This allows various parts of the script to check whether it is locked, by checking that accum 5 is not 0. As this isn't just a flag, it can be used by multiple parts of the script at once, from multiple procedures, as long as they make sure to unlock the tank when they are finished. This was used for the firing sequences and some other things.

### Death and Resurrection

```
death
{
	accum 1 bitset 7
}

trigger deathcheck
{
	accum 1 abort_if_not_bitset 7
	accum 1 abort_if_bitset 9
	accum 1 abort_if_bitset 2
	accum 5 abort_if_not_equal 0
```

```
	accum 1 bitset 9

	changemodel models/mapobjects/tanks_sd/churchhill_broken.md3
	setstate tank_smoke default

	alertentity kill_tank

	trigger self sound_death
	trigger self tracks_stop
	trigger self script_lockout
	trigger self tracks_stop
	startanimation 45 10 15 nolerp norandom
	wait 666
	startanimation 0 1 15 nolerp norandom
	trigger self script_lockout_stop
	resetscript
}
```

The 'trigger death' function simply sets accum 1 bit 7 to on, as a flag for the 'trigger deathcheck' function, so that it knows that the tank's health has reached 0.

The 'trigger deathcheck' function does a few checks at the beginning before changing the model, and playing the shutdown sequence. These checks are: making sure the health has hit 0, the tank hasn't already been made to look broken, the tank isn't moving, and the tank isn't in a script lockout. As before, it initiates a script lockout during the shutdown process.

This also fires the target_kill entity, which is used to damage the func_constructible entity, making the trigger_objective_info entity become active, and so, allowing people to rebuild the constructible, which in turn, will activate the script_mover entity which will call the 'rebirth' function below.

```
rebirth
{
	accum 1 bitreset 9
	accum 1 bitreset 7

	trigger self script_lockout
	changemodel models/mapobjects/tanks_sd/churchhill.md3

	setstate tank_smoke invisible

	trigger tank_sound rebirth
	wait 500

	trigger self script_lockout_stop
}
```

This function simply makes the tank look repaired, setting the appropriate flags, and starts the low engine hum up again.

## The Other Entities

```
tank_trigger
{
	spawn
	{
		wait 100
		attachtotag tank tag_turret
	}
}
```

This is the trigger_multiple entity routine which is used to move the tank. When it spawns, it attaches its origin to a tag positioned at the turret origin on the tank. This allows the trigger to follow the tank around, without having to script its movement independently.

```
tank_build
{
	spawn
	{
		wait 100
		attachtotag tank tag_turret
	}
}
```

This is the trigger_objective_info entity routine, which is used for the repair of the tank. As with the trigger_multiple entity it attaches itself to the tank, for easy movement.

```
tank_construct
{
	spawn
	{
		wait 400
		constructible_class 2
	}

	built final
	{
		alertentity tank
	}

	trigger final_pos
	{
		constructible_constructxpbonus 3
		constructible_destructxpbonus 3
	}
}
```

This is the func_constructible entity associated with the tank. First, it sets its construction class, which is required for the command map marker to show correctly. When the construction finishes, it activates the tank, which will call its 'rebirth' function, and fill its health up again.

When the tank reaches its final position, the 'trigger final_pos' function is called, which changes the amount of XP you get for repairing the tank.

# Foliage Technology

Wolfenstein Enemy Territory adds a new feature to the Quake 3 engine's shader repertoire, the ability to automatically place detail models directly onto map surfaces in an algorithmic fashion. They are designed primarily for foliage, hence the name, but can be used for adding pebbles or rocks scattered randomly around terrain. One key feature of foliage is that it will fade out over distance, but this can cause a problem looking at players at a distance. It is suggested that additional trees and tall bushes are added to the landscape to offer other forms of permanent cover.

When creating foliage, there are 3 things to set up: the foliage model, the foliage model's shader(s), and changes to any existing shader where foliage is desired on.

Foliage is compiled into the BSP hence it is static . If you need to change the foliage in your map, it must be recompiled. This is done for performance reasons -- when foliage is loaded by Enemy Territory it is compiled into a list for fast rendering.

To compile foliage into a map, SDMap2 (Q3Map2) 2.3.32 or higher is required.

## Adding Foliage to Existing Shaders

There is a new shader directive 'q3map_foliage' that specifies how SDMap2 applies foliage to a surface. It takes this form:

q3map_foliage

q3map_foliage models/foliage/grass_5.md3 1.0 16 0.025 0

model: models/foliage/grass_5.md3
scale: 1.0. This is normal size, 0.5 would be half size, 2.0 would be double
density: T16 units. This is the smallest chunk Q3Map will divide a surface up into before it tries to place a foliage instance.
odds: 0.025. This means that a random 2.5% of the potential spots for foliage will be placed. Typically you want to use small values for this; otherwise you will end up with ridiculously high polygon counts.
inverse alpha: 0. this means to use the straight vertex alpha as a scaling factor against the odds of appearing. This is so that terrain shaders with multiple blending layers can have different foliage on each style and have them fade/blend properly together.

If you have brush on terrain0 and grass on terrain1, then the blend shader would have two q3map_foliage directives like this:

q3map_foliage models/foliage/brush.md3 1.0 16 0.025 1
q3map_foliage models/foliage/grass_5.md3 1.0 16 0.025 0

Where the higher-numbered terrain layer/shader uses normal vertex alpha to modulate the odds-of-occurring and the lower-numbered layer uses inverse alpha.

## Creating a Foliage Model

Foliage models should be kept simple and small. Since they're entirely decorative and non-solid, you should avoid making a foliage model that looks as if it could block the player.

For best results, make your models in Max or Maya as a single object (multiple objects will slow down rendering) with a single texture/shader, and try to keep the polygon count as low as possible. Our initial test foliage models were only 6 triangles apiece. When there are a few thousand foliage instances on-screen, the small numbers can add up, so be thrifty.

## Foliage Model Shaders

```
models/foliage/grass_5
{
	surfaceparm trans
```

```
    surfaceparm pointlight

    cull disable

    // distanceCull        distanceCull 256 1024 0.4

    {
        map models/foliage/grass_tan.tga
        alphaFunc GE128
        rgbGen exactVertex
        alphaGen vertex
    }
}
```

# Surfaceparms

The default surface type is stone. For all other surface types you will have to create shaders so that the impact sound/mark is correct for the material. Current surface types are as follows:
stone (default) puff of smoke, heavy boot on concrete surface sound
glass usually on glass func_explosive entities
grass steps brown bullet spray, springy footsteps
gravel steps brown bullet spray, crunchy footsteps
metal steps puff of smoke, clanky footsteps, metallic ricochet
roofs steps clanky footsteps
snow steps white bullet spray, crunchy footsteps
wood steps puff of smoke, hollow steps

### Landmines

A landmine is a small anti-personnel device used to blow up members of the opposing team. In order to allow players to place a landmine, there must be a floor surface with a surfaceparm type of landmine. This is a special surface type which can be used either on its own or with additional surfaceparm types.

As a rule only add the landmine surfaceparm to 'soft' materials like sand, earth etc. In the Enemy Territory game all wood / concrete / stone surfaces were excluded from landmine use.

# Implicit Shaders

Enemy Territory shader language has a set of new directives for simplifying shader scripts: implicit shaders.

Implicit shaders do not have explicit shader stages but rather use pre-programmed settings that work with a variety of lighting conditions (vertex lit, diffuse lit, and lightmapped) to give the shader the best possible representation when it is instantiated in the game.

There are three variants: implicitMap, implicitMask, and implicitBlend. They take the form of a single line with either a path to a texture, or a single '-' character.
implicitMap draws an opaque, single-sided texture
implicitMask draws a masked (alphaFunc GE128), two-sided texture
implicitBlend draws a translucent (blendFunc blend), two-sided texture

For example:

```
textures/wood/plank
{
	surfaceparm woodsteps
	implicitMap -
}
```

This shader will use the default standard mapping, using 'textures/wood/plank.tga' as the texture (shader name tga extension). The only difference internally between it and a defaulted texture shader is the woodsteps. Internally, if the shader is used on a lightmapped surface, it will use a lightmap. If used on a vertex lit surface, it will be vertex lit.

```
textures/wood/plank_nonsolid
{
	qer_editorimage textures/wood/plank.tga
	surfaceparm nonsolid
	implicitMap textures/wood/plank.tga
}
```

This texture differs from the previous one in that it explicitly specifies a texture to use.

# Compiling Maps

### Compiling Maps

Before a map can be played in-game, it must be compiled. This is similar to compiling a C program into an executable, or baking dough into bread. The map editor, SDRadiant, operates on map files, and the game reads bsp files. Generating a bsp from a map is a 3-stage process:

* The BSP phase, where the brushes are used to create a BSP-tree and renderable surfaces are generated
* The Vis phase (-vis) where visibility processing is done
* The Light phase (-light) where the map is illuminated, generating lightmaps and setting vertex colours

While Enemy Territory is based on the Quake 3 engine, and mapping for it is similar to RtCW, there are certain engine/rendering features which require slightly different compile-time options. To compile our levels, we used SDMap2, which is an enhanced version of Q3Map2 with Enemy Territory specific features.

### Key Differences

* Enemy Territory supports up to 1024 vertices per renderable surface and up to 2048 triangles. This is up from 1000 vertices per surface in Quake 3 or RTCW, and up from 64 vertices on lightmapped surfaces. Note this limitation does not apply to patch meshes.
* Foliage surfaces, created with the Enemy Territory foliage shader commands
* Decals, set up with _decal entities projected onto the world
* External lightmaps

### BSP Phase

Enemy Territory maps must be compiled using the -meta switch. This switch creates large triangle meshes from brush faces, allowing more complex scenes than RtCW with an equivalent vertex count. Mappers experienced with Q3Map2 should already be familiar with this switch.

Depending on how your map is constructed, you may wish to use the -mv and -mi switches as well. The -mv switch specifies the maximum number of vertices to support on meta (triangle) surfaces, while the -mi switch specifies the maximum number of indexes. Triangles are composed of 3 indexes apiece, so the -mi value divided by 3 is the max number of triangles per surface. An example usage of these

switches is: -mv 1024 -mi 6144

After the BSP phase is complete, assuming the map hasn't leaked, the following files will have been created:

* .bsp (the compiled map)
* .prt (the portal file used by vis)
* .srf (the surface file used by light)

Note: The prt and srf files do not need to be included with the map when you release it. They are temporary files only used by SDMap2.

### Vis Phase (-vis)

The vis phase is equivalent to Quake 3 and RtCW. It can be run with the -fast switch for cruder visibility processing, but should always be run in 'full' mode (no options) for a final compile. On maps with fog/farplane clipping, put a farplanedist key on the worldspawn entity with a value that corresponds to the distance at which the fog fades out, for example 1024 . This will enable the vis phase to stop visibility processing beyond that distance. This can help make large, open maps more efficient, both in terms of stuff drawn and network overhead.

### Light Phase (-light)

The light phase typically takes the most time. Depending on how the map is lit, how many brushes there are, how much sky, models, and various compile options, the light phase can take anywhere from a few seconds to several hours.

For our maps, we used the following basic switches:
-light -fast -samples 2 -filter -bounce 8 -external -lightmapsize 256 -approx 8

The -fast switch is OK to use for final compiles, as long as you've been designing your lighting around it. The only difference between -fast and a non-fast compile is how SDMap2 handles area (shader) lights. They are typically a bit dimmer at distance with -fast.

The -samples 2 (or 3 or 4, etc) enables adaptive 'smart' antialiasing of shadow edges. It's like the old -extra command, but takes 1/4 the memory and is only about 1.5x the time as a normal compile. This switch is usually for final compiles only.

The -filter switch enables box filtering of lightmap data, similar to how Photoshop's guassian blur works. Some maps look good with -filter and some don't. It's a personal choice.

Radiosity is enabled with the -bounce N switch, where N is the number of iterations you wish SDMap2 to bounce light around. It simulates the diffuse light interaction between surfaces by reflecting light off everything. If you shine a white light on a red wall, for instance, it will reflect red light. The colour reflected is taken from the texture, or the shader's qer_editorimage or q3map_lightimage. The -bounce switch is usually for final compiles only, as it generates thousands of lights with every iteration and can take hours to finish. However, at any point after the initial lighting phase is complete and the radiosity stages have begun, you can cancel compilation and the map will be lit up to the point at which it was cancelled.

Enemy Territory maps were generally much larger than their Q3 or RtCW counterparts, and generated tons of lighting data. To combat this, we used three special switches:-external, -lightmapsize, and -approx.

* -external -lightmapsize 256 instructs SDMap2 to output lightmap data in the form of TGA images in the maps// directory, with a width and height of 256 pixels. Our terrain shaders used 512x512 lightmaps, but generally don't use anything larger.
* -approx 8 sets the vertex-light approximation for lightmapped surfaces. If a surface is too small (less than the size of a lightmap pixel) or has no discernable shadow detail, then its lightmap will be discarded and the surface will be vertex lit. This can be a substantial memory saver on some maps.

### Sample Compile Batch File

To use this batch file, copy & paste it into a text editor and save as 'compile.bat' in some directory. Be sure to edit the paths to fit your Enemy Territory install.

```
@rem to use this batch file, drop a .map file on it or run it from a dos window:
@rem > compile

@set Q3MAP_PATH="C:\SDRadiant\sdmap2.exe"
@set ET_PATH="C:/wolfet"
@set MAP_PATH="C:/wolfet/etmain/maps/%1.map"
@set GEN_OPTIONS=-fs_basepath C:/wolfet -game et

@rem
%Q3MAP_PATH% -meta -mv 1024 -mi 6144 %GEN_OPTIONS% -v %MAP_PATH%

@rem
%Q3MAP_PATH% -vis -saveprt %GEN_OPTIONS% %MAP_PATH%

@rem
%Q3MAP_PATH% -light -fast -samples 2 -filter -patchshadows -external -lightmapsize 256 -approx 8 %GEN_OPTIONS% -v %MAP_PATH%
```

## Converting BSP's to ASE Models

A lot of the geometry in Enemy Territory, such as arches, rocks, pillars and other bits were brushes/patches converted to models. Since rotating brushes tends to screw up texturing, this was a method for our mappers to quickly build geometry in Radiant and place it in maps at odd angles or scales.

To convert a map to an ASE model (3DS Max ASCII Scene Export), it first must be compiled into a BSP. Place the geometry you wish to convert in a caulk box with a single entity (usually a player start). Then compile it with the following options:

-meta -patchmeta -mv 1024 -mi 6144

The -patchmetaoption is important because ASE models don't support patch meshes directly-only triangle/brush surfaces. You can have models in the map as well, they will convert to ASE the same as brush faces. Note: no vis or light phase is necessary.

Then to convert the BSP to an ASE, use the following command:

-convert

SDMap2 will output an ASE file next to the BSP file that can now be loaded in 3DS Max or used as a misc_model entity in another map.

Sample Converter Batch File
To use this batch file, save as 'convert.bat' in some directory. Be sure to edit the paths.

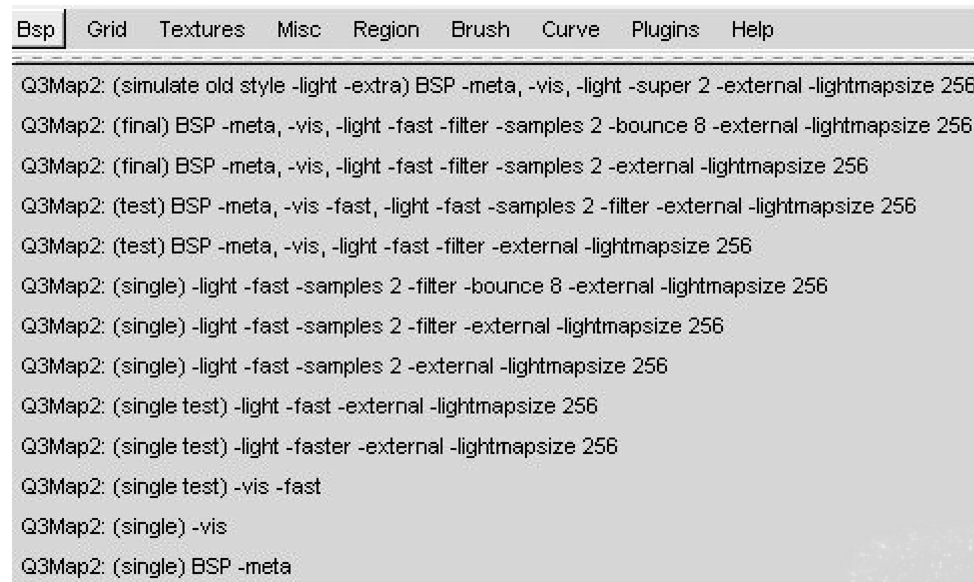
```
@rem to use this batch file, drop a .map file on it or run it from a dos window:
@rem > compile

@set Q3MAP_PATH="C:\SDRadiant\sdmap2.exe"
@set ET_PATH="C:/wolfet"
@set MAP_PATH="C:/wolfet/etmain/maps/%1.bsp"
```

@set GEN_OPTIONS=-fs_basepath C:/wolfet -game et

@rem
%Q3MAP_PATH% -convert %GEN_OPTIONS% -v %MAP_PATH%

GTKRadiant Compile Menu

The map editor (SDRadiant) has been updated with the new parameters for the Enemy Territory compiler SDmap2.




# Console commands

The following console commands are useful when testing a map built for Enemy Territory

/cg_draw2d
This cvar command toggles the display of all the HUD elements except the gun.

/cg_drawfps
This cvar command toggles the display of the current frames per second that the map is running at. This is displayed under the map timer.

/cg_drawgun
This cvar command toggles the display of the weapon model.

/cg_drawreinforcementtime
This cvar command toggles the display of the next respawn time. The counter is displayed next to the map timer. Extremely useful for timing when the next wave of team mates might arrive or when would be a good time to tap out to the limbo menu for a respawn.

Cg_thirdperson 2 will keep the player model in view at all times. This is extremely useful for checking the light values in a map because the player face will reflect the density of the light grid. (You will also need to also change the cg_thirdpersonangle to 180.)

/cg_thirdperson
Cg_thirdperson switches the player's viewpoint between first and third person. This can be useful when checking new player skins/models and the relative scale of the map geometry to the player model. The default value is 0 - first person.

/cg_thirdpersonangle
By default the thirdperson view is of the character from behind. Use cg_thirdpersonangle to alter the angle at which the player model is viewed. The angle is given in degrees where 0 is the default and a value of 180 degrees will give the player a view directly from the front of the player model.

/cg_thirdpersonrange
The default distance that the player model appears from the camera when in thirdperson perspective is 80 units. This value can be altered using /cg_thirdpersonrange. The greater the value the further the camera will appear from the player model.

/con_drawnotify
This command allows developer commands such as r_speeds to display in the main game view and not just in the console. The command is toggled between 0 and 1 where 0 displays the developer commands in console only and 1 displays them in-game also.

/con_notifytime
Ideally set to at least 5 this cvar allows the developer to determine how long (in seconds) the command is displayed onscreen.

/g_scriptdebug
Prints to the server console every line of script that is executed, along with the scriptname of the entity which is calling the script, and the time it was executed at. This is extremely useful for tracking down strange script bugs.

/g_debugconstruct
This will speed up all construction times for quick construction checking.

/r_speeds
The r_speeds cvar displays rendering information which can be helpful in determining what may be having an effect on fps. The default value is 0 which will not display the r_speeds.

r_speeds 1 displays the following information:

xxx/xxx shaders/surfs xx leafs xxxxx verts xxxxx/xxxxx tris x.xx mtex x.xx dc

The renderer batches up drawsurfaces (map drawsurfaces, such as terrain chunks, walls, floors, model sections; model surfaces like players and weapons, and effect surfaces) into a large array before doing render passes. The "shaders" value is actually how many batches there are. Each batch can require one or more rendering pass. For example, weapon shaders often require two: a diffuse/texture pass to add details and an environment pass to make them shiny.

The "surfs" value indicates the number of drawsurfaces visible.

The "leafs" value indicates the number of BSP leafs that are visible in the scene. A leaf is the final, smallest BSP node which contains actual renderable data.

The most important value, "verts" shows the number of vertexes batched up to be rendered in the shader/batches to be calculated, transformed, lit etc. Ideally, the verts count should be as close to 1024 x the shaders (batches) count. Realistically, it will be difficult to achieve this target due to the amount of unique shader effects that only get rendered once.

The number of triangles visible in the scene is indicated by the "tris" value. Ideally the ratio of tris to verts should be as high as possible since you want to maximize the number of tris compared to the number of verts.

### /r_lightmap

Setting this cvar to 1 will turn off textures on light mapped surfaces so that they appear white. This makes it easier to see how shadows affect these surfaces. All drawn surfaces not pastel white are vertex lit and are not affected by the external lightmaps.

R_lightmap 2 will re-colour the map surfaces in light intensity values. This has to be done before the map is loaded. (3 does it in the logical red is hot and has a higher intensity)

### /r_shownormals

Normals indicate the direction a triangle is facing. Use r_shownormals to display normals.

r_shownormals 1: shows you the vertex normals on everything being drawn
r_shownormals 2: shows you the origin, direction and colour of the lighting on entities. This lets you debug the lightgrid.

### /r_showtris

The r_showtris cvar displays the triangles drawn by the game engine. There are three r_showtris levels: 0 - 2. The default value is 0 which does not display triangles.

r_showtris 1 displays depth-buffered triangles, i.e. only those triangles which can actually be seen. This lets the developer see the outline of brushes/models more easily and is especially useful when checking un-VIS'd maps or those with complex geometry.
r_showtris 2 displays all triangles in view which the engine is currently rendering. (This is recognizable as the old Q3 r_showtris 1 cvar).

### /r_triscolor

This cvar sets the colour of the triangles displayed by the r_showtris cvar. The default value is "1.0 1.0 1.0 1.0" which will draw solid white lines around all triangles drawn on screen. The first 3 values are the RBG colour and the final value is the alpha-blend value. This command is useful if you are debugging snow style maps which involve a lot of whiteness.

### /god

In God mode the player will be invulnerable to anything in the map that would normally hurt them, such as damage from a weapon, high falls or remaining under water for long periods of time. Please note that some triggers in the game will still hurt the player regardless of god like status. The trigger_hurt entity has a spawnflag setting to override god player status.

### /map_restart

This command will restart the level and will respawn the player. It will also reload the script file for the map from disk again and reset all construction items in a map. This is useful for tweaking and testing scripts without having to reload the map from scratch each time.

### /nofatigue

When the player is sprinting it will use up stamina. This is indicated by the green bar on the lower right of the HUD. When it reaches the bottom the player will slow down. By setting /nofatigue to 1 will allow the player to continue to sprint even when out of stamina.

### /noclip

Setting noclip to 1 allows the developer to pass through otherwise solid geometry and anything that has been player-clipped.

### /screenshot

Takes a screenshot from the current view and writes it to the 'etmain/screenshots' folder.

### /timelimit

This command sets the length of time (in minutes) that the game will run for before it automatically ends if the offensive objectives haven't been completed. After this time the next level will load. If you

need to test a map over a long period of time then set timelimit to 0.

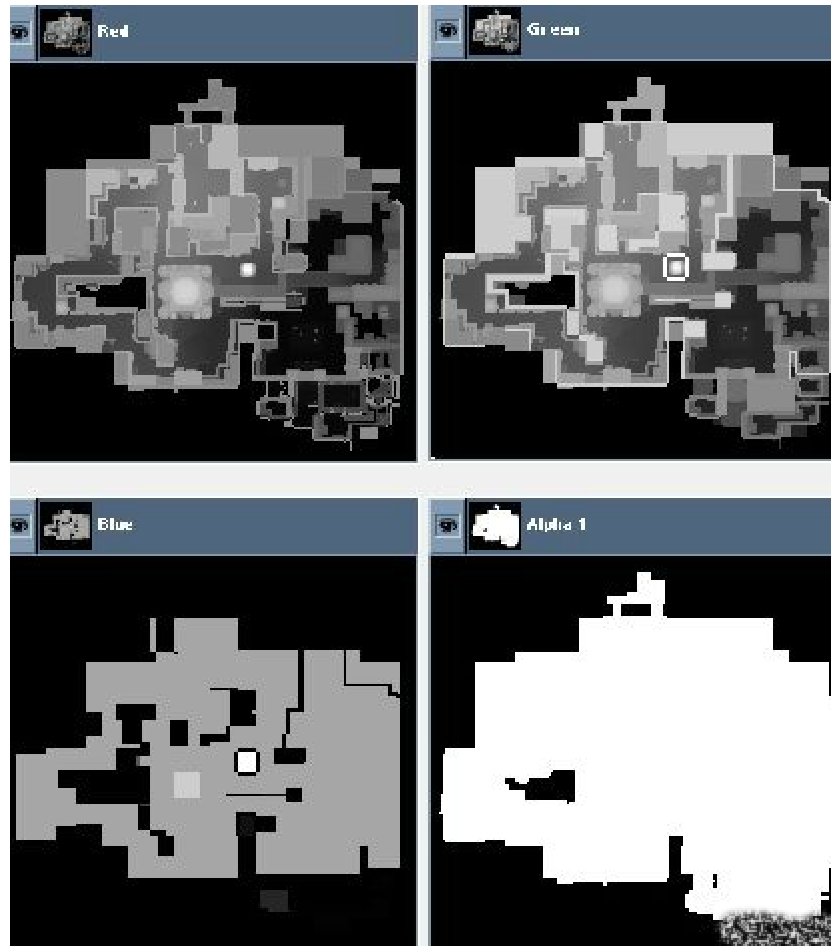
/timescale
Useful when checking large maps and scripted sequences, the timescale cvar will alter the speed at which the game is running. A value of 2, for instance, will run the map at twice the normal speed. (Please check the developer config's for examples of how this command can be useful if bound to several keys.)

## Trace Maps

Precipitation effects and the mortar weapon require a tracemap to work. To generate a tracemap for your map run the game in developer mode (/developer 1) and type the following command '\generateTracemap'. This will generate a tracemap image (.tga) with a 256x256 pixel resolution.

The '\generateTracemap' command stores information in all of the different channels:
Red: Top down mask (height map) of the ground surface of the upper level of the map. (A very good temporary image for the command map as it shows most map details)
Green: Top down ground mask from the perspective of the sky
Blue: Sky mask (The height of the sky brush from the ground)
Alpha: Map brushwork present "yes/no" mask. (Physical map space)

Each channel uses all the values from 0-255 in the palette and some channel images may appear to be blank. If you adjust the gamma of the image then you will see more detail.




If you have any thin roofs in your map, or thin walls outside, due to the resolution of the tracemap some rain or snow might fall through it. To counter this, open the tga and go to the green channel (this is the floor height the code uses). Select the area that's leaking and make it slightly brighter (if you have a pixel with brightness 130, make it 131). If it goes through walls, likely you need to put an extra pixel on the other side of the wall to fake an obstructing object there.

## Arena File

All Enemy Territory maps should/need an arena file defined so that the map will load correctly. The arena file is stored in the 'scripts' folder using the '.arena' format. The arena file deals with information displayed on the loading of the map.

```
{
	map "fueldump"
	longname "Fuel Dump"
	type "wolfmp wolfsw wolflms"
	timelimit 30
	axisRespawnTime 30
	alliedRespawnTime 20
	lmsbriefing "LMS blah."
	briefing "Loading screen blah."
	axiswintext "Final Axis text."
	alliedwintext "Final Allied text."
	mapposition_x 520
	mapposition_y 585
}
```

map: The actual filename of the map to be loaded.
longname: Used in the UI for loading screen purposes.
type: Determines which game types are supported.
"wolfmp" objective/campaign
"wolfsw" stop watch
"wolflms" Last man/medic standing.
timelimit, axisRespawnTime, alliedRespawnTime: These need to be set in the script file for the map as these are UI display only.
mapposition_x, mapposition_y:

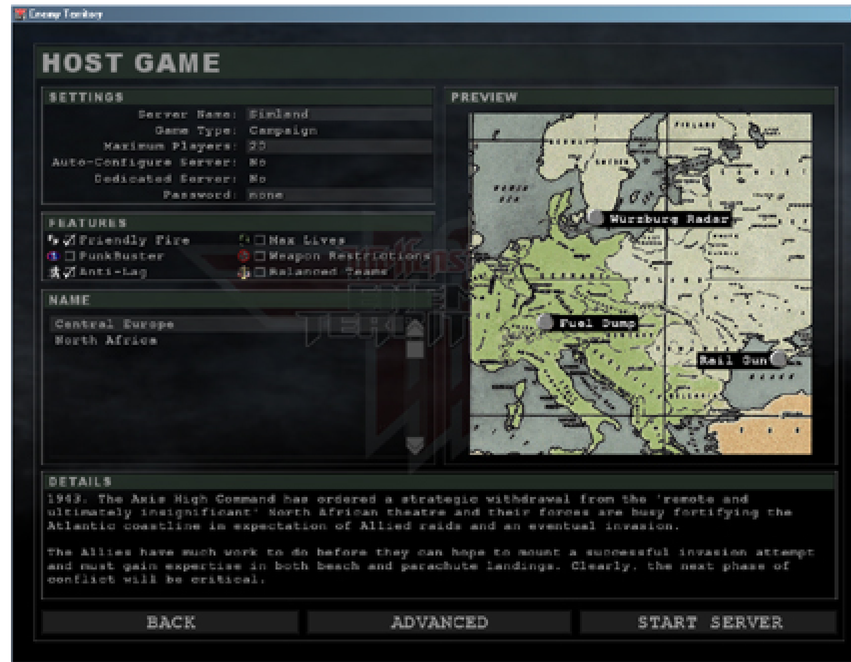
During the loading of the map a picture of Europe is displayed. ('gfx/loading/camp_map.tga') The picture of Europe is 1024x1024 pixels wide/high. Each map is represented on this map with a pin showing where it is located. The map position x/y co-ordinates are a pixel location for where the game will put the pin. The UI puts 0,0 at the top left of the picture of Europe.

The loading screenshot of the map is fixed in one location at the top left of the Europe map. The drawing pin is automatically added to the image by the UI. The map screenshots are located in the 'levelshots' folder. Each screenshot image is 256x256 pixels wide/high with an alpha channel to remove the outside border of the picture.

The screenshot is based on a 4:3 ratio in game screenshot resized to 256x256. The UI automatically resizes the image back to 4:3 ratio again so that is why it must a 4:3 ratio image in the first place. Otherwise the image will look stretched, warped and wrong.

## Campaign File

The campaign file will allow multiple maps to be played as one session using the Experience (XP) system to reward players for completing objectives and playing as a team member. The campaign files are located in the 'scripts' folder. Here is a sample image of the host game UI screen.




The campaign script file contains the following keys:
name: The name to appear in the UI campaign select box.
shortname: Used to start the campaigns from the console (campaign command).
description: Used to describe the campaign on the host game UI.
maps: A list of maps in the campaign separated by semicolons.
mapTC: The map offset on the big map image. The current onscreen map window is only 650x650 pixels. This allows the map pins to be more central located in the onscreen map window.
type: Only one type supported, 'wolfmp'.

/listcampaigns
Will search the 'scripts' folder for all campaign scripts and list their shortnames.

/campaign
This will start a campaign from the console using the shortnames listed above in the /listcampaigns command.

## Appendix A: Scripting Commands

## Message System

wm_teamvoiceannounce

Plays sound to target team.

wm_addteamvoiceannounce
wm_removeteamvoiceannounce

Adds or removes sound to/from the queue of sounds that a player gets to hear when he joins a team.

wm_objective_status

Sets the tick or cross status overlay on the objective panel in the limbo menu for the specific objective.

wm_objective_status

## Trains, splines and animations

abortmove

This will stop any movement currently happening on the entity, this will probably break on spline paths at the moment if you try to stop on a spline path then resume.

attachtotag

Connect this entity onto the tag of another entity.

attatchtotrain

Makes this entity follow another entity with a specified distance (only works in combination with followpath).

faceangles [ACCEL/DECCEL]

The entity will face the given angles, taking to get their. If the GOTOTIME is given instead of a timed duration, the duration calculated from the last gotomarker command will be used instead.

followpath
[wait]followspline

[wait]Direction specifies whether it will travel from start to end (0), or end to start (1). Splinename is the targetname of the spline you wish the train to follow. Speed is how fast the train will travel along the spline. Wait is an optional command that functions the same as on gotomarker, the script will stop execution till this movement has occurred.
Note: Followpath is required when attatchtotrain is used, otherwise use followspline.

gotomarker [accel/deccel] [turntotarget] [wait][relative ]

Note: speed may be modified to round the duration to the next 50ms for smooth transitions.

playanim [looping ] [rate ]

Note: all source animations must be at 20fps.

setposition

Moves entity to position of target entity.

setrotation

Sets the rotational velocity of an entity.

stoprotation

Stops the rotational velocity of an entity.

setspeed [gravity|lowgravity]

Sets the directional velocity of an entity. Use gravity/lowgravity to use non-linear movement.

startanimation [norandom|nolerp|noloop]

Make an entity animate. (Used with misc_gamemodel.)

setposition

Sets the state of an entity.

## Command Post stuff
setchargetimefactor

team: 0 = axis, 1 = allies

sethqstatus

Enables/disables the hq bonus for a team.

## Tank related commands
setdamagable <0|1>

Set damagable status of target entity.

## Construction
construct

Construct stage of target entity.

constructible_class
constructible_chargebarreq

constructible_constructxpbonus
constructible_destructxpbonus
constructible_health
constructible_weaponclass
constructible_duration

See construction classes for details.

repairmg42

Repair target MG42.

setmodelfrombrushmodel [useoriginforpvs]

[nonsolid]Sets model to brushmodel of target func_brushmodel entity.

## Sound Commands

enablespeaker
disablespeaker
togglespeaker

See speaker scripting for details.

playsound

[LOOPING]Currently only allows playing on the VOICE channel, unless you use a sound script. Use the optional LOOPING paramater to attach the sound to the entities looping channel.

stopsound

Stops any looping sounds for this entity.

## Misc. Commands

changemodel

Change the model of this entity.

remapshader
remapshaderflush

Remapshader schedules the replacement of a target shader by a new one. Remapshaderflush executes all scheduled replacement.

(global)accum <

(global)accum inc
(global)accum abort_if_less_than
(global)accum abort_if_greater_than
(global)accum abort_if_not_equal

(global)accum abort_if_equal
(global)accum set
(global)accum random
(global)accum bitset
(global)accum bitreset
(global)accum abort_if_bitset
(global)accum abort_if_not_bitset
(global)accum trigger_if_equal
(global)accum wait_while_equal

cvar

Operation can be any of:
inc
abort_if_less_than
abort_if_greater_than
abort_if_not_equal
abort_if_not_equals
abort_if_equal
abort_if_equals
bitset
bitre set
abort_if_bitset
abort_if_not_bitset
set
random
trigger_if_equal
wait_while_equal

kill

Kill target entity.

remove

Schedule this entity to be freed on the next serverframe.

setglobalfog [float:r] [float:g] [float:b] [float:depthForOpaque]

Changes the global fog in a map.

setautospawn

Set the autospawn of a team to target spawn. The target spawn string is the 'description' key on the team_WOLF_objective.

spawnrubble

Spawn rubble, use in combination with func_debris.

wait
wait random

Wait for a certain duration.

# Appendix B: Entities

worldspawn
script_multiplayer

team_CTF_bluespawn
team_CTF_redspawn
team_WOLF_objective
info_player_deathmatch
info_player_interm ission

info_null
info_notnull
info_notnull_big
misc_beam
info_limbo_camera

light
lightJunior
corona
dlight

misc_model
misc_ gamemodel
misc_mg42
props_chair

path_corner
path_corner_2
info_train_spline_main
info_train_spline_control

trigger_objective_info
func_constructible
func_brushmodel
misc_commandmap_marker
misc_constructiblemarker
misc_cabinet_health
misc_cabinet_supply
trigger_a mmo

trigger_heal

func_bobbing
func_debris
func_door
func_door_rotating
func_explosive
func_group
func_invisible_user
func_pendulu m
func_plat
fun c_rotating
func_static
func_timer

script_mover

target_delay
target_effect
target_explosion
target_fog
target_push
target_rumble
target_smoke
target_speaker

trigger_always
trigger_flagonly_multiple
trigger_hurt
trigger_multiple
trigger_once

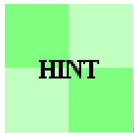
misc_vis_dummy
m isc_vis_dummy_multiple

shooter_grenade
shooter_mortar
shooter_rocket

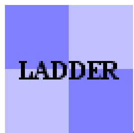
props_skyportal
_decal

# Appendix C: Common Textures

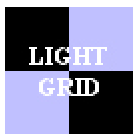
A structural and invisible shader. Used to create the structural 'hull' of the BSP.



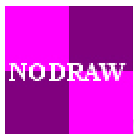
Forces additional portals in the BSP.



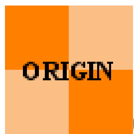
Allows players to climb up or down. Acts like player clip. It must be present on all sides of the brush.



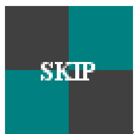
Defines a volume where the lightgrid is active. Any space outside is not calculated.



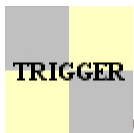
Does not draw in game and is non solid. Used for single sided brushwork like alpha channel textures.



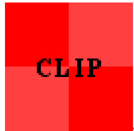
Used by brushwork entities for defining their origin point in the map.



Skip is a nonsolid, invisible, structural shader that does not create BSP splits or portals. Used in conjunction with antiportals or hint faces.



Used on brushwork trigger entities. It's not drawn and is not solid.



Standard Clip brush which will block players.No bullets or projectiles are blocked.

Based on the standard clip brush. Blocks all projectiles.

Based on standard clip brush. Blocks all projectiles and produces a metal sound when hit by bullets.

Based on standard clip brush. Blocks all projectiles and produces a wood sound when hit by bullets.

Terrain metashader with grass step sound.

Terrain metashader with gravel step sound.

Terrain metashader with snow step sound.

Please note that this is not a complete list of texture/shaders from the common directory but a summary of the most useful ones for producing a map for Enemy Territory.

## Glossary

### Entity (point)

An entity refers to a point in a map where a specific object exists. These objects are defined by the entities definition file supplied with the editor tools. Entities can be made of models, specific objects or brushes.

### Entity (Brushwork)

Some entities require that they are made from brushes. Some special brushwork entities require they are painted with texture/shaders from the common directory. For example: A trigger_once entity would be painted with the 'trigger' shader from the common directory.

All brushwork entities should contain 'origin' brushes.

Brushwork entities can also be areas in a map which the player can enter and the game performs a certain event depending on what the entity does.

### Key

All entities have keys which define characteristics of the entity. Examples of these keys are classname, targetname, target , etc.

### Value

All entity keys should have a value by default, whether it is a number or string. These values are usually defined / explained in the entities description box of the editor.

### Spawnflag

A special key defined by a particular set of check boxes in the entity definition dialog box. The selection of the checkboxes usually represents binary values in the final key value.

### Triggering an Entity

To trigger an entity is to activate a certain state / specific function and it varies from entity to entity. With spawnpoint entities they will be turn on or off, while a func_explosion will simple explode and be removed from the game. The only disadvantage to triggering entities is that you cannot set the state of an entity to a specific value with the trigger function.

In order to trigger an entity it must have a targetname key/value so that the script system can refer to it. It's a good idea to make the targetname values more meaningful and not rely on the default 'txxxx' name system of the editor. It can help with debugging of the map / script later on.

### PVS (Potentially Visible Set)

PVS is the set (group) of areas in a map which may or may not be visible from the position at which you are currently standing. Each area is a convex volume, the same as a brush is.

### Scripting - Routine

The script file is made up of routines which define what various entities do and how they react to game events. Each entity in the map has the possibility to run a routine in the script file. To enable an entity to run a routine in the script file give the entity a targetname and scriptname key with relevant values.

### Scripting - Function

Each entity routine is broken down into functions which are triggered by various events which affect the entity. For example: The function 'spawn' is triggered when the entity is spawned into the map at the beginning of the game. Certain functions exist for all entities regardless of what type they are. The standard functions are 'spawn', 'trigger', 'pain', 'death'. Some entities have further functions which are specific to what they do in game. For example: The func_constructible entity has the following unique functions: 'built', 'buildstart', 'decayed', 'destroyed' which are used when certain events happen involving that entity type in game.

### Scripting - Variables

The scripting system supports the ability to keep track of variables or values. The system has local and global variables and each are referenced differently. The scripting system refers the keywords accum and globalaccum to mean variables. There is plenty of ways of testing these variables and making scripting decisions based on their values.

### Local

Each routine can define 8 local variables (accum) which can be used by the routine for keeping track of local conditions. For example, if the entity routine is a button, a local variable could keep track of if the button is on or off. No other routines in the script can access these local variables (accum ).

### Global

Each script can define 10 global variables (globalaccum) which can be used by any routine within the script. For example, the Battery script keeps track of the door generators by use of a global variable. Several routines in the script reference the state of this variable (globalaccum ) to determine what state the generator is in. Built or destroyed.

Quelle:splashdamage.com

*Copyright etpro.de © 2008-2018*