

HOW TO WRITE A PAK MOD FOR MEDAL OF HONOR

by Bits & Crow King

1. Introduction
2. Mods and Modding
3. How MoHAA Works
4. Working with Pak Files Part I
5. Your First Mod
6. Working with Pak Files Part II
7. The TIKI Modeling System
8. TIK Files Part I: Structure
9. TIK Files Part II: Setup
10. TIK Files Part III: Init
11. TIK Files: Init Intermission
12. TIK Files Part IV: Intro to Animations
13. Animations Part I: Point of View
14. Animations Part II: First Person
15. Animations Part III: Third Person
16. Animations Part IV: Non Player
17. Animations Part V: Particle Effects
18. TIK Files Part V: Weapon TIKs
19. TIK Files Part VI: Modding Considerations
20. TIK Files Part VII: Explosions and Effects
21. The State Machine Part I: What is it?
22. The State Machine Part II: Upper Torso
23. TIKI Modeling System: Walkthrough of a Shooting
24. Putting It All Together: Portable Machine Gun Mod Example
25. Writing Shaders
26. Skinning
27. Advanced Animations & TIK Files
28. Putting It All Together: Gore Mod Example
29. And Now for Something Completely Different: User Interface
30. Menus
31. Sound
32. Multiplayer Mods
33. Single Player Mods
34. Integrating Maps & Mods
35. Map Scripts
36. Testing Mods
37. Distributing
38. A Word on Cheating
39. Researching Your Subject
40. References
 - ☐ Tools of the Trade
 - ☐ Resources
 - ☐ A Note on Framerate
 - ☐ Scripting
 - ☐ Client Side Commands
 - ☐ Known Classes and Properties
 - ☐ Glossary

Introduction

We're writing this for two reasons:

1. Because we wish someone else had written it two months ago when we really, really could have used it.
2. Because we hope other people who know more than we do will read it and clear up our misconceptions.

The fact is, there is a ridiculously small amount of information available for modders. Most of what you'll find is the stuff that you can figure out yourself with enough time and patience. But determining an undocumented scripting language's reserved words is not a project likely to yield up success only through investing time in it. We need mentors. We need documentation. We need open communication.

For this reason, we would ask each person who uses this guide to contribute something back to it. The end goal being to have a complete reference for folks who are trying to do neat things with this game. We've come to know a very few things about MoHAA in detail, a few more in general and most not at all. I need your help. Please send e-mails with additions / corrections to CrowKing@autokick.com and we will organize them into this guide.

Our own experience has been to discover that there are quite a few challenges in trying to mod MoHAA. If you are easily frustrated or lack patience, we'd advise taking up some other hobby. Without guidance, you have to have the tenacity to solve many problems through trial and error (even more than is usually required to get computers to do anything). But in the end, when success comes, it is all the sweeter

Now, if you're still reading this, you may be the kind of joe that can take the info here and do something with it. Let's start with clearing the air on our subject.

Mods and Modding

Let's get one thing straight, right off the bat – what the hell is a “mod”? In the loosest sense, a mod is a modification to the game arising out of the user community that has some impact on the player's experience. This could be something as simple as changing the way a gun looks to completely reworking the game's source code into an entirely different game.

But a lot of times you'll hear folks talking about ‘real mods’, as opposed to just changing model skins or weapon damage values. While it is a narrower definition of mod, the line where one ends and the other starts is not clear. However, as visual beings, I think some people's perception of ‘real mods’ begins with eye candy and the scope of things changed. Additionally, ‘real mods’ tend to be those mods that we ourselves don't know how to do. Most people don't think changing a weapon's skin is a “real mod” because they know how it is done, even if they lack the artistic skill. But a mod that completely rewrites the animation engine and adds a lot of new, cool effects will probably be seen as a “real mod” by most people.

At the far end of the mod spectrum is the ‘total conversion’, what I personally would call a “real mod”. This is a special type of mod that uses a game's engine as the basis for creating a whole new game. For example, if MoHAA had come out of the user community for Quake instead of being commercially published, it would be called a total conversion of Quake 3: Team Arena. The creation of a total conversion mod usually requires the actual source code for the original game. However, extending a game like MoHAA by creating a whole series of themed, interrelated single player maps with corresponding on line play, new models, sounds, etc., could be considered a total conversion. The technical differentiation between a “real mod” and a total conversion is not clear (at least I've never seen it made).

As of today (April 7, 2002), we can make “pak mods”, which are mods that make changes to the contents of the pak files that came with the game. In order to make the mods most people are dreaming about, we need two things from EA. The first is a mapping tool. A mapping tool allows you to, as you probably guessed, make new maps. This is what is being referred to as an SDK, or software development kit. It is currently being tested, but has not been released (fingers crossed). (UPDATE – SDK has been released!)

While making new maps is good, what if you wanted to add a new weapon to the game (say, an M1 carbine)? For that, the second thing we would need is the application that converts animation models so MoHAA can understand them. This is a proprietary application that no one can have unless EA releases it. Until then, all the cool new vehicles and guns you see on mod sites can never make it into the game.

While it isn't needed to make an impressive mod, it would be nice to also have a third thing, the source code. There are some things that really cannot be addressed without it, like networking issues, adding more weapon slots, etc. Will it ever see the light of day? Hopefully.

That's the bad news. The good news is that, unlike other games based on Q3:TA, MoHAA shipped with many of the files inside it in text format, so you in essence have the source code for a large portion of the game's logic. If you want to change some minor things around, you can. If you want to make some major changes, the potential is there.

Well, now you have a good idea of what a mod is and the possibilities that exist for modding in MoHAA today. In the next section, I'll go over how the game and its files are organized.

How MoHAA Works

MoHAA's software is adapted from Quake 3: Team Arena (Q3:TA). While the Quake franchise has been in existence for some time, the MoHAA version of it has not and is not yet well documented. Most of the information below is based on Q3:TA and MoHAA seems to not have strayed too far from Q3.

MoHAA has 3 basic components: the **server**, the **client** and the **user interface**. The server handles the chores of managing the virtual world, keeping track of time and objects, what the enemies you can't see are doing, etc. When you play single player, the server runs on your machine along with the other two components. In multiplayer, the server component runs on a remote machine (such as an internet game server). The host's server component is responsible for keeping everyone in synch, deciding who is killed, etc.

The client component manages displaying information about the world for you, the player. It communicates back and forth with the server to let it know what you're up to. The user interface component manages menus and other interface chores.

Communication between the three components is structured through “events”. When you're playing on line and an enemy moves where you can see him, the server component tells your client component about it. When you shoot at him (being the ferocious warrior that you are), the client passes this info back to the server which in turn informs your potentially unlucky victim's client component if he's dead or not (if you're playing on line).

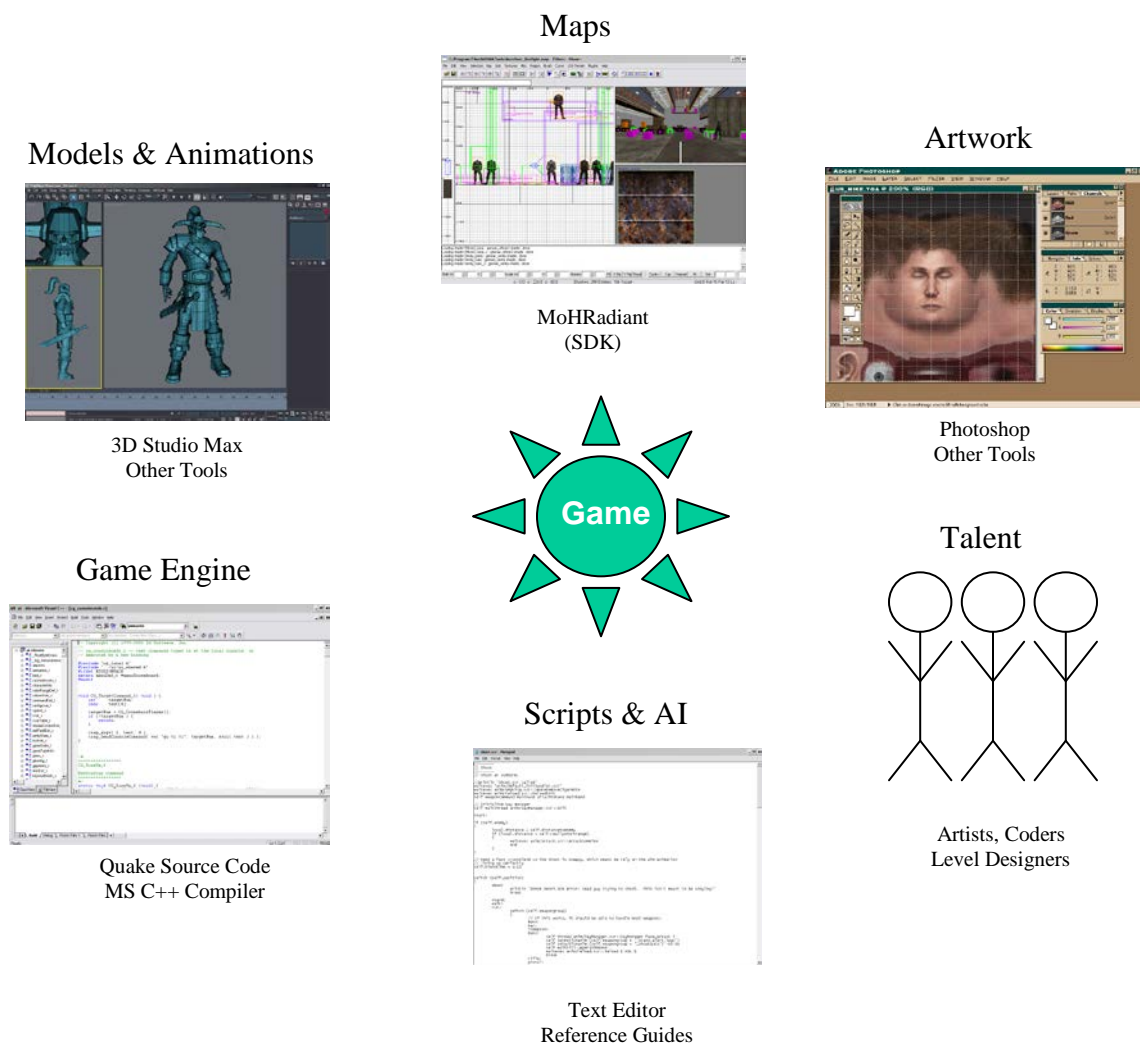
These three components are mostly written in a computer programming language called C and are compiled into binaries. When people talk about getting the source code for MoHAA, they are talking about getting these files in their pre-compiled form. On your machine, these three components are located in EAGAMES\MoHAA and MoHAA\main as executables (.exe) or libraries (.dll). They are surprisingly small – about 6MB - for how much they do. So, what makes up the other 500MB that came with the game? What is in those pak files?

While the 3 game components are powerful, they don't really do anything game like without being told. For example, the user interface component can draw lots of neat stuff on the screen, but it has to know what you want drawn. This information is in the pak files. The game can display models and do things to them, but what models to display and what things are to be done are in the pak files. The animations, the artificial intelligence, the weapons, everything is in there. A pak file is just a compressed (zip) file that contains other files and directories.

Some of the pak file contents are available to you to easily edit, like the AI scripts for single player. They are just text files and, once taken from the pak file, can be modified in Notepad. Modifying them correctly, however, is a whole different story. Other things in the pak files can't be edited directly, but can be edited indirectly.

For example, maps that come with the game are in a format called "bsp", which is a compiled binary (compiled means it has been translated into something the computer can read). While you can't open it directly (at least not in any useful way), you can write a program to make changes to it after it is loaded to run on the server. This is called 'dynamic' editing. Similar things can be done to the models in the game using shaders (more on that later).

The following figure should give you a good overall impression of the components that go into making a Quake based game: Now that you have a general understanding of how MoHAA works, let's look at how the pak files are organized.



Working with PK3 / Pak Files Part I

Before going any further, you need to be aware of a few safety rules to keep you out of trouble (and save you the hassle of reinstalling your game). Ignore these at your own risk:

1. Never (as in NEVER) make changes directly in pak files that originally shipped with the game. Not only is it really bad form, but it will screw up your ability to play. Copy the files to be edited into a directory you've set up to hold your mods and edit them there.
2. Don't give your pk3 files names that follow the same convention as the game's original pak files. For example, don't create and distribute a 'pak7.pk3' file. Use a name that describes your mod and that will be certain to take precedence over the game's pak files (more on this in a bit).
3. Start each new mod from clean copies out of your game's pak files to ensure that they include any patches that you have installed.

Ok, so how the heck do you open up a pak file? If you try to open it with a text editor (like Word or Notepad), it will look like a bunch of funny characters. This is because the pak files are compressed to make them smaller. They are in the same compression format that WinZip uses and that XP can natively read (though I've found XP to be a bit buggy in this regard).

If you're not running XP, you can download freeware or shareware programs that can manipulate compressed files. If you can't find one, you can always buy one from <http://www.pkware.com> or <http://www.winzip.com>. Once you have it installed, go into Windows Explorer (not Internet Explorer) and select "Folder Options" from the Tools menu. Here is where you can associate the "pk3" extension with the WinZip software. From now on, whenever you double click on a pk3 file, WinZip will automatically run to open it. Alternatively, you can just rename the file's "pk3" extension to "zip" whenever you need to open it. But that's extra work and I'm lazy.

If you're running XP, you just need to associate the "pk3" file extension with XP's "Compressed File" format. Go into Windows Explorer, select "Folder Options" from the Tools menu. In the next screen, click the New button. Type pk3 in the File Extension field and click the Advanced button. You should now see an "Associated File Type" drop down box. Scroll down and select "Compressed (zipped) Folder". Click OK.

At this point, you should create a directory (let's call it "MyMods") to hold the files you want to change. It can go anywhere, but you may find it easiest to put it under your MoHAA\main directory. This will simplify moving files back and forth. After you create the directory, copy (don't move, just copy) the pak6.pk3 file into your MyMods directory.

Now, let's take a look inside the copy of pak6 that you just made. Open it from inside Windows Explorer. Inside the file you'll notice that there are multiple subdirectories that organize the files in the pak. Open the 'global' directory by clicking on it. Inside, you'll see a file called DMprecache.scr. This is a script (scr) file that tells MoHAA the names of some other files that it needs to cache in multiplayer. Open the file by clicking on it (you may need to tell Windows to use Notepad when opening scr files) and you'll see the names of the files that will be cached each time in multiplayer.

Three things to note. Take a look at the directory structure for the referenced files, such as the models/weapons/ portion of the first line in the file. This is a convention used throughout the pak files. All the directory references in the pak files have a common origin. So, if you wanted to refer to the DMprecache.scr file in another file, you would write globals/DMprecache.scr.

If you wanted to refer to a file stored in another pak file, you don't need to tell the game what pak file it is in. Just give the directory structure from the root of the pak file. For example, if you

wanted to refer to the bar.tik file found in pak0, all you need to say is models/weapons/bar.tik. The game will figure out what pak file it is in.

These directory structures are important because you'll need to mimick them in your pak modifications. If you wanted to make a mod that included DMprecache.scr, you would need to have a 'global' directory inside your mod's pk3 file and put your version of DMprecache.scr into it.

The second thing to note is that you use the forward slash "/" in your files when referencing directories instead of the backslash "\" convention common in Windows. If you really hate forward slashes, you can also use two backslashes "\\". But that is an extra key stroke and I'm far too lazy for that. The last thing to note is the first two letters in the name of the file, DM. DM (or dm) in MoHAA signifies multiplayer (dm = death match). Whenever you see it, you'll know you are looking at something specific for on line play.

Ok, close everything up without saving and let's make a pak mod.

Your First Mod

For your first mod, we'll make a neat little skin effect for one of the weapons. In Windows Explorer, go to your MyMods directory and delete the copy of pak6.pk3 you put there. The file we want to mod is in the pak0.pk3 file, so copy (make sure you only copy it, you don't want to move it) this file from MoHAA\main to your MyMods directory. If you don't have the disk space, you don't have to copy the pak0.pk3 file, but you must be VERY CAREFUL not to make any changes to the original pak0.pk3 in your main directory.

Now, the file we want to mod is in the "scripts" directory in the pak file, so make a subdirectory in your MyMods directory called scripts. This is where we're going to put the file we change so that when we go to compress it, it will have the right directory structure.

After you've created MyMods\scripts, open your copy of pak0.pk3, go to the scripts directory and copy out the weapons_allied.shader file into your MyMods\scripts directory. You can do this by clicking and holding on the file, pressing and holding the control button (because you only want to copy it, not move it) and drop it on your MyMods\scripts directory. A "shader" file is a little program-like script that tells the game about any special graphic effects you want on a model surface (like making flags wave in the breeze).

Now, close the pak but don't delete it yet. If we mess up, we'll need to go back in and get a clean copy. Open the weapons_allied.shader file (using Notepad) in the MyMods\scripts directory and look at the first few lines. You should see:

```
garand
{
    qer_editorimage textures/models/weapons/m1garand/garand.tga
    {
        map textures/models/weapons/m1garand/garand.tga
        rgbGen lightingSpherical
    }
}
```

Don't worry about what all this means right now. Add a blank line under the "rgbGen lightingSpherical" line, tab over twice and type in "tcmod rotate 5 0 20 //My first mod" without the quotes. The new entry should look like this:

```
garand
{
    qer_editorimage textures/models/weapons/m1garand/garand.tga
    {
        map textures/models/weapons/m1garand/garand.tga
        rgbGen lightingSpherical
        tcmod rotate 5 0 20 //My first mod
    }
}
```

```

map textures/models/weapons/m1garand/garand.tga
rgbGen lightingSpherical
tcmmod rotate 5 0 20 //My first mod
    }
}

```

What does this command do? You'll have to see it. The forward slashes indicate a comment.

Now, save the file and close your text editor. You're done with the changes and now it is time to compress the file. In Windows Explorer, go to your MyMods directory, right click on the scripts subdirectory and do one of the following:

1. If you're using XP, select "Send To" and then "Compressed (zipped) Folder". This will create a file called scripts.zip in your MyMods directory.
2. If you're not using XP, there is probably a zip option somewhere on the menu that is brought up when you right click on a file in Explorer. Select it (consult the program's help file if needed) to create the scripts.zip file in your MyMods directory.

Open the scripts.zip file to make sure everything went in alright. You should see the scripts directory in there and, in that directory, you should find your new version of weapons_allied.shader. Close the zip file and rename scripts.zip to zSkinCrawl.pk3 by right clicking on it and selecting "Rename". Make sure you don't forget the "z" as the first letter in the name. This ensures that your version of the shader file will be used instead of the original (more on this later).

When this is done, copy the zSkinCrawl.pk3 file into your MoHAA\main directory. Fire up MoHAA, join a server and choose the Garand. If you did everything right, you should see some interesting effects. This is a client side mod and no one else on the server can see the effect. Fortunately, the game is designed so you can't do this type of thing to files that affect gameplay on your machine alone. They have to be done on the server so everyone has them.

Congratulations on your first mod! Was it "real" for you? Before moving on, you should remove or delete the zSkinCrawl.pk3 file from your main directory, unless you want to keep the mod active.

Working with PK3 / Pak Files Part II

We did some things in the last section that I didn't explain in detail that you will need to know about when working with pak files.

Naming Files: When MoHAA has multiple copies of the same file in the paks, it uses the name of the pak file to decide which one to use. It does this by always using the file from the pak with the highest name in alphabetical order. For example, in the mod you just created, MoHAA saw your version of weapons_allied.shader in a file called zSkinCrawl and it saw the original shader in a file called pak0. It overrode the original with your new version because zSkinCrawl is alphabetically higher than pak0.

(Advanced Note – if you're making a mod that will have multiple pak files, try to have them all start with the same letters in the first few positions. This way, if another mod on the same machine contains changes to some of the same files your mod changes, MoHAA won't be mixing and matching files from the two mods).

File Types: So far, we've seen two different types of files in the paks that you can edit - scr files and shader files. Here's the complete list and a brief description:

shader	Shaders are short text scripts that tell MoHAA about surfaces and their functions
--------	---

	in the game world. Most of them are in pak0/scripts. (text)
scr	A powerful type of script (program) used to do many things, like controlling the AI in single player to monitoring events in a multiplayer map. They are located throughout the pak files, but a good collection is in pak0/global. (text)
urc	User interface script for making menus. Located in ui directory in pak0 and pak6. (text)
inc	User interface script. The difference between an inc and a urc is not clear. Also located in the ui directory in pak0 and pak6. (text)
skc / skd	Skeletal models and animations used throughout the game world. These files cannot be modified since they are in a proprietary format. They are located in many directories. (binary)
lod	Level of detail that is part of the model / animation. (binary)
wav	A sound file in the popular wav format. Most are located in the sound directory in pak3 and pak4. (binary)
cfg	A configuration file, mostly used to configure the game at startup by setting your key bindings and default settings. (text)
dsp / dsw / ncb	Microsoft VisDev files – do not open or edit. If you open them with MSVS, the files will be changed automatically by MSVS. (text)
ifr	files in the ffx directory (binary)
RitualFont	A font definition file. Do not edit unless you are familiar with the format. (text)
tga	A targa graphic file, which can be edited with most popular graphics tools. They are located throughout the paks, but many can be found under the texture directory in pak1 and pak2. (binary)
txt	A text file. These files are generally used for multiple purposes (ie. they hold scripts or tikis) and just stored in text format. Located throughout the paks. (text)
st	An action state machine. This is a special type of script that controls animations and the states that drive them. Found in pak0/global. (text)
tik	A tik file. Tiks are script like files that describe the properties and animations of objects in the game. Pretty much everything you see in the game besides architecture has an associated tik file, from the King Tiger to the bratwurst. (text)
mus	A music script file. Located in pak0/music. (text)
psp	A Paint Shop Pro graphic. (binary)
jpg	A jpeg graphic. (binary)
dds	A Photoshop compressed texture. (binary)
bsp	The main map file created by the mapping tool. Cannot be directly edited by the mapping tool – it is compiled. Most are located in pak5/maps. (binary)
min	An accompanying map file (seems to define inclusions for the map). Located in pak5/maps. (text)
pth	An accompanying map file (archive?). Located in pak5/maps. (binary)
dcl	An accompanying map file. Located in pak5/maps. (binary)

One thing you'll notice right off is that there are multiple file types that hold "scripts". Scripts are nothing more than small programs and there are a few different types of them in MoHAA. Each has its own keywords and rules.

Some of these file types are well known while some are not. Those that you will need to know in detail to do a pak mod are described in their own sections later on.

File Names: Some file names have special meaning. Here are the few I know of:

fps	First person animations. These are the animations for the player.
anims	Third person animations. These are the animations you see for other players and yourself in third person view.
tps	Third person animations. These are animations for non-players and only apply to the single player game. In my experience, the animations used in these files

	cannot be used in multiplayer.
mike	Player state machine.
dm	Multiplayer file. The contents of the file are specific to multiplayer.
hud	The contents of the file are part of the “heads up display” of the user interface.
allied / a	The contents of the file are for the allies.
german / g	The contents of the file are for the axis.
loading	The contents of the file are for the loading screens for maps.
multiplayer	The contents of the file are specific to multiplayer.
obj	Objective map multiplay file.
clip	
human	
spr	Sprite. Used for particle effects.

General Tips: To save you some sweat, here are a few tips that you may find helpful when working with pak files. While not necessary, they can save time and pain.

1. Unpack all the pk3 files. If you have the disk space, your life will be simpler if you unpack all of the pak files that come with MoHAA into subdirectories of your MyMods directory. This way, you don't have to spend a lot of time dealing with the compression issue. It is easy to run searches and get clean file copies when you can work with unzipped files. Don't forget to refresh them if you update your game with a patch.
2. Have a good file search program. It is essential that you be able to search through the pak files for references. You'll be doing it quite often and you'll need a good search tool. I use EFS (<http://www.sowsoft.com/search.htm>). Just as an aside, the search utility in XP doesn't work.
3. Get XP. While the search tool doesn't work, the native compression in XP is great for working with pak files.
4. Create file associations. You should go into Windows Explorer and create file extension associations for all of the text file types in MoHAA (identified for you in the table above) so you can easily open them.
5. Get a home network. If you have more than one computer, you can do most of your testing yourself if you have a home network. It is essential to test mods from machines other than a non dedicated server.

Okay, I'm tired now. If you're not, go on to the next section. I'm gonna rest here for a minute...if I don't catch up, send a medic.

The TIKI Model System

Ahhhh, much better. So, what the heck is a tik file and where did that name come from? I can answer the first question but I have no clue on the second. The term TIKI refers to the game's modeling system, of which tik files (such as bar.tik) are one part. The other parts are the models and their associated animations. The TIKI modeling system is hierarchical, which means it is made up of multiple pieces that can be used together in different combinations (as opposed to 'monolithic', where all the info for each model and animation is stored together and makes it difficult to re-use). Got that? An example would be nice.

If you were going to add a car in the game, it would be nice to add it in such a way that the game can handle some of the common chores for your car without you having to tell it about every little detail each time it needs to be done. Where the wheels go, how they turn, what color the car is and all that kind of stuff. If you car blows up, what parts should fly off? It's a pain to have to deal with all the details all the time.

The TIKI model system gives you the ability and structure to do it. First, you draw out your car's structure in a wire frame and give names to the parts where you need to attach other parts. For example, you would want to give names to the axles so you can attach wheels. You would want to give a name to the seat so you can attach your ass to it. You would want to give a name to the door handle so you could grab it to open the car. The hierarchical model system understands that these pieces are all interrelated. So, if you stuck cheeseburgers instead of GoodYears on your car, they would still get blown off when the car blows up. You'll need to know these names later on so you can refer to them in your script and tik files.

These names you create are called tags (sometimes referred to as "bones"). So, once you have all your tags, you're ready to go, right? Wrong. How does the game know the top of your car from the bottom, the front from the back? Every model needs a special bone (there's a good joke there, but I ain't saying it) called Origin that handles this. Origin provides a constant reference throughout the animation sequences to tie them all together.

Once you have your car (wire) frame built, you need to put the panels on and paint it. This is called surfacing and skinning. The surfaces define the 2 dimensional connections between your frame's wires and the skins are the "paint" you put on the surfaces. When people release new uniforms or gun skins, they are modifying this portion of the game's TIKI models. But think of the skins as a semi-finish. The full finish for each surface is put on with a shader, a little program that can add special effects to the surface, like making it non-solid (so you can go through it) or invisible. More on shaders later.

Ready to go? Not yet. How does the game know what to do when you open the car door or turn the steering wheel? What should it show you when that happens? You need to define the standard animations for a car. When you open the door, what does it look like in each frame? How many frames will it take to open the door? What the hell is a frame, anyway? You can think of a frame like each little picture in a roll of film (not entirely accurate, but close enough).

This type of information is needed later on when you're synchronizing one model (the car) with another (the guy opening the car door), the sound that is going to be played, the speed of the computer, etc. etc. So, are we ready to get in and drive off? Not quite yet.

How fast can the car go? Is it a stick or an automatic? Can players use it or only the AI guys? How does the game know what animation to play with each specific event? For example, you don't want the car door open animation to play when you turn the ignition key, or the car to blow up when you push the accelerator. All of these details are handled in the tik files. They tell the game about the properties and behaviors of your car and give you the ability to make different kinds of cars without a whole lot of trouble. The "realism" mods for MoHAA modify this component of the TIKI system.

How is all this information organized on your computer? The skeletal models are in files ending with "skd", the animations are in "skc" files (and the "scr" files that control them), the skins are in "tga" and "shader" files and the tik files are in files ending with, you guessed it, "tik". The vast majority of information for the TIKI model system in the game is stored in the models directory (pak0 and some in pak6). There is a standard way that the information is organized that we'll get into later. The largest component of the TIKI model system that isn't stored in the models directory is the skins (mostly under textures).

And that is how it all works, folks. Well, not really. This is a vast, vast oversimplification, but good enough for now because we don't have the ability to make models at this point. But we can get down to brass tacks with the tik files themselves. Which brings us to.....the next section.

TIK Files Part I: Structure

Now that we know just enough to get ourselves into a whole lot of trouble, let's talk tiki, or more specifically, let's talk about tik files. Tik files define for the game the basic behaviors, properties and animations for every object in the world that requires behaviors, properties or animations. That means there'll be a tik file for pretty much everything except the architecture (unless it is going to blow up....then it will need a tik file, too).

All tik files will have the same basic format. Each file must begin with the word "TIKI" on a line by itself and is followed by three sections: Setup, Init and Animations. Setup tells the game about the basic nature of the object and how it should be skinned. Init tells the game about the properties the object will have and the animations section tells it what animations go with what events the object can have (events are defined elsewhere.....yes, more on that later). A tik file can have multiple Setup, Init and Animation sections, but since this is not done very often, I haven't yet figured out the rules with it.

There's one additional, optional piece of information that can go in a tik file, all the way at the bottom. Here's an example from the cockroach.tik file:

```
/*QUAKED fx_cockroach (0.5 .25 0.25) (-12 -12 0) (12 12 80) a cockroach */
```

This information is used by the mapping tool, mohradiant.exe. If you're creating a new class of objects in the game world (and their associated animations, etc.), you may want to have this at the bottom of your tik file so it will show up in the mapping tool. We'll cover this in more detail later.

The basic layout of a tik file would look something like this:

```
TIKI
Setup
    {
        <bunch of setup crap here>
    }
Init
    {
        <bunch of initialization crap here>
    }
Animations
    {
        <bunch of animation crap here>
    }

/*QUAKED <bunch of quaked crap here> */
```

Quick side note: you'll often see the sections in tik files separated with braces ("{" and "}"). This just tells the game that everything within the braces belongs in that section of the tik. If your section is only one line, you don't need them. But if your section is going to contain multiple lines (a compound statement), put them inside the braces.

Another quick side note: If you didn't know what the braces did before I told you, you should probably learn a programming language before trying to tackle modding. But, hey, ya gotta start somewhere. Might as well be with the next section.

TIK Files Part II: Setup

The Setup section is where you tell the game about the surfaces of the object the tik file describes, how big it is in relation to the game world and any special info about it's location. There are up to four important pieces of information:

scale

This is where you tell the game how big the object is in relation to the rest of the game world and the format is "scale x". Ever see mod where the players are giants? Here is where it was done. When objects are placed on maps, they have to be sized in proportion to everything else on the map. MoHAA maps are measured in standard units where 16 units is one foot. The number you put here is what the game needs to multiply your model's size by to get it into 16 units per foot.

So, if 16 units in your model is also one foot, then you would put a 1 in for the scale. However, chances are, you probably didn't build your model (if you could build a model for MoHAA) to the same scale the mapping tool uses. Building maps and building objects requires vastly different scales to be practical. In MoHAA, the models were built to centimeters, not feet. So, to convert them for use in the game, the scale is set to 0.52 (because there are 30.5 centimeters in one foot and $16 / 30.5$ equals 0.52 map units per centimeter of model). So, if you had a five foot long gun, that is 80 units in the map (16×5). In the model, it is 152.5 units (5×30.5). To convert the model's units to the map's units, we multiply the model's units by the scale: $152.5 \times 0.52 = 79.3$. The game will draw the model 79.3 units long - close enough. If you're using one of the model's that shipped with the game, don't change the scale.

path

Use the path keyword to specify where the game can find the files you are referencing below it. This makes life a little simpler because some of the path names can get pretty long. You don't want to have to type the whole path each time you reference a file. If you have a second path line later in the file, it will be used for all references underneath it. However, it seems like you can fully qualify the path at any time and override the path keyword (it remains in effect for all other lines, though). For example, referencing models\weapons\... in a line will not cause the game to think it should be looking for a models\weapons... directory underneath the path you've already defined.

skelmodel

Use this keyword to define the skeletal model for the object being described by the tik file. Note that the file name is relative to the path you defined above. Skeletal model files will end with skd.

surface

Use the surface keyword to tell the game about the object's surfaces and the shaders that will be used on them. The format is "surface <surface name> shader <shader name>". We'll go over shaders later, but the important thing is that you know this is the spot where you draw the link between the shader and the surface it is to shade. If you're not sure what surfaces are on a model, you can open the skd file using a binary editor to see what has been defined. If you don't know what I'm talking about, don't try it...but then again, if you never try it, how will you learn? Also, you can define multiple surfaces with similar names to be shaded by using an asterisk. For example, "surface mg42* shader someshader" tells the game to shade all the surfaces in the model that start with "mg42" with the shader "someshader".

radius

The size of the object's shadow. If I had more info on it, I'd tell ya. It seems to mostly be used for humans.

The first two keywords (scale and skelmodel) are required in the Setup section. However, there may be times where you want to make a tik file, but not include a skeletal model. This is often the case for special effects, like explosions. To do this in MoHAA, you can reference some dummy files set up by the developers. Look in models/fx/dummy or examine some of the explosion tik files in models/animate.

These keywords can also be repeated multiple times, particularly for composite models (ie. a model made up of multiple parts). The TIKI model system includes other keywords that I have not seen used in MoHAA. Since MoHAA has been customized, they may not be supported, but here they are:

origin <x y z>: Adds an offset to the Origin tag in all the animations.

lightoffset <x y z>: Adds an offset to the lighting of a model. Not sure what the effects are.

surface <surface name> damage <damage multiplier>: I believe this is used for armor, to increase or decrease the amount of damage taken when it comes though the named surface.

surface <surface name> flags <flag keywords>: I believe this allows you to state some flags similar to what can be done in a shader. I have not seen it used in MoHAA.

You can also put some script into the setup section (like a case statement), but we'll cover that later. Here are two examples of the Setup section in a TIKI file. The first is from models/vehicles/fockwulf:

```
setup
{
    scale 0.52
    path models/vehicles/fockwulf
    skelmodel fockwulf.skd
    surface fockwulf3 shader fockwulf
    surface fockwindow shader fockwindow
    surface fockwulf1 shader fockwulf
}
```

Simple enough. The second is from models/player/allied_Airborne.tik:

```
setup
{
    path models/human/allied_airborne // Set path to set skelmodel from
    skelmodel airborne.skd // Set body model

    surface shirt shader airborne_top
    surface pants shader airborne_pants
    surface sleeve shader airborne_top_cull

    path models/human/heads
    skelmodel head1.skd
    surface head shader tom

    path models/human/hands
    skelmodel hand.skd
    surface hand shader handsnew

    path models/equipment/USGear/airborne
    skelmodel airborne_gear.skd
    surface gear shader airborne_gear

    path models/equipment/USGear/helmets
    skelmodel us_helmet.skd
}
```

```
        surface us_helmet shader blank_web  
    }
```

This one is a bit more complex (but not the most complex one in the game, by any means). Notice that the model has multiple parts and each must be setup. Now, bravely on to the next section.

TIK Files Part III: Init

The initialization section tells the game what to do when the object your describing in the tik file is first created in the game world. However, if you remember from our discussion about how MoHAA works, there are two places where the object needs to be initialized. One place is on the client, where the player interacts, uses, stomps on or otherwise badly treats your object. The second place is on the server, where the object must be managed, tracked and (eventually) buried or otherwise disposed of.

So the initialization section can have two parts, the server section and the client section. It does not matter what order they come in. If they contain compound statements, these statements should be enclosed in brackets. One thing to keep in mind if you're making mods for multiplayer is that just because you put something in the server or client section does not mean that that is what will actually run on the client or the server. If your mod is only physically installed on the network server, the client may only be taking pieces of your tik file and combining it with pieces of its version of the tik file. Conversely, if your mod is ojnly on the client's machine and not on the network server, it may be using parts of the network server's tik file for the object instead of your mod's tik file. This can cause a lot of grief, crashed machines and mass hysteria. It is critically important you test your mods before releasing them on an unsuspecting (and unforgiving) world. To help you out, I'll share what I've learned so far later on.

So, what kinds of things go in the init section? Here's where you tell the game all about the properties of your object. Going back to the car analogy, you would put the make, model, transmission, engine size and all the other type of info you see on a dealer's sticker in the init section for the server. Because the server controls the world (while the client presents it to you), it is the server that needs this type of info.

How do you know what to put here? Each type of object in the game has a "class" and the items you put in the init section are determined by it. Different classes have different properties. For example, vehicles are one type of class and have a very different set of properties than the weapon class. Cars don't have clips and weapons don't have drag. While you can add new instances of the class, you can only add new classes and the properties that go with them by modifying the game's source code. Since no one but EA has the source code, we won't be adding new classes any time soon. The server has to have pre-existing knowledge of the class for it to make sense of what you put in the init section and it gets this information from a programmer via source code. If it doesn't recognize the class, it doesn't know what to do with the info in the tik file.

For many of the things you will want to do, this won't be a problem as MoHAA gives you a lot of control over each class' properties. For this reason, if you wanted to add flame throwers to the game (which is significantly different than all the other weapons), you probably wouldn't need to create a new class. But you would need to create new models and animations, and that is not possible right now.

If you're really desperate, you can create a tik using a generic animation so people can see the object you want to create and do some generic type things with it. But you can't add those special, cool functions that make it a unique object. In the best case scenario, you may be able to fudge the property settings for an existing class to mimic the new class. Takes time and a lot of

patience. You can't take the properties from one class and use them in the init section of another class.

It may be possible to add your own properties to a tik file that the game does not know about, but that you can access from script files. This way, you could modify some things about pre-existing classes by writing custom scripts, but it would have limited application to the single player game. I haven't tried this so I'm not sure if it is possible or not.

It is not possible for me to say what is required to go into the init section, as it depends on the class you are using. However, I think you should always at least define the object's class. This takes the form of "classname <name>", where <name> is a class that the game already knows about.

It is also not possible for me to tell you what the properties are for every type of class. I have this information for weapons and some of their related classes, but that is it. If someone were to go through the game (or better yet, write a program to do it) and record all the different types of properties for each class, their values and formats, they would have my eternal gratitude. For now, you should just look at the existing tik files for the class you are using to see what they contain.

What about the client section...what goes in there? You would put in this section the type information that is relevant to what the client machine does – showing stuff to the player. Here is where you would tell the game about what graphics and other types of things it needs to cache in memory to use with the object your defining in the tik file.

In my experience, the client section is not always needed. However, the server section is where you tell the game what the class of the object is so you probably always want to include at least that much information in the init section. However, the game does contain tiks without class declarations (see models/ammo). As with the server section, I cannot provide to you an exhaustive listing of items or considerations for what should be in the client section. You should review the tiks for other files of the same class as your tik for pointers.

It is possible to put some types of script commands and functions in the init section of a tik file. If you have anything like that in a client section, it will get executed when the model gets registered on the client. Code in the server section will execute when the object is spawned on the server. For example, each time you reload your BAR, a clip is spawned in your model's hand so the server code will execute each time you reload. But the BAR's clip is only registered on the client one time, so that code would run only once. This is a good thing, because you only want to load graphics for the model one time, but you may want to have different things happening with the clip depending on the circumstances

TIK Files: Init Intermission

Let's take a break from the abstract description of what is in a tik and take a look at one in a bit more detail. Here's an example from the init section of the BAR (models/weapons/bar.tik). I've added comments in black in each section explaining what the heck is going on. We'll get into the nitty gritty detail of what each line does later on.

```
init
{
    server
    {
        // Notice that the different sections are indented in order to make seeing
        // the structure of the init a lot easier. This is a common convention that you
        // should use.
```

```

// The first order of business is to tell the game what the class is. Remember
// that most properties in a tik are specific to the class, and just about all tiks
// will have 'classname' at the top of their init's server section.
classname      Weapon
weapontype      mg
name           "BAR"
rank           410 410

// The server will be telling the client what the sounds are for these
// weapon class specific actions.
pickupsound     bar_snd_pickup
ammopickupsound bar_snd_pickup_ammo
noamosound      bar_snd_noammo

// You'll often see sections of tik's commented out by the developers. Usually,
// the section is some standard part of the Quake model that the MoHAA
// developers decided to do in a different way. In this instance, the normal tik
// holster stuff is commented out and done via animations in MoHAA. Note – just
// because something has been commented out does not mean it won't work.
// Holstering info
//
// holstertag      "Bip01 Spine2"
// holsteroffset   "8.0 -7.75 6.5"
// holsterangles   "0 185 -25"
// holsterScale    1.0

// Another common element of the weapon tiks is that those properties that apply
// to both multiplayer and single player, or just single player alone, come first.
// Primary fire type info
firetype        bullet
ammotype        "mg"
meansofdeath    bullet
bulletcount     1
clipsize        20
startammo       20
ammorequired    1
firedelay       0.12

// Another common convention in the tik files is to group those properties together
// that have some type of relationship. For example, the properties that affect a
// weapon's accuracy are put into one section of the init.
//=====//
//                      WEAPON ACCURACY MODELLING    //
//=====//

// The weapon tiks all have a blurb about the particular weapon's real world
// ballistics. They are usually right out of the army manual.
// Springfield) BAR: Max Eff. Range is 500 yds with a muzzle velocity of 2650 ft/s. (30-06

bulletrange     4000 //the range at which
bullets spread is applied
bulletspread    11 11 45 45 //minpitch minyaw maxpitch maxyaw original 11
11 60 60
// firespreadmult 0.38 0.68 200 0.8 //add falloff cap maxtime
firespreadmult 0.38 0.61 200 0.6 //add falloff cap maxtime
bulletdamage    60

```



```

tracerfrequency      3 //original 0

crosshair            1

movementspeed        0.9

// Often you'll see properties that have specific AI uses grouped together.
// However, this doesn't mean that all the properties that affect AI behavior
// are in this section. ("rank" at the top of the init section is also an AI property).
// AI animation group info
weapongroup          bar

//
airange              short

// The DM (death match, or multiplayer) properties can typically be found at the
// bottom of the weapon tik's init section. Notice the naming convention – 'dm'
// followed by the single player name.
// DM Attributes
dmbulletcount        1
dmstartammo           200
dmammomorequired      1
dmfiredelay           0.12
dmbulletrange         4000
// Note how the devs kept track of their different test settings via comments.
// You'll see this often, particularly in the weapon tiks. Be sure you are modifying
// the active setting (and not one that is commented out),
dmbulletspread        12 12 52 52           //10 10 54 54   //10 10 62 62
//fourth run 10 10 99 99 //third run 8 8 90 90   //second run 12 12 60 60
//original 14 14 75 75

//
dmfirespreadmult      0.38 0.68 200 1.5           //add(%) falloff(%/sec)
cap maxtime //1st run (too accurate, whoa, probably no spread)
//
dmfirespreadmult      0.4 0.4 200 0.8           //add(%) falloff(%/sec) cap
maxtime //2nd run Carl (a bit too accurate, but usable)
//
dmfirespreadmult      0.55 0.45 200 0.9           //add(%) falloff(%/sec)
cap maxtime //3rd run Steve (too inaccurate, back to 2nd) Oct. 25
//
dmfirespreadmult      0.475 0.425 200 0.8 //add(%) falloff(%/sec) cap maxtime
//4th run Steve (adjusted to between 2nd and 3rd runs, slightly too inaccurate)
//
dmfirespreadmult      0.4 0.6 175 0.7           //add(%) falloff(%/sec) cap
maxtime //5th run Steve (need to compensate for the slow rate of fire more)
dmfirespreadmult      0.35 0.65 160 0.7           //add(%) falloff(%/sec) cap
maxtime //6th run Steve

dmbulletdamage        30                       //28   //third run 32
//second run 40 //original 35

dmcrosshair           1

dmmovementspeed        0.89 //orig 0.76

// The last entry in the server section is the caching of the tiks for any other
// objects that will be used by this object. Since we will need to spawn a clip
// in the hand of the player during reload, we have to make sure the tik has
// been cached on the server

```

```

// this is attached to the player during reload
cache models/ammo/bar_clip.tik

// The brace below closes the server section of the init. Don't forget to have a
// matching closing brace for all your open braces.
}
client
{
    // In the client section, we need to tell the game to cache up all the
    // graphics that aren't going to be cached elsewhere. This is usually
    // just the graphics for the muzzle flash for weapon tiks.
    cache tracer.spr
    cache muzsprite.spr
    cache models/ammo/rifleshell.tik
    cache models/fx/muzflash.tik
    cache corona_util.spr
    cache vsssource.spr
}
}

```

This is by no means the complete list of properties for the weapon class (nor is the weapon class the only class that can be a weapon). In a later section, we will go over the different properties from all the various weapon tik files.

Another thing to note is that many of the settings in the init section are tied to cvars and console commands (sometimes undocumented). You can sometimes alter the settings of the object's properties defined in the tik while the game is being played by modifying the appropriate cvar or executing the right command (from the console or in a script).

TIK Files Part IV: Intro to Animations

You're going to accomplish two main things in the Animations section. First, you'll tell the game what animations to play for the different events / states that can occur with the object described by the tik file. Second, you are going to define any spawned particle effects associated with the object (usually muzzle flash). Depending on the class, you may also need to define some special items here (for example, viewkick for weapons). Since animations are so important to the game, we'll cover them in depth first. Particle effects will be covered later on.

To figure out what to do for animations, you need to understand what they are and how they're put together in MoHAA. An animation is an skc file that describes for the game how a model moves from frame to frame. They're like little movies that play whenever models need to move. You should note that animations are associated with models, so if multiple models are used together to do something, you will need animations for each model (and a script to synchronize them together, if needed). Let's look at an example.

When you reload your Thompson, there are multiple animations that run in a sequence. You click your reload button and this tells the game (through the state machine, which we'll cover later) that you want to reload your weapon. First, you rotate the gun and remove your spent clip. Then, you pull out a fresh clip and put it into the gun. Finally, you chamber a round and reposition the gun to fire. These are all animations, some of them for the gun and some of them for your player. Since the clip is never animated itself, only manipulated, it does not require an animation for this sequence.

Let's start with the weapon. If you look in the Thompson's tik file (models/weapons/thompsonsmg.tik), you'll see the following lines in the Animations section

```

reload ThompsonSMG.skc          crossblend 0.1
{
    server
    {
        9 surface Clip +nodraw
        47 surface Clip -nodraw
        last idle
    }
    client
    {
        entry sound thompson_snd_reload
    }
}

```

The most important thing here is that you notice where the animation (ThompsonSMG.skc) is tied to the event (reload) in the first line. The word “reload” is a special word that you will use in all the weapon tik files to handle the reload event. Don’t change it, as it is tied to a lot of other stuff. I have never seen any documentation on ‘crossblend’, but my guess is that it blends the animation into the current view to make it a bit more natural. The number following it is probably a fade time or some other time indicator.

The server section handles pretty much everything except sound. The first command tells the game that the surface named ‘clip’ should not be drawn on the gun model beginning in frame 9 of the animation. This coordinates the player’s hands removing the clip from the gun. It doesn’t actually pull a chunk of the model off, but it would look strange if the hand came up and went back down with a clip in it while the clip in the gun never disappeared. The second line just tells the game to start drawing the full model again, starting in frame 47. The last line tells the gun to go back into the idle state (which runs the animation on the “idle” line in this same file).

The client section simply tells the game to play the sound of the Thompson being reloaded when the animation is entered (“entry”). The sound itself (Thompson_snd_reload) is associated with a wav file in ubersound.scr. Ever start to reload your gun in the game and had to switch weapons real quick? You may have noticed that the sound of the reload continues to play in many circumstances. That sound is kicked off here.

A couple of things to note. First, the removal of the clip from the gun is just an illusion. The gun model always has a clip on it, but the game just stops drawing it while it is supposed to be in the player’s hands. Second, notice how well coordinated the animation is. The developer specifies the frame in the gun’s animation that corresponds to the frame in the player’s animation that removes the clip. The player’s animation that moves the hands around is in another tik file, namely fps_anims_smg.txt.

Whoa! fps_anims_smg.txt? That’s a text file, not a tik file. What’s happening here? Well, the player’s tik file is actually built on the fly from other text files. We’ll cover later how it is done, but the important thing is that you know that the fps_anims_smg.txt file contains the animations for the player’s tik file for submachine guns.

Inside fps_anims_smg.txt, you’ll see the following line for the reload animation:

```

thompson_reload      viewmodel/smg/reload_tommy_stand.skc      crossblend 0.3

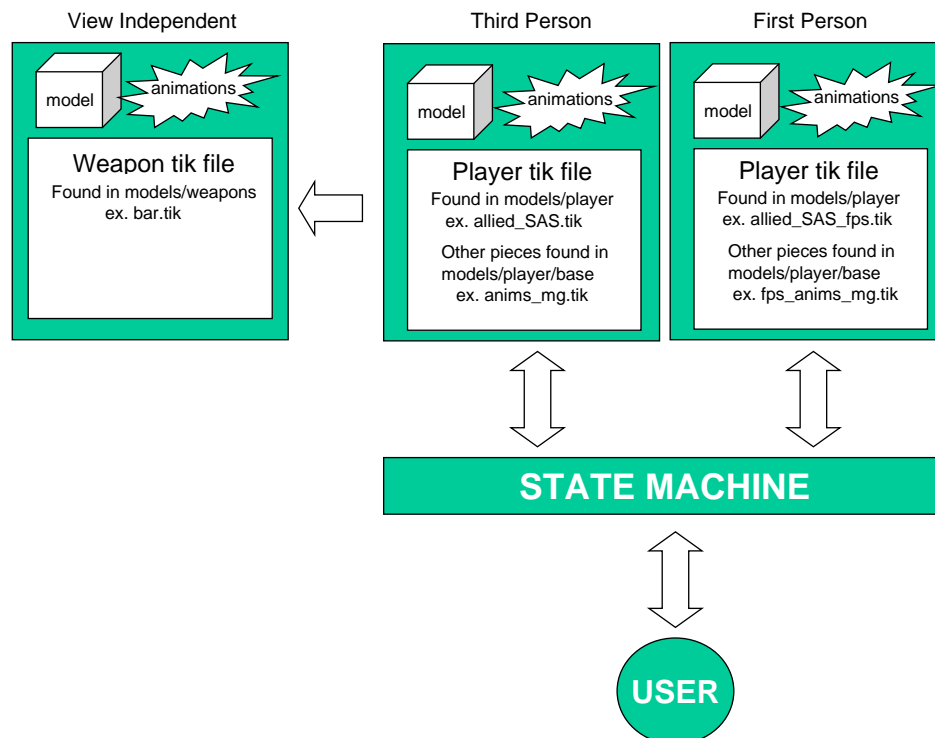
```

This tells the game that, when the player reloads his Thompson, play the reload_tommy_stand.skc animation found in the viewmodel/smg directory (with a crossblend 0.3). As usual, there are a couple of things to note.

First, the event name for the reload sequence (thompson_reload) is different than the event name in the weapon's tik file (reload). The event name is determined inside the state machine and is typically the weapon's name (as defined inside the weapon's tik file) followed by "_reload". You'll see this same convention used for other events (ie. mp40_fire, kar98_idle, etc).

Second, the MP40 and Thompson have different animations, but they're both included in this file (as opposed to the weapon's tiks themselves, which are in separate files). This is a matter of convenience on the part of the developers. They organized the animation sections of the player's tik file by including the axis and allied versions for the weapon type (machine guns, submachine guns, etc.) in the same file. There can be multiple animation sections in the tik file.

The diagram below shows you how it all works together. Note that it is up to the developer to make sure that the animations are in synch between the player and the weapon.



Animations Part I: Point of View

In the last section, we assumed that you were playing in first person view when we went over the Thompson reload. We also only talked about the animations from the player's view point. But if an animation is required for models to be animated, and other players are models, what about their animations? And where do the animations come from when you're playing in third person view?

The skc file animations for other players, and the animations for yourself when you're playing in third person view, are one and the same. As far as the game is concerned, viewing yourself in third person view is exactly the same as viewing other players, except the camera is in a specific spot behind one player (you).

If you looked in some of the directories we talked about in the last section, you may have noticed a few different types of file names in models/player/base. Some of the files start with "fps", and some do not. The files that begin with "fps" are the first person view animation sections for the player's tik file. The ones that do not start with "fps" are the third person view animations for the player's tik file. If you're making a mod that will change the animations, you have to make sure you account for both the first and the third person views.

Note that the animations themselves (the skc files) are different for first and third person views for the same action. It is possible to create different animations for each view for the same action. For example, you could create a situation where the player in first person view sees himself picking his nose while everyone else in the game sees the player picking his butt. Aside from the anatomical comedy, I can't think of any situations where you would want to do this.

The first person view files (those that start with "fps") are pretty straightforward, just calling the animation for each appropriate event. This is because a lot of the details are handled in the weapon's tik file (like playing sounds on the client side). Note, however, that those events which are not covered in the weapon's tik (such as pulling out and putting away of weapons) require the details to be in the player's tik file (which, if you recall, is made from these animation files).

When you look inside the anims text files, you'll notice one big difference right away. The third person view files have a lot more events in them. Why? From the diagram above, you can see that it is the "Third Person" files (ie. anims_mg.txt, etc.) that control the weapon, not the "First Person" files (ie. fps_anims_mg.txt, etc.). Even when you're playing in first person view, the third person view is still running. It is the third person animation file that, in effect, pulls the trigger. The first person files are just concerned with presenting you, the player, with the right view of the world.

You'll also notice that a lot of the animation events in the third person files (ie. anims_XXX.txt) include descriptions about what the player is doing. For example, in the first person file, you just see one event for the weapon firing (ie. bar_fire in the fps_anims_mg.txt file). But in the third person view, there are numerous events for firing. Upon closer inspection, you see that the names for these events include some information about the player's activity (ie. mg_stand_run_fire in anims_mg.txt). Because the game must play different firing animations in third person depending on what the player is doing, you have many more entries in the third person view file. From a first person perspective, the animation is not different if you're walking, running or standing. But in third person, you may need to have a rifle at the hip or the shoulder depending upon your movement.

One final thing to note. Only the player has first person animations defined. You don't need to worry about managing two versions of the tik file's animations for weapons or any other object in the game. They are always in third person view.

Animations Part II: First Person

Let's take a look inside one of the first person view animations for the player's tik file. We're going to look at one of the more complex ones, models/player/base/fps_anims_rifle.txt. I've added comments throughout the file below to explain what is happening:

```
// Since this file will be inserted into a tik file by the game, we need to follow the tik file format
// and include the section heading.
animations
{
    // Cuz we're lazy, set the path so we don't have to keep typing it over and over. All the file
    // references following this line will have the path inserted to the front.
    $path models/human/animation

    // You'll notice that the animations files tend to contain all of the info for the weapon's
    // class. So, every rifle will be in the fps_anims_rifle.txt file. Because this information is
    // being inserted into the player's tik file by the game, the player can have any of the rifles
    // and the game will know what to do. You'll see later that the animation files for all the
    // weapons gets included in the player's tik file every time, so the game is never in doubt
    // when you switch weapons.
```

```

// Animations for the Springfield sniper rifle. Most of these events have corresponding
// events in the weapon's tik file to animate the gun.
//=====
// Springfield Anims
//=====
// The line below tells the game what animation to play when the player isn't doing
// anything. The crossblend and weight parameters are not documented, but probably
// help smooth the transition when the player switches between states (ie. from firing to
// idle, or from idle to reloading).
springfield_idle1      viewmodel/rifle/idle_rifle.skc      crossblend 0.2 weight 1.0
// You will often see commented out versions of other idle states the devs tried.
//springfield_idle2 viewmodel/idle_rifle2.skc crossblend 0.2 weight 0.5 dontrepeate
//springfield_idle3 viewmodel/idle_rifle3.skc crossblend 0.2 weight 0.1 dontrepeate
// The animation to be played when firing the weapon. It is called by the 'viewanim'
// command in the state machine. This animation moves the gun back in the player's
// hands. The animation for the gun itself during firing is in the weapon's tik file.
springfield_fire      viewmodel/rifle/fire_rifle_stand.skc      crossblend 0.05
// The animation to be played when chambering (ie. working the gun's bolt action). This is
// needed between shots to put a new round in the chamber and also during reloads.
springfield_rechamber viewmodel/rifle/vm_springrecham.skc      crossblend 0.1
// The reload animation for weapons that have to load one round at a time are broken
// down into sections. By doing this, the devs are able to break the player out of a partial
// reload in case they want to fire the weapon. For example, if you're caught in the middle
// of reloading your shotgun, you may want to shoot right away.
// This next line is the beginning of the reload animation, where the player throws open
// the gun's bolt.
springfield_reload    viewmodel/rifle/springfield_reload_start.skc      crossblend 0.05
// The next line puts a single round in the rifle.
springfield_reload_single viewmodel/rifle/springfield_reload_fill.skc      crossblend 0
// The next line plays the ending animation for the reloading sequence.
springfield_reload_end viewmodel/rifle/springfield_reload_end.skc      crossblend 0.0
// This next line tells the game what animation to play when the player pulls out the rifle.
// Note that, since the weapon's tik file has no corresponding event, we must make sure
// the sound for the pullout is handled here.
springfield_pullout    viewmodel/rifle/raise_rifle_standplayer.skc
{
    client
    {
        enter sound snd_rifle_pullout
    }
}
// The next line does the same thing, but for the put away event. Again, we must handle
// the sounds here since this event is specific to the player and does not include the
// weapon.
springfield_putaway    viewmodel/rifle/lower_rifle_stand.skc
{
    client
    {
        enter sound snd_rifle_putaway
    }
}

```

Three things to note about the first person animations. First, any event that is specific to the player and not covered in another tik file (such as the putting away of a weapon) must have it's

client side instructions here. So, to play the 'putaway' sound, you need to have a client section under that event in this file.

Second, the first person animation section does not coordinate the firing of the weapon. This comes from the third person animations. It does not matter if you are playing in first person view or not, the third person anim files control the gun.

Finally, the ordering of the events listed is not important. For example, the order of the different reload events does not dictate the order that they will be played in the game. These are just 'event handlers', and each runs when it is fired off by the state machine. It is not like a program where the order of each line dictates the order in which it will be executed.

Animations Part III: Third Person

Now let's take a look inside one of the third person animation files. As was stated in the previous section, it is this file that controls the weapon in the player's hands. Also, this file has many more event entries because the game has multiple animations for each event depending on what the player is doing (running, standing, etc.) To save space, the following excerpt from `models/player/base/anim_springfield_rifle.txt` has been edited to just show those sections relevant to the Springfield sniper rifle:

```
// This file is put into the player's tik file's animation section, so we have to follow the tik format
animations
{
    // Set the path for future file references
    $path models/human/animation

    //=====
    // Reload Springfield (one shell at a time)
    //=====
    // This is the animation that begins the reloading sequence.
    springfield_reload_start viewmodel/rifle/tps_springfield_reload_start.skc crossblend 0.05
    {
        server
        {
            // On the first frame of the reloading sequence, tell the weapon that
            // we're beginning to reload. This will fire off the weapon's reload
            // animation. The weapon itself is intelligent (via the game code) and will
            // keep track of how many rounds are in it.
            first reloadweapon
            // On the second frame (frames start at 0) of the reload animation above,
            // move the weapon from the right hand to the left. The right hand will be
            // used to put the shells into the gun.
            1 weaponcommand mainhand attachtohand offhand
        }
    }

    // This is the animation that is played for each shell that is loaded in the rifle. While it says
    // "loop" in it's name, there is no programming loop in this file. The state machine contains
    // the logic that maintains this loop.
    springfield_reload_loop viewmodel/rifle/tps_springfield_reload_fill.skc crossblend 0
    {
        server
        {
            // On the first frame of this animation, tell the weapon what animation we
            // want it to play.
            first weaponcommand mainhand anim reload_single
            // Put a shell in the right hand of the player. The 'springfield_clip_reload'
```

```

// tik file is for the shell. The 'tag_weapon_right' is the standard model tag
// for the player's right hand.
0 attachmodel models/ammo/springfield_clip_reload.tik
  tag_weapon_right
// On the ninth frame of this animation (frames start at 0), tell the weapon
// to add a round to the gun. Since the gun is intelligent via the game
// code, it keeps track of how many rounds it has.
8 weaponcommand mainhand clip_add 1
// On the same frame, remove the shell from the player's hand.
8 removeattachedmodel tag_weapon_right 0
  models/ammo/springfield_clip_reload.tik
}
}
// This is the animation that completes the reloading sequence by moving the gun back
// into the gun hand.
springfield_reload_end viewmodel/rifle/tps_springfield_reload_end.skc crossblend 0.0
{
  server
  {
    // On the first frame of this animation, tell the weapon to play the
    // reload_end animation.
    first weaponcommand mainhand anim reload_end
    // On the 32nd frame, tell the weapon to move back to the mainhand.
    31 weaponcommand mainhand attachtohand mainhand
  }
}

//=====
// Walking/Cautious
//=====
// These are the animations to be played when the rifle is in the player's hands and the
// player is performing the movement. Since the gun doesn't do anything in this state,
// there is no need to send it any commands.
rifle_stand_walk_fwd      multiplayer/mp_rifle_curious_walk_forward.skc
rifle_stand_walk_back     multiplayer/mp_rifle_curious_walk_backwards.skc
rifle_stand_walk_left     multiplayer/mp_rifle_curious_walk_left.skc
rifle_stand_walk_right    multiplayer/mp_rifle_curious_walk_right.skc

//=====
// Jogging/Relaxed
//=====
// These are the animations to be played when the rifle is in the player's hands and the
// player is performing the movement. Since the gun doesn't do anything in this state,
// there is no need to send it any commands.
rifle_stand_run_fwd      weapon_rifle/walks_runs/rifle_jog.skc      autosteps_run
rifle_stand_run_back     multiplayer/mp_rifle_jog_back.skc         autosteps_run
rifle_stand_run_left     multiplayer/mp_rifle_run_left.skc         autosteps_run
rifle_stand_run_right    multiplayer/mp_rifle_run_right.skc        autosteps_run

//=====
// Standing
//=====
// These are the animations to be played when the rifle is in the player's hands and the
// player is performing the movement. Since the gun doesn't do anything in this state,
// there is no need to send it any commands.

```


rifle_stand_idle multiplayer/mp_rifle_stand_curious.skc crossblend 0.2

//=====

// Crouching

//=====

// These are the animations to be played when the rifle is in the player's hands and the
// player is performing the movement. Since the gun doesn't do anything in this state,
// there is no need to send it any commands.

rifle_crouch_idle weapon_rifle/crouch/rifle_crouch.skc crossblend 0.2
rifle_crouch_walk_fwd multiplayer/mp_rifle_crouch_walk_forwards.skc crossblend 0.1
rifle_crouch_walk_back multiplayer/mp_rifle_crouch_walk_back.skc crossblend 0.1
rifle_crouch_walk_left multiplayer/mp_rifle_crouch_walk_left.skc crossblend 0.1
rifle_crouch_walk_right multiplayer/mp_rifle_crouch_walk_right.skc crossblend 0.1

rifle_crouch_run_fwd multiplayer/mp_rifle_crouchrun.skc crossblend 0.1
autosteps_run
rifle_crouch_run_back multiplayer/mp_rifle_crouch_run_backwards.skc
crossblend 0.1 autosteps_run
rifle_crouch_run_left weapon_rifle/walks_runs/rifle_crouchrun_left.skc crossblend 0.1
autosteps_run
rifle_crouch_run_right weapon_rifle/walks_runs/rifle_crouchrun_right.skc
crossblend 0.1 autosteps_run

//=====

// Rifle Firing

//=====

// This is the animation to be played for each round the weapon fires. This animation is
// usually just of the weapon recoiling back in the hands of the player. The animations of
// the gun itself (ie. the bolt moving, etc.), are in the weapon's tik file.

rifle_stand_fire viewmodel/rifle/fire_rifle_stand.skc crossblend 0.05

```
{
    server
    {
        // As we enter the animation ("entry"), tell the server we are firing the
        // weapon. This does not actually fire the gun, but it tells the server to tell
        // the gun to fire itself. It is the server entry in the weapon's tik file that
        // actually fires the weapon. Commenting out the line below disables the
        // weapon, whereas commenting out the fire line in the weapon's tik file
        // causes the fire animations to play ok, but no bullets come out (and it
        // will make full auto weapons semi auto, because the state machine
        // does not get called by the gun firing).
        entry fire
    }
}
```

// The animation in the next section is not located in the path we previously defined, so
// we redefine the path temporarily.

\$path models/human/protoanimations

// This is the same firing of the gun, but uses a different animation because the player is
// running.

rifle_stand_run_fire fire_rifle_trot.skc crossblend 0.05

```
{
    server
    {
        entry fire
    }
}
```

```

}
// The line below resets the path because only the rifle_stand_run_fire event used an
// animation in a different location.
$path models/human/animation
// This is the same firing of the gun, but uses a different animation because the player is
// walking.
rifle_stand_walk_fire    viewmodel/rifle/walk_player_rifle.skf    crossblend 0.05
{
    server
    {
        entry fire
    }
}
// This is the same firing of the gun, but uses a different animation because the player is
// crouching.
rifle_crouch_fire        viewmodel/rifle/fire_rifle_stand.skf    crossblend 0.05
{
    server
    {
        entry fire
    }
}
// This is the same firing of the gun, but uses a different animation because the player is
// crouching and moving.
rifle_crouch_run_fire    viewmodel/rifle/rifle_fire_crouch.skf    crossblend 0.05
{
    server
    {
        entry fire
    }
}

//=====
// Springfield Rechambering
//=====
// The rechambering animation shows the player working the bolt of the rifle. Notice the
// animation's file name includes "tps" – this stands for third person. Not all animations
// that have "tps" in their name can be used for player animations.
springfield_stand_rechamber    viewmodel/rifle/tps_springfield_rechamber.skf
                                crossblend 0.1
{
    // The weapon commands run on the server side.
    server
    {
        // As we enter the animation ("entry"), tell the weapon to execute what it
        // finds in the "rechamber" section of the weapon's tik animation section.
        entry weaponcommand mainhand anim "rechamber"
        // On the second frame (frames start at 0), tell the weapon to switch
        // hands (from the right hand to the left).
        1 weaponcommand mainhand attachtohand offhand
        // On the 48th frame, tell the weapon to move back.
        47 weaponcommand mainhand attachtohand mainhand
        // At the end of the animation, tell the weapon to move back to the idle
        // state.
        last weaponcommand mainhand idle
    }
}

```

```

}
// Same rechambering, just in a different position.
springfield_crouch_rechamber    viewmodel/rifle/tps_springfield_rechamber.skc
                                crossblend 0.1

{
    server
    {
        entry weaponcommand mainhand anim "rechamber"
        1 weaponcommand mainhand attachtohand offhand
        47 weaponcommand mainhand attachtohand mainhand
        last weaponcommand mainhand idle
    }
}

// Why isn't there separate rechambering events for running and walking, like there was
// when the weapon was fired? The firing animations are broken out because the rifle has
// to be in different positions when being fired in these two positions. Since the position
// does not change for reloading the weapon, the devs did not need to break them out.
// When we cover the state machine, you'll see in more detail how it works.

//=====
// Raise Animations
//=====
// THESE ARE ALL PLACEHOLDER ANIMATIONS!!!!
// The dev comment above appears in all the files. Why? My guess is that the lead
// developer created a generic animation file that included this comment to show the other
// devs where to put the anims. All the different versions were probably created from this
// base file and the comment was not taken out.

// The animations below are executed when the player pulls out the weapon. Note that,
// since this event has no entry in the weapon's tik file, the sound must be included here.
rifle_stand_raise                viewmodel/rifle/raise_rifle_standplayer.skc
{
    // We need to tell the game that we're activating a new weapon, so this goes on
    // the server side.
    server
    {
        // In the first frame of the animation, activate the weapon being pulled out
        // so it is ready to be used. Put the weapon in the 'garandhand'. This is a
        // tag for the weapon hand defined in the model.
        0 activatenewweapon garandhand
    }
    client
    {
        // Start playing the pullout sound upon entering the animation.
        enter sound snd_rifle_pullout
    }
}

// Same sequence, but executed when the player is crouched.
rifle_crouch_raise              viewmodel/rifle/raise_rifle_standplayer.skc
{
    server
    {
        0 activatenewweapon garandhand
    }
    client

```

```

        {
            enter sound snd_rifle_pullout
        }
    }

//=====
// Lower
//=====
// THESE ARE ALL PLACEHOLDER ANIMATIONS!!!!
// These are the animation events for putting the weapon away.
rifle_stand_lower      viewmodel/rifle/lower_rifle_stand.skc
{
    server
    {
        // On the 19th frame of the animation, turn the weapon in the right hand
        // off.
        18 deactivateweapon righthand
    }
    client
    {
        enter sound snd_rifle_putaway
    }
}
rifle_crouch_lower     viewmodel/rifle/lower_rifle_stand.skc
{
    server
    {
        10 deactivateweapon righthand
    }
    client
    {
        enter sound snd_rifle_putaway
    }
}

//=====
// Jumping & Falling
//=====
// These are the animation events for jumping and falling when you're carrying a rifle.
rifle_jump_takeoff     viewmodel/rifle/jump_rifle_start.skc    crossblend 0.1
{
    server
    {
        // Though I haven't tested it, my guess is that this tells the game to test to
        // see if any other entities (ie. the enemy) heard you jumping.
        first ai_event    footstep
    }
    client
    {
        // Unknown – possibly causes a footstep sound to be played.
        first footstep    "Bip01 R Foot" run
    }
}
rifle_jump_run_takeoff viewmodel/rifle/jump_rifle_start.skc    crossblend 0.1
{
    server

```

```

        {
            first ai_event    footstep
        }
    client
    {
        first footstep    "Bip01 R Foot" run
    }
}
rifle_fall    viewmodel/rifle/jump_rifle_loop.skc    crossblend 0.2
rifle_land    viewmodel/rifle/jump_rifle_end.skc    crossblend 0.1
{
    client
    {
        // Unknown – probably plays the sound you hear when you land and
        // triggers some other things (ai check, pain check).
        first landing
    }
}
rifle_land_hard    viewmodel/rifle/jump_rifle_end.skc    crossblend 0.1
{
    client
    {
        first landing
    }
}

//=====
// Pain
//=====
// Pain is a special state in the game. These are the animations to be played based on
// what the player's level of pain is. The state machine makes this decision.
rifle_pain    viewmodel/rifle/jump_rifle_end.skc    crossblend 0.1
$path models/human/protoanimations
rifle_pain_ducked    jump_tuck_loop.skc    crossblend 0.1

//=====
// Misc
//=====

rifle_run_into_wall    jump_land.skc    crossblend 0.1
}

```

Just as with the first person files, the order in which the events are defined in this file is not important. Each is just an event handler that is fired off by the state machine.

Animations Part IV: Non Player

Animations Part V: Particle Effects

The last portion of a tik file's animations section that we have to talk about is the particle effects. What is a particle effect? These are special effects, like dynamic lighting, that you can use to enhance a model's animations. Muzzle flash is probably the most common example of a particle effect in a tik file.

Particle effects are always part of the client segment of the animations because the game code that supports them is in the client portion of the game (cgamex86.dll). To create a particle effect, you tell the game what the particle is (based on a model), how many to create and how they are going to behave. The models used in particle effects are called tempmodels, because they are only in the game world temporarily. Each has a life span associated with it.

Particle effects are different than shaders. A shader tells the game what effects you want to have on a specific surface of a model. That is, it describes and is concerned with model surfaces and will run continuously. A particle effect does not focus on the surface. Instead, it generates a whole new temporary model (usually small in size) and describes how that model will behave.

Before we get into the guts of an example, you should be aware of the following:

- ❑ Offsets and coordinates are almost always in reference to the parent model. Known exceptions are noted in the table.
- ❑ Units of measure are:
 - distance = game units
 - velocity = game units per second
 - acceleration = game units per second squared
 - angles = degrees
 - angular velocity = game units per second
 - time = seconds
 - colors = 0 to 1 (normalized)
- ❑ Coordinates are:
 - positive x = forward (linear) or roll (angular)
 - positive y = left (linear) or pitch (angular)
 - positive z = up (linear) or yaw (angular)
- ❑ Some parameters may be affected by scale and others may not. Unfortunately, there is no rule of thumb to tell. It seems that the choice was made when the command was created based on whether or not it was required for the immediate need.
- ❑ The three functions 'random', 'crandom' and 'range' can be used interchangeably, ie. anywhere you can place a 'random', you can also place a 'crandom' or a 'range'.
 - random x generates a random number between 0 and x.
 - crandom x generates a random number between -x and x
 - range x y generates a random number between x and y.
 - random x is shorthand for range 0 x and crandom x is shorthand for range -x 2x.
 - the ranges for each are inclusive.

Let's take a look at an example – the muzzle flash from the shotgun in models/weapons/shotgun.tik. As usual, the comments precede the line being commented on.

```
// muzzle flash
// On entry to the animation, spawn a dynamic light at the tag_barrel tag on the model. The
// number parameters are red, green, blue, intensity and life. This causes a flash of light when the
// gun fires.
entry tagdlight tag_barrel 0.25 0.2 0.15 140 0.11
// On entry to the animation, spawn the following particle effect at the tag_barrel tag on the model.
// This effect is the small jet of flame coming out the end of the barrel.
entry tagspawn tag_barrel
(
    // The tempmodel will have it's scale set to 0.4 (as opposed to the standard 0.52 set at
    // the top of the tik file).
    scale 0.4
    // Offset the particle effect by 1.5 game units along the x axis (the coordinate is in
    // reference to the parent model, the gun). The other parameters are y and z. This puts
    // the effect a bit in front of the barrel.
```

```

offsetalongaxis 1.5 0 0
// Count tells the game how many particles you want to create.
count 1
// Model tells the game what model you want to use for the particle. Particle models are
// called sprites. Sprites are covered at the bottom of this section of the guide. A model
// can also be defined as a tik file.
model muzsprite.spr
// Set the initial orientation of the particle. Format is 'angles x y z'. You can precede any
// of the values with 'range', 'random' or 'crandom' to randomly vary the value. The x, y
// and z values are relative to the gun model's coordinate system (not the game world).
angles 0 0 crandom 25
// Give the tempmodel a life span of 0.06 seconds.
life 0.06
)

// At entry to the animation, spawn the following particle effect at the tag_barrel tag on the parent
// model (the shotgun). This effect causes a few sparks to fly away from the end of the barrel.
entry tagspawn tag_barrel
(
    // Spawnrate is a command for emitters (explained later). Since the muzzle flash is not an
    // emitter (ie. it is a one time effect), this command should not be here and will not affect
    // the particle. It is probably a legacy line that got left in when the effect was changed
    // during game development.
    spawnrate 1.00
    // The tempmodel to be spawned is defined in a tik file, not a sprite
    model models/fx/bh_metal_fastpiece.tik
    // Create two tempmodels
    count 2
    // Sets the color of the tempmodel. Format is 'color r g b alpha'. The color is modulated
    // with the tempmodel when it is drawn. (Note – the alpha argument may not be a part of
    // the MoHAA version of the 'color' command).
    color 1.00 1.00 1.00
    scale 0.20
    life 0.30
    // Spawn the tempmodels on a circle with a 3 game unit radius. The models will
    // spawn at random locations on the perimeter of the circle.
    radius 3.00
    // Give the tempmodel the indicated speed. This is not a vector and direction will be along
    // the parent model's x axis (ie. forward from the shotgun model).
    velocity 100.00
    // Add a random velocity to the base speed. Format is randvel x y z relative to the model's
    // coordinate system. "range", random' and 'crandom' can precede any of the arguments.
    // If 'range', 'random' or 'crandom' are not specified, the velocity will not be random, but
    // instead be constant.
    randvel 100 0 0
    // Specify the tempmodel's acceleration in the x, y and z axis (in reference to the world,
    // not the gun model). A negative z value is used to imitate gravity.
    accel 0.00 0.00 -120.00
    offsetalongaxis 4 0 0
    // Fade the tempmodel to translucency over its life span. Note there must be an alpha
    // channel defined for this to work.
    fade
    // Align the tempmodel to it's direction of travel. Why? Because the tempmodel 'spark' is
    // shaped like a needle to imitate a spark and it's trail, so this command causes the spark
    // trail to correctly follow the spark.
    align

```

)

// On entry to the animation, spawn the tempmodel at the tag_barrel tag on the model. This effect
// is the larger, outside flame of the muzzle flash.

entry tagspawn tag_barrel

(

// Again, spawnrate is an emitter command and has no effect here. It is probably a legacy
// line.

spawnrate 1.00

model muzsprite.spr

color 1.00 1.00 1.00

scale 0.60

life 0.09

// Animates the tempmodel by varying it's scale by the scalerate specified per second.

// Creates an effect of expansion. In one second, this flame will increase 12 times in size

// (but this tempmodel only exists for 0.09 seconds, so it doesn't get too big).

scalerate 12.00

velocity 33.00

offsetalongaxis 4 0 0

fade

// Causes the tempmodel to rotate randomly.

randomroll

)

// This effect is the flame between the barrel's muzzle flame jet and the larger, outside flame.

entry tagspawn tag_barrel

(

spawnrate 1.00

model muzsprite.spr

color 1.00 1.00 1.00

scale 0.60

life 0.10

scalerate 3.00

velocity 40.00

offsetalongaxis 5 0 0

randomroll

)

// On entry to the animation, execute the command originspawn, but delay it by 0.010 seconds.

// Originspawn spawns the particle effect at the model's origin (a standard tag on all models). This

// effect is the smoke from the shot.

entry commanddelay 0.010 originspawn

(

model vsssource.spr

count 5

// Sets the alpha for the tempmodel. Format is 'alpha value'. Note that there must be an

// alpha channel defined for the skin.

alpha 0.30

color 1.00 1.00 1.00

// spritegridlighting tells the tempmodel to get its vertex colors from the light grid. If the

// shader has "rgbGen vertex", the effect will light properly in dark areas, bright areas

// and colored areas. It will also take dynamic lighting.

spritegridlighting

scale 0.70

life 0.50

scalerate 6.00


```

// 'cone h r' spawns the tempmodel randomly inside a cone h units long with a base r
// units in radius. The cone is along the x axis of the parent model with the tip at the
// origin.
cone 6.00 2.00
velocity 320.00
// radialvelocity sets the tempmodel's velocity away from the origin, like a spoke coming
// out from a wheel. Format is 'radialvelocity scale min max'. The velocity is determined
// by multiplying the tempmodel's offset from the parent model's origin by 'scale' and
// adding a random number between 'min' and 'max'.
radialvelocity 11.00 60.00 120.00
accel 0.00 0.00 -30.00
// 'friction x' causes the tempmodel to lose velocity. It is the percentage of velocity lost
// each second, smoothed evenly over the frames in each second. The percentage is
// calculated as x divided by the physics rate.
friction 5.00
// Sets an offset for the tempmodel from the tag. Format is 'offset x y z', where any of the
// x, y or z values can be preceded by range, random or crandom.
offset crandom -5 crandom -5 crandom -5
offsetalongaxis 30 0 0
fade
randomroll
)

// 0.2 seconds after entry to the animation, spawn the particle effect at the tag_eject tag. This
// effect is the spent shell being ejected.
entry commanddelay 0.2 tagspawn tag_eject
(
    count 1
    model models/ammo/shotgunshell.tik
    // spawnrange defines the maximum distance, in game units, that you can be away from
    // the tempmodel for it to be generated. If you're beyond this distance, the model is not
    // generated as you couldn't see it anyway.
    spawnrange 1024
    scale 1.0
    velocity 70
    randvel crandom 10 crandom 10 random 20
    // Probably a legacy line – applies to emitters (and this effect is not an emitter).
    emitterangles 0 0 0
    // 'avelocity pitch yaw roll' rotates the model at the specified velocity (game units per
    // second) for pitch, yaw and roll. Specify a 0 if you do not want angular velocity for any
    // given parameter. Can be used to make interesting and unpredictable tumbling effects.
    avelocity crandom 90 crandom 90 0
    accel 0 0 -800
    // 'physicsrate fps' specifies how often (in frames per second) physics should be
    // calculated for the tempmodel. Physics checks for collisions and applies friction and
    // acceleration.
    physicsrate 20
    life 2.0
    // Delay the fading of the tempmodel by the time specified. Since the life of the
    // ejected shell is 2 seconds, this will cause it to fade in the final 0.3 seconds.
    fadedelay 1.7
    // Causes the tempmodel to be solid and collide with other solids. Consumes CPU cycles
    // so don't use it unless it will make a difference.
    collision
    // Set the 'bounciness' of the tempmodel. Bounce factors greater than 1 will cause the
    // tempmodel to gain speed by bouncing. Since the model will bounce, 'collision' is

```

```

// implied.
bouncefactor 0.2
// When the tempmodel bounces, play the bounce sound one time. The sound will be
// associated with a wav file in ubersound.scr.
bouncesoundonce snd_shotgun_shell
)

```

As usual, there are a few things to note.

1. The tag names used in the particle effects are standard across weapons. That is, every weapon will have a tag_barrel tag defined on the model. To see all the tag names for a particular model, use a binary editor on the model's skd file. The tags will be listed in text near the top or bottom of the file.
2. There can be more than one particle effect in the tik file. The order that the effects are listed in is not important. They are synchronized by the frames of the parent animation (ie. "entry").
3. The order of the individual commands in each particle effect also does not matter. However, the order of parameters for each command does matter.
4. To create an effect without a model, use one of the dummy models for the tik file.
5. Tik files for particle effects are usually located in models/fx directory.
6. The flame effects do not generate the flash of light of the muzzle flash. This is created by the dynamic light command at the very beginning.

Sprites

To define the model used by the particle system, you need to specify either a sprite or a tik file. A sprite is just a shader with an 'spr' extension stuck on it. For example, if you wanted to render your model with the bigzoom shader, you would use 'model bigzoom.spr'. The muzzle sprite shaders for most of the weapons are in scripts/effects.shader.

Emitters

Particle effects spawned on specific frames of an animation are called "Spawned Particle Effects". The muzzle flash above is an example of a spawned particle effect. Particle effects that are not spawned on animation frames are called "Emitters". Emitters are special in that they run continuously and can be turned on or off in script. A good example of an emitter is the welding sparks seen in the U-Boat level of the game. These effects are not being drawn on any specific frame of an animation, but are 'emitted'. Emitters have some special commands that are used in place of normal particle effect commands, such as 'spawnrate' instead of 'count'.

A reference for all the different commands, parameters, etc., for particle effects is in the reference section of this guide.

TIK Files Part V: Weapon TIKs

If you haven't noticed by now, this guide tends to focus on the weapons class of models. That is because I've spent most of my time working on them. While I can't give you the goods on all the properties for all the classes, I can give you some info on the properties that are specific to the weapon class.

In the reference section of this guide, you'll find a listing of weapon properties and how to use them. While the list is too extensive to go into detail on every one here, there are a few special ones that you may like to have more info on.

weapontype

MoHAA recognizes a fixed number of weapon types – pistol, rifle, smg, mg, grenade and heavy. They are hard coded in the game (ie. you can't modify them) and they correspond to the slots you see in the weapon bar of the HUD. They also correspond to the console command 'useweaponclass' (ie. 'useweaponclass pistol' is bound to the 1 key on your keyboard). The game

chooses which weapon to give to you based on your team (axis or allied) and your weapontype. This is also hard coded, which makes adding new weapons to the game impossible unless you replace an existing weapon (which creates its own problems – covered below). The code that manages weapon selection in multiplayer is in ui/dm_selectprimary.urc. It executes the console command 'dmprimaryweapon <weapontype>' to assign the player's weapon. The one exception to this whole scheme is the shotgun. It has a weapontype of 'heavy', but the argument for dmprimaryselect is 'shotgun'.

name

While it appears that changing the value for 'name' would be harmless, it in fact is not. In the documentation, name is supposed to just control the name shown in the HUD while you're playing. However, it is referenced in the state machine to detect what weapon you have in your hand. If you change the name, you will cause errors in the state machine and animations won't play correctly.

firedelay

Firedelay is the minimum amount of time that must pass between shots for the weapon. In a full auto weapon, this sets the rate of fire. The formula to use to convert rounds per minute to the argument for this property is:

$$\text{value} = 60 / \text{rounds per minute}$$

For example, the Thompson shoots approximately 700 rounds per minute. By the formula above, the firedelay setting should be 'firedelay 0.086', which is equal to $60 / 700$. For semi-auto weapons, if firedelay is longer than the time for the chambering animation, the player will not be able to shoot until the timer runs out. If firedelay is less than the time for the chambering animation, the player cannot start shooting until the animation completes.

bulletdamage

This is the base setting for bullet damage. It is further modified in the game based on shot placement (head, torso or limb) and distance to target. May also be affected by target movement, but I have not verified this. Head shots seem to be at some value greater than base damage, torso shots are at base damage and limb shots are reduced from base damage. Downward adjustment to damage for distance is not known, but it does not appear to be linear (as radiusdamage is).

bulletrange

Bulletrange specifies the distance from the muzzle the bullet travels before accuracy modeling kicks in (as set by the properties listed below). This does not mean that setting a large bulletrange will give you perfect accuracy. It appears (and is stated in the documentation) that it sets the distance for bulletspread being applied, but not the other accuracy properties. My personal testing has shown this to be true. The scale for bulletrange is $4000 = 5$ yards. The best way to think about bulletrange is that it sets the diameter of your bullet group, as if the value for bulletrange specified the length of a cone extending out from your barrel – your shots may hit anywhere inside the area of the cone's end. A value of 100 will cause shots to go anywhere to the player's front or sides.

bulletspread / firespreadmult / viewkick

These three properties affect the accuracy of the weapon being described in the tik file. They interact with one another in very complex ways that, frankly, I have not completely figured out. While the documentation is plentiful (especially for viewkick), it is not very helpful (and sometimes wrong). Bulletspread seems to be the base bullet deflection and is randomized, whereas firespreadmult comes into play in extended firing and movement. Viewkick models the effects of recoil in the weapon. All three of these factors come together to give you the resultant shot placement.

The easiest way to think of it is like this: if your weapon shot a laser instead of bullets, setting all these values to 0 would give you pinpoint accuracy. Changes to viewkick will move your shot placement in a pattern, like an exclamation point for the MP40 versus a reversed candy cane for the Thompson. Bulletsread deflects your bullet from the line drawn by viewkick to add more randomness to the pattern. Firespreadmult modifies this even further, but has a delay before kicking in. The best way to make changes to these variables is to isolate them by setting the others to have no effect, make and test your changes, then test everything together to make sure you're getting the effect you want.

The table in the guide's reference section provides more detail on parameters and settings. Here's a few notes for viewkick to get you started:

- ☐ Pitch is the vertical plane, yaw is the horizontal plane
- ☐ Negative pitch is up, negative yaw is left
- ☐ If min/max pitch is set to 0/0 in a V pattern, there will be no yaw displacement. In a T pattern, you will get displacement.
- ☐ If min/max yaw is set to 0/0, you will get yaw displacement after a few rounds due to the pattern shape. If absolute yaw is zero, then all yaw will be eliminated.
- ☐ Recovery rate (deg/sec) is adjusted in 0.1 intervals. Cannot be set to 0 - defaults to 0.1.
- ☐ Setting the min/max pair equal to each other causes the same amount of movement for each shot.
- ☐ When no pattern was specified, the MP40 defaulted to V (ie. would not allow yaw without pitch).
- ☐ For each shot, a number is generated between the min and max values. This gets applied to the last point of impact (ie. the muzzle movement is cumulative). The total amount of movement from the original point of aim cannot exceed the absolute values. However, it would appear that there is some recovery between shots since a min/max pair of 2 does not cause the muzzle to move 30 (ie. 30 shots x avg. move per shot of 1). The shot pattern also alters the movement (V or T).

Aside from the tik file properties, here are some other considerations for weapons:

projectile tiks: If your weapon will shoot a projectile, you will need to have a separate tik file modeling the projectile. Examples can be found in models/projectiles. For example, if you wanted to model a sniper rifle with the bullet affected by gravity, you could have it shoot projectile like bullets instead of regular game bullets. (I have this mod in rough form if you're interested in it).

explosions: Explosions from projectiles (and just about everything else) are modeled as particle effects using tik files and are cached in the projectile's tik file. Look at models/fx/bazookaexplosion.tik for an example.

grenades: The tik files for grenades are broken into a few different pieces and put together by the game at run time with the '\$include' command. You'll find basic weapon tik files in models/weapons and they're broken out by single player (m2frag_grenade_sp.tik) and multiplayer (m2frag_grenade.tik). The file models/weapons/m2frag_grenade_base.txt contains the majority of the multiplayer tik file. There are also projectile files for grenades (since they are thrown). These files are in models/projectiles and are broken out by primary and secondary attack (the difference between SP and MP being the velocity of the projectile). The game code knows which tik to use based on the type of attack.

alternate attacks: While it is possible to add secondary attacks to weapons that don't currently have them, you cannot simply add in the secondary attack animations from the single player game. You need to use an animation that was created for the player.

file names: If you are adding a new weapon to the game, you will have to replace one of the existing weapons. This is because the code that associates the tik file to the weapon selected by

the player is hard coded to the weapontype. For example, if the player selects a submachine gun, the game code will automatically associate mp40.tik (if axis) or thompson.tik (if allied) with the player's primary weapon. You can't create a new weapon and put it in a file called 'greasegun.tik' and expect the game to find it.

multiplayer: One thing to be aware of if you are creating mods for multiplayer is that most server admins only want mods that don't require installation on the client side. That is, they want mods that only need to be installed on the server. An important consideration here is that the animation section of tik files is not taken from the server, but from the client. So, if you're mod changes something in the animation section of a tik file, it won't work unless it is installed on both the client and the server machines. Try to keep all your changes in the setup and init sections of tik files.

TIK Files Part VI: Modding Considerations

This section is sort of a 'catch all' for the things you need to know, but that don't really fit anywhere else.

TIK Keywords

In your adventures through the tik files, you may have noticed a couple of keywords that are used quite a bit that we haven't covered yet. Namely, these are \$define and \$include.

\$define <macro_name> <macro>

If you're familiar with macros in C, you'll know what this is. In essence, you create an alias with the "define" keyword that you can use in place of some other text. For example, the following line could be used in a tik file:

\$define lazytext IamFarTooLazyToTypeThisOverAndOverAgain.skc

From this point on in the file, you don't have to type in the long name of the skc file. You can just type in \$lazytext\$ instead, and the game will automatically replace it with the skc's file name. Note that you have to precede and follow your macro name with a dollar sign when you reference it.

\$include <file_name>

This keyword will insert the contents of another file at the point in the tik file where it occurs. For example, you could include the following line at the end of the setup section of a tik file:

\$include MyInitFile.txt

This would insert the contents of the file "MyInitFile.txt" at the end of the setup section. This keyword is useful when you have a section of a tik file (or some other file) that is shared among many files. Rather than maintain multiple copies of the code inside each tik file, you can keep one copy in a separate file and just include it in all the tiks that need it. This makes changes really easy since you don't have to remember every file that includes the shared code. This is how the game builds the player's tik file.

Map Considerations

In a later section, we'll get into the nitty gritty of combining pak mods and maps. But if you need to integrate your changes with new maps now, here's a few things to get you started.

QuakEd

You've probably seen this at the bottom of the tik files and wondered what the heck it is. The QUAKED section (Quake Editor...get it?) contains the information that the mapping tool (MoHRadiant.exe, the Quake Editor) uses to include your tik objects so you can place them in maps. Its format is

```
/*QUAKED <menu_location> <color> <bound_box_min> <bound_box_max> <description> */
```

If you've used MoHRadiant before, you'll probably understand the following explanations. The **menu location** tells MoHRadiant where to put your object in the pull down menu. So, when you right click on a brush, you can select the object. To start a new branch in the menu, insert an underscore ('_') in the location. For example

```
/*QUAKED ammo_box_mg (0 0 0) (0 0 0) (0 0 0) power. */
```

will cause you to see 'ammo' in the top level, 'box' underneath it and 'mg' underneath 'box'. When you first load MoHRadiant, you'll notice quite a few 'scan files' scrolling by on the bottom of the screen. One of the things it is doing is scanning the tiks for the QuakEd statements and building your menu.

The **color** parameter tells MoHRadiant what color to make the brush once you've assigned this tik to it. The format of the command is (r g b), where r, g and b are red, green and blue values between 0 and 1. You should try and follow MoHRadiant's color convention (ie. similar objects have similar colors). I have no idea what that convention is, but you can just look in a tik file for an object that is similar to the object you're building.

The **bounding box** parameter tells MoHRadiant what the bounding box for the object should be. A bounding box sets the boundaries for the object in the game world to detect things like collisions. It is defined by two sets of three numbers in (x y z) format that define the offsets (in game units) from the model's origin. The first set of 3 numbers provides the minimum values for the bounding box and the second set is the maximum.

Pretty simple, huh? Another way to think of it is that each set of (x y z) coordinates is an opposite corner of a rectangle around the model's origin. Anything that crosses the boundary of this 'box' makes contact with the model in the game.

Note that the values you provide for the minimum should always be less than or equal to the values for the maximum (ie. x-min <= x-max). Setting all the values to 0's will default the bounding box to how it was defined by the model's creator. The bounding box can also be modified inside the tik file (see the bazooka projectile's tik as an example).

The final parameter is just a text **description**.

Precache

Depending on how you went about adding your tik stuff to a map, you may need to add them to the map's precache. Just check out any of the map scripts that came with the original game to see how this is done. Unless you really need to, you should not modify DMprecache.scr to add your tik files. Each map should have it's own precache script that will call DMprecache. You should add your tiks to the map's precache script.

File Organization

When you add new tiks to the game, you should follow the game's conventions for placing them in the file structure. Most tiks describe models, which go in the models directory. Inside this directory, the models are divided up into categories and you should place your tik in the directory where it fits. These directories have specific structures. At their top level, they hold the tik files themselves. Under this directory, you'll see directories with the same names as the tik files. These directories hold the models that are specific to the tik file.

For example, if you created a new Halftrack vehicle, you would place the tik (probably called halftrack.tik) in the models/vehicles directory. Underneath this directory, you would create another directory called 'halftrack' and place all your model files (skc, skd, etc.) in this directory.

TIK Files Part VII: Explosions and Effects

[tbd]

The State Machine Part I: What is it?

All throughout this guide, you've been hearing about the "state machine" and how it affects animations. Well, what the heck is a state machine, anyway? On the surface, it is one of the more bizarre files....very cryptic. However, once you dig into it, you'll see how clever it is.

When you play MoHAA, the game engine listens to your input devices (mouse, keyboard, etc.). When you push the left mouse button to fire, how does the game engine know what animation to play? This can't be scripted because you, the player, can do just about anything at anytime. It can be hard coded in the game engine, but this takes away some of the expandability of the game and puts restrictions on developers – they may have any number of animations they want to play.

So in MoHAA, the game engine manages things by setting your 'state', ie. it determines what you want to do and tells a special game script called the 'state machine' so it can play the right animation. This state machine runs for every frame of the game.

For example, when you press the left mouse button, the game engine tells the state machine that you are trying to shoot. The state machine looks at what weapon you have in your hands (if any), and runs the right animation. As we saw in the animations section of the tik files, a lot more than just playing an animation can occur at this point. Commands in the animation section of the tik file may tell the gun to fire, reload, activate or any other number of things.

There are two distinct state machines in the game for the player. While both fire off animations that can control the entire player's body, one focuses on the legs and the other on the torso. The torso state machine takes precedence over the leg state machine and can override animations being run by the leg state machine. While the two state machines are similar in structure, there are some important and subtle differences we'll cover later.

Both state machines have the following form at the highest level:

```
state <state_name_1>
{
    (state definition stuff)
}

state <state_name_2>
{
    (state definition stuff)
}
etc.
```

When the game first starts up, the game engine creates the player in a beginning state and all further states are transitioned to based on events (ie. what the player does or what something else does to the player). Each of the "state <state_name_x>" sections defines the animations to be played for the state, as well as the criteria for transitioning to the next state. For the legs state machine, each of the state names in the example above will have the following format:

```
state <state_name>
{
    entrycommands
    {
        <command1> "<arg1>" "<arg2>" .....
        <command2> "<arg1>" "<arg2>" .....
```

```

.....
}
legs
{
    <animation1> : <condition(s)>
    <animation2> : <condition(s)>
    ....
    <animationx> : default
}
states
{
    <state_name_1> : <condition(s)>
    <state_name_2> : <condition(s)>
    ....
}
exitcommands
{
    <command1> "<arg1>" "<arg2>" .....
    <command2> "<arg1>" "<arg2>" .....
    .....
}
}

```

In the above example, **state** signifies a state definition and tells the game that you are defining a state that the player can enter and exit. By convention, states should always be in upper case. You cannot create new states – they have to be defined in the game code.

The first section in the state definition is **entrycommands**. These are commands that get executed when the player first enters the state. That means they get executed once, not every time the state machine runs (ie. every frame). I am not sure if they will be executed if the player transitions from a state back to the same state, but I would assume so. In the leg state machine, the entrycommands are often used to set flags that can be referenced elsewhere to tell what the player is up to (ie. walking, crouching, etc.). Another good example of the use of entrycommands is the deactivation of active weapons when you mount a vehicle. I have not compiled a list of all the entrycommands and their functions....any volunteers?

The second section of the state definition, **legs**, tells the game what animation to play when in this state. The animations are defined in the animation sections of tik files (you remember those, right?). So, if you had "mp40_run_fire" defined in your tik file, here is where you would reference it to be played. It is very important to not have any typos here. The game does not behave well when it can't find the animation you are referencing. Also, I believe animation names have to be unique since you are not specifying for the state machine what tik file holds the animation to be played.

Here's how the legs section works. When you first enter a state, the state machine evaluates the conditions on the right side of the colon. The first animation to have all its conditions evaluate to true is the one that gets run. So, the order that you list the animations is important – if more than one would evaluate to true, the one listed higher will be the one to run.

How do you know what you can use to test conditions? I've listed in the references section all the functions that I know of that you can use to test conditions. Please note that some of these may not actually be used anywhere in the game – I've pulled them directly from the game executables. They may just be leftovers from Quake or FAKK2. Also, you can't add new functions. They are defined directly in the game code.

Here's a quick example:


```

mp40_stand_walk_back: IS_WEAPON_ACTIVE "mainhand" "MP40"
mp44_stand_walk_back: IS_WEAPON_ACTIVE "mainhand" "StG 44"
....
unarmed_stand_walk_back : default

```

The state machine will first test the conditions for the “mp40_stand_walk_back” animation, because it is higher in the list. The “IS_WEAPON_ACTIVE” function takes two arguments – the location to check for the weapon and the name of the weapon to be checked for. If the “MP40” is in the “mainhand” of the player and it is active (which is set by the ‘activateweapon’ command you put in the ‘pullout’ section of the weapon’s tiki animation), then the state machine is going to play the “mp40_stand_walk_back” animation. If the condition tested was false, then it moves onto the next line. If no set of conditions evaluates to true, the last line provides a default animation to run. Without it, the game would not have an animation to run for the state.

So, if the conditions to the right of the colon in the first line evaluate to true, the game would look for an animation called “mp40_stand_walk_back”. This is defined in the pak0/player/base/anim/mp40.txt file. The corresponding entry in the file is:

```

mp40_stand_walk_back  multiplayer/mp_rifle_curious_walk_backwards.sk

```

This tells the game the name of the animation file and the game goes and plays it. How does the machine know to look in the anims_mp40.txt file for the animation? It doesn’t. When MoHAA first loads up, the developers tell the game what tik files to cache in a couple of different ways. One of the tik files that gets cached is the players tik file. If you recall, anims_mp40.txt file gets included in the player’s tik file in the animation section of the tik. So, the state machine doesn’t know what file the animation is in, but it has a list built by the game when it first loads that tells it where all the animations are located. Clever. This may seem overly complicated, but it makes the environment very powerful so you can do many different things without requiring the game code to be rewritten.

The next section of the state definition, **states**, tells the game what state to transition to next when certain conditions are met. When you are in a specific state, this section gets checked each frame to see if it needs to perform a transition. The states section is very similar to the legs section, except you have state names (all in upper case) on the left side of the colon instead of animation names. Also, the order in which you list the state names is important, as the first one to evaluate to true will be the one that is transitioned into. Here’s an example of a states section:

```

FALL_DUCKED : FALLING
STAND : JUMP_CHECK_HEIGHT "stand"
STAND : +CROUCH_CHECK_HEIGHT "stand"
STAND : VIEW_IN_WATER_CHECK_HEIGHT "stand"
CROUCH_RUN_FORWARD : RUN_FORWARD HAS_VELOCITY !BLOCKED "2"
.....

```

The state names listed on the left side of the colon have their own state definitions within the state machine file. The conditions on the right of the colon are evaluated and the first set of conditions to evaluate to true will transition the player to the state specified. Since the player does not have to transition to another state each time these are checked (ie. you can stay in the same state from frame to frame), there is usually no ‘default’ condition. Some states, like the secondary attack check for the upper torso, do use a default condition in this section.

The final section, **exitcommands**, is rarely used. It is the same as entrycommands, but only executed when exiting the state.

Got all that? Here are a few things to take note of with respect to state machines:

1. The conditions in the legs section concentrate on *things* the player has because that section is concerned with what animation to play. On the other hand, the conditions in the states section focus on the *player's* current condition – running, falling, jumping, etc. This is because this section is concerned with determining the next state the player will transition to.
2. When you load up MoHAA, the game code loads up all your tiks and catalogs the animation names you've put in them. Make sure they're unique, descriptive and match what is in the state machine.
3. The arguments for commands and conditions are enclosed in double quotes, regardless of data type.
4. Always keep in mind that the conditions for different animations and states can be true at the same time. The game will choose the one that appears first in the file.
5. Commands that are run in other files (tiks and scripts) can modify variables and settings that are checked as part of the conditions for animations and states. You'll see more examples of this in the next section.
6. You may have noticed that the weapon name used as an argument for the condition functions in the state machine are based on the 'name' property in the weapon's tik file (in the Init section). This is why the guns stop working if you change their names in the tik files.
7. Some states don't have animations associated with them, but you still need to have a state definition in the state machine so that the appropriate entry commands and state transitions can occur.
8. While the leg and torso state machines run independently, if they conflict in their animations, the torso animation wins. For example, if the leg state machine calls an animation for running while the torso calls an animation that causes the player to stop, the torso animation will be the one to play.
9. The state machine files are located in pak0/global. They are mike.st, mike_legs.st and mike_torso.st.
10. State machines for other games related to MoHAA also have behavior and time related sections in their state definitions. I haven't seen any examples of this in MoHAA and I doubt if it is supported in the game's binaries.
11. Sometimes you'll see "+", "-" or "!" in the condition sections. The "+" signifies the player is pressing the key to do the action (ie. +USE means the player has pushed the 'use' key), the "-" means the player has stopped pressing the button (ie. -ATTACK_PRIMARY would mean the player has let up on the left mouse button) and the "!" is logical negation (ie. !ATTACK_PRIMARY would mean the player is not attacking with the primary weapon).
12. Just based on the game's behavior, it appears that the first states called when the player spawns are STAND and RAISE_WEAPON. If you wanted the player to do something else when they spawned, you may be able to put some code in these events to create the effect you want.

The State Machine Part II: Torso

The torso state machine differs from the legs state machine in four main ways:

1. It has a slightly different format.
2. It takes precedence over the legs.
3. It manages what is in the player's hands (ie. weapons).
4. It handles pain, death and climbing (ladders).

Format

The top level format for the torso state machine is identical to the legs state machine - both are a sequential listing of state definitions. They differ at the state definition level in two ways. First, the torso state machine often precedes the entrycommands section with precedence commands like *movetype*, which tells the game whether the leg state machine should be overridden or not. Second, the torso state machine has a section called *action*, which is similar to (but does not

replace) the legs section in the legs state machine. The full format for a state definition in the torso state machine is:

```
state <state_name>
{
    <precedence_command1> <arg1> <arg2> ...
    <precedence_command2> <arg1> <arg2> ...
    ....
    entrycommands
    {
        <command1> "<arg1>" "<arg2>" .....
        <command2> "<arg1>" "<arg2>" .....
        .....
    }
    action
    {
        <animation1> : <condition(s)>
        <animation2> : <condition(s)>
        ....
        <animationx> : default
    }
    legs
    {
        <animation1> : <condition(s)>
        <animation2> : <condition(s)>
        ....
        <animationx> : default
    }
    states
    {
        <state_name_1> : <condition(s)>
        <state_name_2> : <condition(s)>
        ....
    }
    exitcommands
    {
        <command1> "<arg1>" "<arg2>" .....
        <command2> "<arg1>" "<arg2>" .....
        .....
    }
}
```

The torso state machine also organizes its states slightly different than the leg state machine. In the leg state machine, states that had similarities were listed together. For example, all the states that were about running were listed together.

In the torso state machine, this same convention is followed with one addition. There are some groups of similar states that require some type of thinking (for lack of a better term) before the state machine can know which one is the correct state to run. For example, when the player attacks with their weapon, the torso state machine needs to do different things based on whether or not the weapon is full auto or semi auto. For this reason, there will sometimes be a state definition listed first among groups of similar states that has no animations (ie. no action or legs section) in it. It is just looking at the situation and transitioning to another state based on the circumstances.

Precedence

The torso state machine sometimes needs to override the leg state machine and control the entire player. It accomplishes this via the movetype precedence command and specifying the animations to be played by the player.

movetype

States in the torso state machine that will only control animations for the torso will have 'movetype legs' as a precedence command. This allows the leg state machine to continue to control the animations that are running the player's legs. States that will override the leg state machine will have a different argument for movetype (ie. 'movetype climbwall' or 'movetype anim'). Some state definitions do not specify movetype at all. I have not experimented with them, so I am not sure if they override the leg state machine or not. The only other precedence command that I have seen is 'camera behind', which would appear to set the point of view.

action

Action defines the animation to be played, just like the 'legs' section in the leg state machine's state definitions. Why create a whole new section instead of just using 'legs' like the leg state machine? If an animation in the torso state machine is going to override the leg state machine, then it may need to specify two animations – one for the torso and one for the legs. So, the torso state machine needs the ability to have two sections to define animations. Why call it 'action' instead of 'torso'? The authors of the torso state machine call it the "action state machine". Why? You got me there. One further thing to note: if the action animation is defined as "none: default", then the game takes its torso animation from the leg state machine. For an example of both action and legs in the same state definition, review any of the putaway state definitions in mike_torso.st.

Manual of Arms: Point of View

Since the torso state machine manages the weapon animations, it will also have to manage the first person view animations. When you play in first person, you can't see your legs, so the leg state machine doesn't have to worry about it. But you can see your hands and your weapon in first person, so the torso machine has to manage both first person and third person animations.

If you recall from the chapters on animations, it is the third person animations that control the weapons, regardless of what view you are playing in. Because of this, the third person animations require some discretion on the part of the state machine when dealing with weapon events. Are you shooting a full auto weapon? Does it have a secondary attack? However, firing off the first person animations from the state machine doesn't require a lot of code. Basically, you just play the animation and that is it. This is accomplished using the **viewmodelanim** command.

viewmodelanim <animation> <number>

First person animations are fired off in the torso state machine in the entrycommands section of the state definition using the viewmodelanim command. This command takes up to two arguments. The first, animation, is required and specifies the animation to be played as defined in the fps_anims_<weapon>.txt files in pak0/models/player/base. Note that the game automatically places the weapon's type and an underscore in front of the animation argument in order to determine the animation to play. For example, "viewmodelanim fire" will call the mp44_fire animation if the player has the mp44 active. If he were holding an allied pistol, it would call colt45_fire. It is important to know this when you are creating an fps_anims file so that you use the correct animation names.

The second argument is included for weapon firing animations and single reloading animations. For example, the submachine gun's entrycommands looks like this:

viewmodelanim fire 1

I am not sure what it does, but it is always included in the “fire” and “single_reload” animations, with one exception: the firing animation for grenades does not include it.

Tools of the trade

- good file search tool
- text editor
- binary editor
- winzip or XP
- MoHRadiant (SDK)
- 3d Studio Max (hopefully)
- Photoshop
- Character Studio (hopefully)
- MSC (please god)

Resources

So, where did I get all this crap? Unfortunately, most of it has had to come through just experimenting with the game. Going in, making changes, packing them, kicking up a server and trying things out. The few resources I've found I have stolen from mercilessly for this guide and I'd like to thank the folks that wrote them, wherever they are. Check 'em out for yourself if you don't find what you're looking for in the guide:

- shader manual – by far the best resource made (that I've seen) for modding an element of Quake. It only deals with shaders, but it is detailed and complete.
- alliedassault.com – the Board of Developers forum is frequented by devs and ex-devs for the game.
- q3 source code – gives you a rough idea of the internals of the game engine.
- radiant web site – gives you some references for map making and tying your maps into the game.
- planetquake.com – a good resource, but not MoHAA specific.
- SDK – some reference files were included with the map editor, most importantly an html formatted listing of script commands.
- FAKK2 SDK – a little bit better reference files and (most importantly) includes the game source code. FAKK2 is the closest relative to MoHAA in the Quake world.

Particle Effects

The table below lists all of the commands I could locate for particle effects. I believe most of them are supported in MoHAA, but you may need to experiment. Please note:

- ☐ Offsets and coordinates are almost always in reference to the parent model. Known exceptions are noted in the table.
- ☐ Units of measure are:
 - o distance = game units
 - o velocity = game units per second
 - o acceleration = game units per second squared
 - o angles = degrees
 - o angular velocity = game units per second
 - o time = seconds
- ☐ Coordinates are:
 - o positive x = forward (linear) or roll (angular)
 - o positive y = left (linear) or pitch (angular)
 - o positive z = up (linear) or yaw (angular)
- ☐ Some parameters may be affected by scale and others may not. Unfortunately, there is no rule of thumb to tell. It seems that the choice was made when the command was created based on whether or not it was required for the immediate need.
- ☐ Anywhere you can place a 'random', you can also place a 'crandom' or a 'range'. 'random x' is shorthand for 'range 0 x' and 'crandom x' is shorthand for 'range -x 2x'.

Weapon Properties

The table below lists all of the properties for the weapon class that I have encountered. Some properties are specific to one type of weapon and may or may not work with other types. If you need to see one used in action, just run a text search on tik files for the specific property. Properties in **blue** are specific to single player, while those in **red** are for multiplayer.

Weapon Tik File:

Property	Description	Values	Value Notes
classname	The type of object being defined by the tik file.	weapon	The standard class.
		projectile	Class for projectiles that go with specific weapons.
		turretgun	Mounted machine gun or tank gun. Has tracers.
		vehicleturretgun	turretgun mounted on a vehicle. Has tracers.
		animate	Static weapon (canon).
weapontype	Defines what animations to use with the weapon. Corresponds to the different segments of the weapon bar on the HUD. Projectiles do not have a weapontype. Console command 'useweaponclass' refers to this property. Used heavily in the state machine.	mg	Machine guns (BAR, STG44)
		rifle	Rifles (Garand, 98k, etc.)
		pistol	Pistols
		smg	Submachine guns (MP40, etc.)
		heavy	Heavy weapons (rockets, shotgun)
		grenade	Grenades.
name	The name of the weapon as it is shown on the HUD.	Any text.	Unfortunately, it is used in the state machine to detect what weapon is held in the player's hands. Changing it will affect the animations.
rank	Used by the AI to determine when a weapon should be intelligently employed, ie. "do I use a pistol or this here panzerschreck?"	2 integer values separated by a space.	The first value is order ranking, second is the power ranking. Values are relative to other weapons.
pickupsound	Sound to be played when the weapon is picked up. Unlike the actual playing of sounds, this goes in the server section.	Sound alias.	Sound aliases are defined in ubersound.scr.
ammopickupsound	Same as pickupsound, but played when ammo is picked up.	Sound alias.	Sound aliases are defined in ubersound.scr.
noammosound	Same as pickupsound, but played when firing without ammo.	Sound alias.	Sound aliases are defined in ubersound.scr.
firetype	Tells the game if you will be shooting bullets or separately modeled projectiles.	bullet	Fires a standard bullet.
		projectile	Fires a projectile defined in a separate tik file.
ammotype	Type of ammo that is used in the weapon. Used to determine if you can pick up and use ammo that is lying on the ground.	Any valid weapontype.	Linked to weapontype, so you can reload an Allied smg (45 ACP) with an Axis smg (9mm). Dag nabbit!
		Shotgun	Specific to the shotgun.
projectile	Defines the tik file if the firetype was projectile.	Projectile's tik file name.	Needs to be a fully qualified path (ie. models/projectiles/mypr.tik).
meansofdeath	Tells the game what type of animation to play when something is killed with this weapon. Not specified in the grenade's tik (goes in the projectile tik for the grenade).	bullet	Regular ol' death.
		shotgun	More violent.
		rocket	A bit of the ultra violence. Haven't seen it used in regular weapon tiks, only projectile tiks.
		grenade	Can you say 'boom'? Haven't seen it used in regular weapon tiks, only projectile tiks.
bulletcount	Number of bullets consumed when the weapon fires.	Integer value.	May or may not cause more damage if > 1; haven't tested it.
semiauto	Causes the weapon to fire one round with each pull of the trigger, ie. firing with ammo remaining does	No argument.	

	not cause the weapon to go into a sequential firing event. Must also be implemented in the state machine this way.		
cantpartialreload	The weapon cannot be reloaded unless the clip is empty. Must also be implemented this way in the state machine.	No argument.	
clipsize	Number of rounds in the weapon's clip.	Integer	HUD ammo display does not need to be adjusted.
startammo	Amount of ammo the gun starts with, either in the player's inventory or on the ground.	Integer	Includes rounds in the gun at start.
ammorequired	The amount of ammo required to fire the weapon.	0	No ammo required.
		1 or more	Ammo is required.
firedelay	Time between shots. If semiauto, this is the minimum time between shots, regardless of animations.	Number	Seconds. To calculate for full auto rate of fire, value = 60 / rounds per minute.
maxchargetime	The maximum amount of time to charge the weapon (ie. to hold down the mouse button). For grenades, sets the throwing distance.	Number	Seconds.
minchargetime	The minimum amount of time to charge the weapon (ie. to hold down the mouse button).	Number	Seconds.
bulletrange	In past Quake games, this was the maximum range of the bullet. In MoHAA, it appears to be the range at which bullet spread is applied. In game units.	Integer	4000 = 85 yards.
bulletspread	Randomized bullet deflection caused by firing. Aggregates with firespreadmult when moving. Spread is maximum at top movement speed. Offsets are in world units.	Min pitch	Minimum deflection in the vertical plane.
		Min yaw	Minimum deflection in the horizontal plane.
		Max pitch	Maximum deflection in the vertical plane.
		Max yaw	Maximum deflection in the horizontal plane.
quiet	AI will not be alerted when weapon is fired.	No argument	
autoputaway	The weapon will automatically be put away when it is out of ammo.	No argument.	
firespreadmult	Cumulative deflection to weapon accuracy when moving or firing for an extended period. This is a time delayed effect.	Add	% additional deflection
		Falloff	% falloff (per second)
		Cap	Cap
		Maxtime	Time delay to apply
zoomspreadmult	Boost to accuracy when zoom from scope is used.	Number	The higher the number, the less accurate the shot.
usenoammo	Do not allow weapon to be used if there is no ammo (cannot be activated).	0	Cannot be activated without ammo.
		1	Can be activated without ammo.
tracerfrequency	Frequency of tracers in the single player game.	Integer	One tracer will be fired for every x number of shots.
zoom	Scope magnification.	Integer	The higher the number, the lower the zoom. 30 is max to retain scope overlay (possibly fix with a new shader).
crosshair	Show an aiming crosshair when the weapon is active. Related to cvar ui_crosshair.	0	Off
		1	On
shareclip	Primary and secondary attacks both use the same clip.	No argument.	Just a guess on my part.
movementspeed	Multiplier of base movement speed set on server.	Number	Less than 1 = slower, greater than 1 = faster.
weapongroup	AI setting that tells the game how to employ the weapon (similar to weapontype for player). If you add a	bar	
		mp44	
		rifle	

	weapon to multiplayer that non-players will use, you'll need to specify this.	thompson	
		mp40	
		pistol	
		grenade	
		mg42	
		bazooka	
airange	AI setting that tells the game the distance at which to employ the weapon. Rockets do not have an airange defined.	short	Short range weapon.
		long	Long range weapon.
		sniper	Sniper weapon (how it differs from long I am not sure).
secondary <property> <value>	Sets the weapon property for the secondary attack.	firetype	
		ammotype	
		projectile	
		clipsize	
		ammorequired	
		meansofdeath	
		bulletrange	
		bulletdamage	
		bulletknockback	
		firedelay	
		maxchargetime	
		minchargetime	
		quiet	
		dmprojectile	
		dmammorequired	
		dmfulletrange	
		dmbulletdamage	
		dmfiredelay	
dm<property> <value>	Sets the weapon property for multiplayer.	dmbulletcount	
		dmprojectile	
		dmstartammo	
		dmammorequired	
		dmbulletrange	
		dmfiredelay	
		dmbulletspread	
		dmfirespreadmult	
		dmzoomspreadmult	
		dmbulletdamage	
		dmcrosshair	
		dmmovementspeed	