Advanced FPS Kit
Tutorials
FPS Design Techniques and Strategies

By: Robert C. Fritzen

Contents

1. Introduction

I'm usually not one for ridiculous sized tutorials or documentation regarding anything, but this will be the one time where I "step out of the box" per say and pretty much lay down everything I have on you in terms of design, implementation, and strategies in terms of developing a rock solid first person shooter style game. In this "document" I will pretty much walk you through the general thought process and the general steps of design and implementation in terms of creating stuff that is new, inventive, and also at the same time, original, and remarkable in terms of bringing in the playability factor such that your players will learn to love your game and bring in new faces to the party. So, without further ado, let's jump right into it.

2. Important FPS Factors & The Do's and Do-Nots

I want to hit you right away with some of the most important points, these will pretty much set yourself in a general guideline process of making a great FPS game, a good FPS game, a bad one, and one players will only touch for the sake of having it (You know of which game series I speak). In general, these key points of "engagement" will help you along the path to creating a game that is fun and enjoyable for players of all levels. So, I'll list them up and then give you a nice block of text to go along with it. The reason I want to get these points across quickly and more importantly, frequently, is because it seems like the FPS Developers of the current day are completely ignoring some of these points, and it can ruin what would appear to be an amazing game really quickly.

1. Weapon Balance is the Key

Probably the most important factor in any shooter based game, is to introduce the balancing factor, and nail it down quickly and heavily. About the worst thing you, as a developer can do is to overpower a specific class of weapons just because certain players can do well with them or are "drawing in the crowd" per say by using these weapons. While this might help you in the short term of players who think they are good at your game, it will quickly alienate quite the crowd of players in your game, and in the long run, this will hurt you more than it well help you.

In order to create a balanced play style that will work for you, I generally place weapons into three different categories, and you will notice with time and close attention, that your weapons will also always fit into these three categories.

i. High Risk / High Reward Guns
ii. Generalized Weapons
iii. Low Risk / Low Reward Guns

Now, these categories are easy to pick up on. First and foremost, you have the high risk weapons, such as shotguns and sniper rifles. These weapons will work only in the moment that suits them perfectly, for example, if a player holding the shotgun catches the opposing player in a close quarters fight, they're going to have a huge advantage for being in the right place at the right time, but if they run into the guy with a machine gun, or an assault rifle at a more medium to long range fight, they're done for. Next up, you have the Generalized Weapon category, this is where mainly the assault rifles and sub-machine gun type weapons will fall into, and they are your all-purpose weapons that handle most of the engagements with ease. Lastly, you have the Low Risk/Reward guns that will likely compose of machine guns or other low damage/high ammo weapons. Basically, these guns will be effective in almost every situation, but the low damage output will only help the player win the fight if they have either the advantage of getting a few shots off first or catching the other player by complete surprise.

This might take a little while to actually understand from the perspective of seeing concepts come together, but the sooner you identify where your weapons need to fall, the sooner you can introduce balance to your game. Now, I'll get to this in a later point, but don't be afraid to put a gun that honestly wouldn't feel like it would ever belong in one of those three categories, in that category. I'll explain this later on, but you'd be surprised how making an interesting concept could quickly affect your gameplay.

So, continuing along with Weapon Balance, the next topic is the actual process of balancing a weapon. As you're well aware by now, Torque 3D uses datablocks, or static points of memory stored on the server to transmit information that needs to be kept constant to the client. This can prove to be a hefty challenge for post-launch balance scenarios for game developers without pushing a patch to actually go through and send new .dso /.cs files to every client in the game. However, as you probably know by now, this is where one of my "ninja scripting" techniques will come into play. And this is a two part method that I use for my Legacies game. The first thing you'll need to do is to code up a table of forms (You can store it on SQL, XML, TS or whatever format you need) that includes damage, firing speed, spread, recoil, etc, for each of your guns and store this on your main web server (The one clients connect to for updates). Then, every time a client connects to the game they quickly pull this table and then using a script, parse it into .cs datablocks for each of the guns and their respective projectiles and store it in the proper format locally (IE: compile(), exec(), or for an even more ninja-like approach, eval()). This will allow you to deploy rapid "hotfixes" to your game to balance out factors in a quick and easy fashion.

The last thing I need to talk about is the actual balancing process and how to be good at it. So, once you've coded up a hotfix system to deploy balances, you'll actually need to determine what needs to be fixed. I find that using the community's response to your game to be the most important thing, and again with the number one flaw of big name games, don't just fix something because some guy with a YouTube account with 'x subscribers' or 'x views' says that gun 'y' needs 'z' change. Instead, actually probe the community for what appears to be a common trend with weapon usage, find out where individual elements are causing cases to become overpowered/underpowered and fix them accordingly, and don't be afraid to "buff" a gun, balancing isn't just about "nerfing". Also, the most important factor to a good balance change is not only the pre-process and the actual change, but to pay very close attention to what happens after you make a change. You generally want to ask yourself: "How did this change affect the overall game flow?", "How many players are using 'x' now that is has been 'nerfed/buffed'?", and most importantly "Did changing 'x' interfere with 'y'?". You'll find that sometimes when you apply a balance change, you might actually bring another gun from one of the three categories into a completely different category, only because now that one gun may or may not fit into a good gunfight, that gun 'y' might or might not be in the mix for the same gun fight. You will quickly discover that balancing is a constant fight with no ending victory, but it's an important one to keep your community excited to see how you react to what they think about your game, and this brings me to point #2.

2. Do Not, I Repeat DO NOT Ignore the Community

Too many developers are falling into the pit of death due to their ignorance of the players who are actually playing their game. And with this document fully read, you would easily see how just with a perfect Torque 3D, and some great art assets how easily I could "destroy" Call of Duty AND Battlefield just by applying these principles. No developer and I do repeat for importance, no developer, will ever be successful without the help of their community because it was their community who put them in their position in the first place. If you become ignorant of your community and ignore good requests, suggestions, and other important comments (via forums, emails, social media, etc.) you will place yourself on a one way trip to losing your company / game in a hurry. This is why when you're in the process of developing a first person shooter game, you want to actually incorporate the help of the community. Show them some in-game action, you don't need to be secretive about everything. You should even provide community members a chance to try out your game (alphas/betas) to have them help you along the way to making a good and successful game. And the last point of this, is to always bring in some of the community members to "moderate" or serve and "community moderators" to kind of bridge the gap between the developer and the community. You can let these individuals run your forums, or create special events for you to host to draw in a bigger crowd, and help your game succeed in the long run.

3. Don't Copy Game 'x'

The cardinal sin of FPS development is to say, I want to make the next 'x'. You need to realize that unless you are actually contacted by the publisher of title 'x' (power to you if you somehow manage that), you should <u>NEVER</u> go down that road. It would take quite a hard drive to be able to make a decent variation of game 'x' that is different enough just to avoid being in the copyright violation area, and even then, the hard part is going to be trying to alienate the long time followers of game 'x' to even play your game since they will believe it's nothing more than a clone attempt. Now, there is absolutely nothing wrong with borrowing concepts and making them better (or worse), but don't copy, word for word, and concept by concept, another game, you'll only run into trouble.

4. Stand Out

This one's probably one of the more tricky parts of FPS design, and while it's hardly my favorite, it's just the world we live in anymore, and it works like so. Players only like eye-candy anymore, except for your die-hard vintage gamers (Think people born in the 1970's – 1980's) and your game developers who respect exceptional gameplay. You could have some of the most exciting gameplay in the world, but fall flat in terms of getting anywhere without visuals. This is where you'll need to strive hard in the art department. You'll generally want to assemble a team consisting of a few good scripters and programmers who can nail the gameplay factor and leave a huge portion for the art. I've seen games that have logic failures a plenty, but still thrive because the visuals look absolutely stunning. As a developer, you will want to shoot for a balance between the two, or if you've got the time and the money, throw down a 3 year development period to make the next "Halo", and become legend as you throw down visuals and gameplay like nothing ever seen before.

5. No Rehashes!!!

Under absolutely no circumstances, is it "ok" to simply slap together another game using all of your existing code and call it a "new" game. It is also not ok to try to re-sell all of your old assets to players. This is a huge letdown for your long time community members if you go down this road as they will believe you're simply "milking" the game at that point and are a greedy money thirsty developer. When you're making a FPS game, even if you're using similar (or the same) guns in your next title, you'll want to put some new effort down. Heck, even re-create it if possible, the goal of development is to get better at it with each successive title, and the more guns you make, the more you'll discover that it's easier to do what you did allowing you to branch out into more complex and even cooler design elements.

6. Customizations

Being the last key point for FPS Development, I need to end it with something else that needs to be kept in the open. While players love to "play" a game, the factor that will sustain it well beyond its release is to give the players customization options, and lots and lots of customization options. This can come in a wide variety of shapes and forms, such as in-game map editors, the ability to "mod" your game, unlockables, etc. Anything that lets the player create something unique and something they can personally take away from the game will give you some major bonus points in the long run in terms of how many people play your game well after launch.

So, that covers the "big" ones. There's plenty more to actually discuss in terms of "important" dos and don'ts but those can come across later on.

3. Conceptualizing, The First Steps

A great game always begins with an idea, something like imagining a scenario of “what if” or “what would happen if”, and First Person Shooter games are no different at all. When you first start off the game, you’ll want to create a concept and it needs to be something that everyone on your team can jump aboard with and stick to, as the concept of the game is what will eventually become the game itself.

I usually find that it helps to write down ideas for games, or just to pen some notes or maybe a level design concept or two just to get the idea. I even got bored enough one day to use MS paint to draw a map idea (and then stuck that map design on a screenshot of the map itself while I was developing it):

Blizzard
Lwr Area
Crevass
Elv. Area
Crossing
Tactical Uprising
Beginnings

The point of this is to commit a concept to be designed. And you will quickly find that most elements of your game will revolve around conceptualizing the idea as the first step before actually jumping on it. For example, let’s say I want to make a brand new laser rifle. Well, what’s the first thing I did? I thought: “I want a laser rifle”, which immediately is followed by this: “Ok, so what do I want my laser rifle to do?”. Bingo, right there you’ve got it, the big concept phase. So, this is where the pen and note taking comes in handy, you’ll want to scratch down your ideas somewhere to easily remember them, and honestly, you’ll have those days where something big just slaps you in the face and it could honestly become a huge hit, it becomes up to you at that point to turn the concept into something further. The conceptualizing phase will pretty much encompass every element of your game as you design it. And it’s important that you give everything a place and a purpose in the game, the last thing you’ll ever want is something completely random that just doesn’t fit at all.

Next up, we’ve got the Genre of the game. So while your new shooter game will classify to most players as an action game, or a first person shooter game, you’ll be thinking of it from a different view. You’ll say: this is my game ‘x’, which takes place in ‘y’. ‘Y’, is the Genre of the game, or basically the setting of your game, and it will encompass a wide variety of gameplay aspects, such as players, weaponry, hud elements and many more things. One of the big no-no's of FPS design is to stick something completely out of Genre in your game. If the big boss orders up a modern shooter game, you better not say, "I want a bow & arrow" or "Alright! Star Wars Blaster time!", your players will pretty much laugh at you before never returning to your game. Now, there are some exceptions to the Genre rule, obviously if your game incorporates some element of space/time distortion (Time/Space Travel by some means), then it's alright to break the Genre rule to a certain extent, but in all honestly, this is something that should be automatic, and you'll be able to pin that problem in a heartbeat if it's there, the game just won't "feel" right.

The last thing you need to always keep in mind when conceptualizing a game is to think, "How can I make my game new and exciting?". The absolute last thing I want is to grab a game that looks awesome, and then plays exactly like Call of Duty, and you need to keep that in mind too. When you're designing the first concepts of a game, you always need to have the new elements in your back pocket and you need to find places to deploy them in a way that players will be able to pick up on quickly and have that sense of the "New FPS Game" feel. It's kind of like how you'd have that "New Car Smell" when you buy a brand new car from the dealer; you want your game to have that same feel when players get it for the first time.

4. The System Approach, and Task Management

Computer programming is an amazing thing, and it has quite a deal of advanced features to help you get a game cranked out in quite a fast time period if you're capable of designing it, but honestly the biggest feature I believe you should devote your time to learning and loving, is object oriented programming. Object oriented programming (shorthanded OOP) is a way of "systematically" creating a new object to perform a set of tasks that only that object would be suited to handle. For example, let's use a T3D math object, the Box3F. Box3F is a class that defines a 3D box using x, y, and z coordinates, and has precision down to the floating point level. Now, Box3F can handle anything related to world space area coordinates or hitboxes perfectly, because it has specially coded methods (Object specific methods) to handle tasks based on the individual element fields of the Box3F. Box3F however, would be a poor choice in circular or spherical calculations without using a large number of boxes to accomplish the calculus definition of the area method, at which point we'd create a new Circle3F or SphereF object to do this work.

OOP works for a huge amount of jobs in the FPS world, and it even handles some of your scripted systems, because an object is simply a "class" in terms of computer programming. And as you've learned a lot about by now through programming, is that classes can do some amazing things, especially when it comes down to inheriting classes through others to create new objects. For example, assume I have a base class called "Object", now Object would define a great amount of parameters such as the properties of the object, where it's located, etc, and from Object, I could then define a whole set of different SubObjects based on the code of Object, such as Player, Weapon, Vehicle, Projectile, etc, because they work in a similar fashion, but differ on how they are applied. You can take this exact same concept and apply it to your work in FPS design.

When you're making a FPS game, it's best to try to picture where each individual element of your game belongs, and then to place it in a hierarchy map. The ultimate goal of this is to minimize the amount of work needed to accomplish a project, and you'll see that putting a day or two into designing a good FPS hierarchy map will trim a huge amount of time out of your work load, and can even lead to performance gains for your game in the long run. This fits in the job of Task Management, as you want to assign individual elements of your game to a certain task. For example, let's say I have a specific game mode. I could just jump right into the script and define a new game object to store individual game elements, teams, scores. See how doing this would take a huge chunk of my time away? Now, instead let's assume a base "Game" object, which contains important methods such as Start / End / Score. From there, I could create another object MyGameMode and have it inherit Game. In Torque you can inherit objects like so:

```
%base = new ScriptObject(Game) {
  class = “game”;
};
%new1 = new ScriptObject(MyGameMode : Game) {
  class = “MyGameMode”;
};
%new2 = new ScriptObject(MyGameMode) {
  class = “MyGameMode”;
  superclass = “Game”;
};
```

As you can see, there’s a few ways to accomplish inheriting in Torque, but the overall concept here is that MyGameMode now has all of the functioning of Game, but now, I can code the individual pieces that make MyGameMode unique, without needing to re-code elements that all games share. And for those of you who haven’t seen those Torque keywords, here’s a quick tutorial. Class tells Torque what class the object belongs to, this by default in our case is ScriptObject, unless I tell it what it is, it will simply be a default parameter. If you do define it however, Torque then says superclass = defaultClass and class = definition. the general object creation syntax is:

```
%localObject = new class(objectName [: InheritedObject]) {
  Object paramaters
};
```

And from there you could work with your object as you normally would. Now, SuperClass is different. SuperClass is the “Absolute” class of the object, now this by default is the same as the class. However, when you define a Class of a new object, the absolute class still remains as the normal class even though; you’ve created a new class. Example here is for the %base object. I’ve told Torque that %base is class “game”, but Torque doesn’t have a class named “game”, it’s a custom class I’ve defined in script, so how the engine then works is to define a SuperClass field of the original class, or in our case “ScriptObject”. This is the general essence of object inheritance. Imagine the SuperClass as another way of saying “Parent”. Now, to show you how parenting would work, here’s some more script:

```
// Define Game Class/Object
%base = new ScriptObject(Game) {
  class = “game”;
};
function game::start(%this) {
  //perform game start here.
}
function game::end(%this) {
  //perform game end here.
}
//Define Sub-Class
%new1 = new ScriptObject(MyGameMode : Game) {
  class = "MyGameMode";
};
function MyGameMode::doSpecialCode(%this) {
  //Game specific code goes in here.
}
function MyGameMode::start(%this) {
  %this.start(); // Call Parent function (Parent::start(%this); works too!)
  %this.doSpecialCode();
}
```

Now let's talk a little bit about the above and how it works. So I would code up game::start() and load it with elements that all game beginnings would have in my FPS game such as team assignment, player spawning, weapons, etc. Now, let's say in MyGameMode, everyone needs to have weapon 'y'. Well, I could then simply use doSpecialCode to change the starting weapon, or augment the existing start(); code with MyGameMode::start() and write my own code in that place to do so. You'll also notice there that I did not write MyGameMode::end(), this is because it's not necessary, don't believe me. Throw an echo in each of the above, and try doing MyGameMode.end(); You'll see it works, and that is because MyGameMode is inheriting the methods from Game, and anything not overwritten by my script above, will simply co-exist in the new object as it does in the existing one.

Right away you should be able to pick up on the numerous different applications that the code sample above can do for you. By simply forking off a multitude of tasks that share similar applications but function slightly differently into a nested object chain, you can cut down on a load of code writing, which in turn cuts down on the time needed to deploy a specific element. This is why when you design a FPS game, after you get your concepts together, the next thing you should immediately follow with is to create a game hierarchy map to determine which systems can be coded in the class style, and which ones can suit themselves outside. Once you've got that general design scheme figured out you can dive into the next step of FPS development.

5. Settings and Places

Now, we're in a great position to begin some elements of the code work. If you're still lacking on some gameplay ideas, this is the time to stop and actually go through how your game needs to play. Most FPS games offer a single player campaign mode and a multiplayer mode where players can compete with each other or battle it out against each other. In terms of your game, you'll want to keep both environments close together, which is why you'll see some FPS games go down the road of setting their multiplayer maps in environments seen in the campaign experience. While that's not always a bad idea, I'd steer clear of the simply copy/paste ideology for making MP maps from SP maps, while it's alright to place them in a similar setting, don't simply copy elements of one directly to the other. You'll find that it can introduce some map based imbalances that will really hurt the overall playability of the map.

This is also the time to actually dive into how you want the game's storyline to come into play. While you could simply have a multiplayer-only deathmatch arena, there's still the backstory element that needs to fill in somehow. Where did these soldiers come from and why are they now in a gladiatorial duel to the death per say? You'll want to incorporate your individual characters and place in the elements of the story in a way that fits the environment, again, no medieval dudes or dudettes with Star Wars Blasters… just… don't do it ☺, the last thing I need is a game where I hear ye-olde accent complaining about their heavy shiny armor while holding a lightweight laser pistol thingy (although it would be freaking hilarious).

You'll want to start to create the story for your game here, try to fit as much of your current hierarchy systems into this as possible, it will make the work of accomplishing individual gameplay – story elements much more easy. The ultimate goal of this design phase is to get your game's story in order to absolutely nail down the element of the Genre and to start to idealize where your game will take place, which is where map development comes into play. While I already gave singleplayer maps a mention, let's now talk about maps in general and then we'll get the important concepts for good multiplayer maps.

Try to design map elements and events that fit the story and the surrounding environment. If you have a large city under siege, try to have giant buildings exploding or have them come crashing down at the right moment after getting struck by a missile or some other projectile, you'll want players to "Feel" the experience of the game while running through it, and nothing does it better when the combat environment plays a pivotal role in the gameplay. You might event want to go down the dynamic map road for Multiplayer, give the players the ability to use the map to kill off someone in a bad spot, or even change the flow of the map to disrupt the enemy or help out their teammates.

This leads into the next topic of map design. When you make a FPS game, this will have a heavy influence on the game development process as it will rely on it. When you make a FPS game, if your game incorporates any kind of vehicular combat, then your map sizes need to fit that accordingly. Generally it's a good idea to have maps of differing sizes to cater to all playstyles. For example, give the players who like vehicles and sniper combat a few big open maps, maybe have a little town in the middle for some rural combat and then keep the outskirts of it open with some hills to allow for vehicles to move around in. Then you'll want your all-around favorite game friendly medium sized maps that allow pretty much all forms of playstyles, maybe they have vehicles, maybe they don't but they'll still cater to everyone in some way or form. And lastly, make a few high paced action small maps for those lovely free for all deathmatch experiences where the games explode quickly and play at a high pace.

The overall goal of map design is to have the map suit the battle type, the best way to determine if you've done a good job is to play the map from both sides of the field and to see if both sides get into the fight zone of the map around the same time. The worst thing you could do (unless you're making a defense/assault game mode) is to give one team a map based advantage over another team. Make sure for each disadvantage of one side, there is an equal disadvantage on the other side, and make sure each advantage is kept with another. Towers of Power are evil so make sure nobody gets to lock down a single zone of the map and dominate the entire game for doing so.

The next important point is to incorporate a game friendly spawning system. One of the worst things to happen in a FPS game is to spawn and die a few seconds after spawning with absolutely no chance of winning the gunfight you're thrown into. Make sure that if the maps are going to be too small to prevent this that spawn invincibility is provided to scare off potential spawn campers. Nobody will spawn camp when they try to kill someone with spawn protection active carrying a shotgun. You'll also want to consider making spawns occur in what I call "one-stop" shops, or basically a part of the map that is inaccessible once you leave it, this will prevent anyone from camping the room, and by creating multiple or even randomized (teleports) exits, you can prevent it altogether.

By placing all of these elements on the table, you can create exquisite environments that can change through player controls (Dynamic Maps), which feel right for the Genre of the game, and provide a lot of action when there needs to be while keeping the player away from it if it seems to be ridiculous. Also, remember that as a developer for multiplayer maps, it's important to design maps that are devoid of "Towers of Power" and team based advantages, and that spawn protection and prevention of spawn camping makes for a better gaming experience that your players will love and respect. And remembers points 1 and 2, who said Balancing only can be done for weapons?

Hey, Balance your maps too! Give the spawns a change now and then to mix up the game flow from time to time.

6. Lovely Weapons

Now, we need to talk about the important part about any first person shooter game, and that's what kind of "toys" you'll be giving your players to use. Like we've already discusses before, the Genre of the game will play a pivotal role in the types of weaponry you need to incorporate to your game, certain guns will fit perfectly, and others will seem completely outlandish and out of place compared to the others, the specifics I'll be going over in this section will be in general regards to actually craft interesting weapons that fit the game.

First and foremost, we need to establish a "Palette" of weapons to create, and these will act as the different types of guns your players will be able to use, such as Assault Rifles, Shotguns, Machine Guns, Etc. Once you've established the categories of weapons, we need to start thinking of what types of weapon fit the Genre. Your space shooter games will likely have that futuristic feel to it, but could have a modern day weapon or two lying around, whereas a modern shooter would do well with weapons known by name to many people such as AK-47, Tar-21, and so on, It's up to you to decide which weapons will make the final cut for your game.

For more creative games, you can find that by blending different parts of weapons together, you can create entire new weapons that can perform differently compared to their counterparts, I briefly mention this in the AFPSK Modelling Series if you're more interested in checking that out, but the general idea to accomplish weapon making is to get a general "Reference Image" together, something that you can design a gun from.

Once you've established what guns will make the cut, you need to fit the weapons into their respective balance categories, and establish how the gun will perform in certain situations, and Torque3D provides you quite a load of projectile options such as speed, damage, and effects to accomplish pretty much any type of projectile you need for the occasion. As for the other factors of weapon balancing, most of the other properties will be handled by the gun itself, such as ammo, recoil, projectile spread, all of these can be easily customized and even adjusted using overrides of general weapon functions such as ::onFire(), as a designer, it's up to you to determine how the gun is supposed to work, and how the projectiles will interact with objects, and how damaging they will be.

While we're still on the topic of weaponry, now is also a good time to talk about "Heavy Weapons" or "Super Weapons". Some games incorporate vehicular combat in their games, which allow players to hop into vehicles and tanks to take on the opposing team, and usually these vehicles and their weapons will outmatch a standard soldier's gun with ease. The trick to making a good FPS Game with vehicle combat is first and foremost to ensure both teams have access to these vehicles and that the vehicles are not overpowered/underpowered to each other, but most importantly, you need to ensure a way to break a potential lock-down by vehicles with heavy soldier weapons such as missile launchers. Think to yourself here, because this will also introduce a little “task” of ensuring you don’t overpower the heavy anti-vehicle weapon to the point where players might think they can take it against players as a primary weapon. A good design is to make it a secondary weapon that only works well against vehicles, maybe if a direct hit, on a player, and obviously ensure the ammo and reload speeds are accommodated to the fact of the power weapon.

The other class of weapon in this category would be a “Super-Weapon”. Games that incorporate these weapons usually associate players needing to accomplish a task to obtain the weapon, such as obtaining ‘x’ kills in a row without dying or reaching a certain point value. Super-weapons are usually triggered by the player on the ground and they either work by the player designating a target for some form of artillery strike, taking control of a powerful object (such as a vehicle, or a turret), or even (with the AFPSK Radar Add-on) selecting a target position on a mini-map and watching the fireworks come in to devastate that location.

For super-weapons the most important thing is the aspect of balancing them to a point where the method to obtaining the weapon matches the power of the weapon. IE: Don’t give nukes to every player that scores a 2 kill-streak. You also need to consider the properties such as area of effect, lingering damage, and how deployment of this weapon will change the game (or even the map). All of these factors to fit in the “Wheel-Of-Balance”, otherwise the game itself could quickly turn in the favor of players who might be more skilled at obtaining these weapons. And what I find to be very important is ensure every player has a fair chance, regardless of their skill level or play style, to get a fair shot at these weapons.

7. Game Modes

Next up we need to discuss the modes you provide your players. I briefly went over the implementation idea for the systematic approach a little while ago, but now we need to actually discuss the gameplay and different styles of gameplay for your game, was that confusing enough? Good!

First Person shooter games generally invoke a level of competitive gameplay for player versus player interactions on a player based (FFA) or team based (TDM) experience where basically players compete directly against others with what the game provides itself, these are commonly your deathmatch variants.

Something to realize is that in all FPS games, deathmatch is always the game mode, but there could be an underlying primary objective to the game itself which is where other variants of the game are introduced.

Now, the actual process to developing a game mode for your players is first to go back and fit all of the prior factors together, when making game modes you need to consider how individual maps will work with the game mode, how the weapons will come into play at different times and other important concepts such as vehicles, superweapons, map related events. All of these things you’ve come up with so far will play their important roles in the actual gameplay itself, so in order to accomplish the game design you’ve originally intended to complete, you need to always consider how the balance of your individual game elements will play important roles for the gameplay.

Next up is the design of how the game is won, lost, or even how the game can be “Deadlocked” to a point where a draw might fit the final outcome. To do this, you usually want to create point values for doing in-game tasks, such as killing an enemy player, saving an ally from death, or even completing an objective in the game. All of these point values need to be totaled up as they’re added to see if a player or a team has eclipsed the necessary maximum score for a victory.

And that should about cover all of the stuff you need to do outside of the computer, hurrah! Now, let’s step into the computer and actually get cranking on this project!

8. Program Logic 101

So up to this point, we've been pretty much covering design principles and how the workflow of the pre-programming phase of the standard FPS project works, at this point in time you should have all of your ideas straightened out and pretty much in the solidified point where only slight logistical tweaks may be made here and there. Now we can actually step into project development. I know many of you, unlike myself, may not be the most familiarized when it comes to developing logical structures and programs that make the most use out of localized memory for performance, but I'm here to alleviate those issues within this next section of this document. First however, I'm going to do a brief overview/review of some of the "big" TorqueScript syntax properties. Just a little forewarning, this chapter is a little longer, and while relevant to some people (mainly starting scripters), some of the stuff here might be familiar to you guys who already have programming experience, so feel free to jump ahead if you already know how to do general coding.

First off, all of our work is done in functions or definitions of code where individual logic structures are executed. You could just as easily remove all of the functions in the engine and have one giant .cs file for the entire game, but it would be extremely difficult to manage the individual and redundant (repeated) tasks that need to be done in games, especially shooter games. There are a few different ways of actually using a function (calling):

```
function exampleCode() {
  echo("testing one, two, three");
}
//Method 1: Direct Call
exampleCode();
//Method 2: Timed Call (Indirect Call)
schedule(1000, 0, exampleCode);
//Method 3: Indirect Method Call
eval("exampleCode();");
```

All three of the above will call exampleCode at the respective interval. Methods 1 and 3 are the quickest, although method 1 is the absolute quickest in terms of function calling. Method 2 is used when you need to put a delay between functions, for example a timed respawn code for example would fit the bill. Method 3 is a more advanced method for calling, as you can nest code in the function call for advanced filtering calls. I won't actually cover that topic in this document, but know that they can be done by using the Indirect Method Call.

Now some general "rules" for naming. Functions cannot start with numbers and cannot contain special characters, generally, you're restricted to letters and numbers for functions, and you should treat variables the same way. In Torque, there are two types of variables: Local Variables, and Global Variables. The definition of these individual variable types is determined by the leading character on the variable:

```
%localVariable = 1;
$globalVariable = 1;
```

These variables are subject to what is called Variable Scope, or where the variable is "valid" to be used. Global Variables are as suggested, global in scope, and can be used pretty much anywhere (obviously disregard client/server differences here). These variables are usually best suited to store game constants and settings. Local variables however are a little more restrictive, and more important in terms of development.

```
%IAmAVariable = true; //Illegal & Access Violation Prone
function myTestCall(%var1, %var2) {
  %var2 = 1;
  echo("Variables: "@%var1@" : "@%var2);
}
function mySecondTestCall(%var1, %var3) {
  echo("Special Variables: "@%var1@", "@%var3);
  echo("Var2? "@%var2);   //Illegal, %var2 is not local to mySecondTestCall()
}
myTestCall(1, 2); //Prints: Variables: 1 : 1
mySecondTestCall(1, 3); //Prints: Special Variables: 1, 3 and then Var2?
```

This above sample should cover variable scope as it pertains to local variables. Pretty much the rules are as follows:

1. Do not define local variables outside of functions (This can cause a game crash if the script is loaded twice)
2. Local variables can only be used in the function they are defined in
3. Local variables are not "usable" in other functions unless passed to them as a variable.

And that covers variables. Next up is logic structures. There are a few different variations of these in Torque so I'll cover them briefly, and quickly.

First and foremost (mostly used), are your standard if/else-if/else trees and they work very easily:

```
function logicStructure(%one) {
  if(%one == 1) {
    echo("1 == %one, Huzzah!!!");
  }
  else if(%one == 2) {
    echo("%one == 2*1, Huzzah!");
  }
  else {
    echo(“Y u disappoint???”);
  }
}
```

Feel free to send numerous different numbers to `logicStructure()`, and you’ll quickly see how this works. You could just as easily expand this logic tree of else if on and on to any point you would deem necessary for the structure. Now, the second way to accomplish this same tree is to use a **switch** statement. It works in a very similar fashion although with a few differences:

```
function logicTreeTwo(%one) {
  switch(%one) {
    case 1:
      echo(“ONE!!!”);
    case 2:
      echo(“TWO!!!”);
    default:
      echo(“Nope…”);
  }
}
```

And this one will pretty much work like the one before it. Now there’s one more operation to perform logic checks, and it’s by the mean of the conditional operator. This operator is a little less known to novice programmers as it’s a little more advanced, and it works on two conditions only, the condition that is true, and the condition that is false. Through this though, you can easily nest a structure, here’s the operator syntax:

```
%operation = [Condition] ? [If True] : [If False];
```

To expand with an example:

```
function oneOrNo(%var) {
  %result = %var == 1 ? “ONE!” : “NOPE”;
  echo(%result);
}
```

And it’s that simple. Now, I mentioned nesting these together. Say I wanted my 2 condition in there as well? It’s pretty easy to do, all I’m going to add is a second run through on the [If False] condition to test again to see if it’s two:

```
function NumberMe(%var) {
  %result = %var == 1 ? “Got a One” : (%var == 2 ? “Got a Two!” : “Bad Number”);
  echo(%result);
}
```

And you could continue nesting as long as you wanted to. That covers your logic structures. Next up we need to dive a little further into the condition statements. Up to this point, we've simply been asking if a single condition is met (or true). You can easily expand to do multiple checks in the code by means of the following operators:

&& - And Statement (Both must be true to evaluate)
|| - Or Statement (One must be true to evaluate)
! – Not Statement (Flips result to false)
== - Equal To Statement
!= - Not Equal To
$= - String Equals
!$= - String doesn't equal
> Greater Than
< Less Than
>= Greater than or equal to
<= Less than or equal to.

All of these are your standard operations to perform when checking for conditions on statements. I'll let the standard coding documentation take it from here in terms of learning these since it should be straight forward to accomplish, about the only mention I need to make is when you want to use switch with strings, you need to define the call as:

```
switch$(%stringVar) {
```

Next up, we'll do loops, and there are only two that we need to cover for our purposes here. There is a while() loop and a for() loop. The while() loop is more applied when the condition of a test will have an unknown ending to it, and a for loop is more applied when the ending value is known. Examples of each are presented below:

```
function loopingExample() {
   //while:
   %myNumber = getRandom(0, 10);
   while(%myNumber != 10) {
      echo("Got :"@%myNumber);
      %myNumber = getRandom(0, 10);
   }
   //for:
   for(%i = 0; %i < 10; %i++) {
      echo("Now on "@%i);
   }
}
```

Both cases here are fairly standard in terms of how loops work. The first one will keep going until it generates the number 10 (which sometimes might take a bit ;)), the other however will run a grand total of 10 times printing numbers from 0 to 9. As you can clearly see, these loops use the exact same condition statements as I have presented above, so make good use of them.

If you need more scripting references or need additional reading, you should be sure to read through the engine documentation as it provides a great deal of insight. You can find all of that here: http://docs.garagegames.com/torque-3d/official/index.html?content/documentation/Scripting/Overview/Introduction.html

Now, we're going to jump gears into some developmental strategies using these principles, I'll give you some key improvement pointers as well as some insight on my coding style and how I usually approach the problem presented to me.

9. Programming Strategies: Control Systems

When it comes down to actually programming the game, it takes some code that is a lot more complicated and advanced than the basic sample pieces I've given you above. However, you'll notice that everything I've done above will always apply to the actual programming challenge. So, now let's actually tackle something bigger in terms of programming perspectives.

Let's say I've just been awarded the contract for the next "CoD" (I'd honestly throw myself off a cliff if I ever did) and I need to put together the killstreak reward system for players who go on killstreaks. Let's assume my game has 25 different killstreaks, some of them with shared point values, and let's also assume that when players die, they lose their current streak progress.

While some of you would immediately jump into Player.cs to start coding up some bits and pieces, let me walk you through my design process for tasks. First and foremost, I need to identify everything that needs to be referenced here. Since we're dealing mainly with player death, I'll be using the Player. Other references may include vehicles being spawned from streaks or Projectiles, so I'll keep those handy too.

Now, I've got to think, what kind of main tasks need to be done for this system, and is it feasible to create a new Class to handle it? For a killstreak system, I need to store individual player killstreaks, I need to store what streaks they are using, which streaks function in which way, and a few other things, so yes a Class would work here. At which point, I draw up some method stubs:

```
$Game::KillstreakCore = isObject(KSCore) ? KSCore : new ScriptObject(KSCore) { class =
"KillstreakCore"; };
function KillstreakCore::initialize(%this) {
   for(%i = 0; %i < ClientGroup.getCount(); %i++) {
      %cl = ClientGroup.getObject(%i);
      %this.killCount[%cl] = 0;
      %this.streaks[%cl] = %cl.streaks;
   }
}
function KillstreakCore::onPlayerDeath(%this, %killer) {
   if(%this.killCount[%killer.client] !$= "") {
      %this.killCount[%killer.client]++;
      //check for streak here…
   }
}
```

And then from there I can expand into more methods, such as the awarding methods, the calling methods, and more. Once I've coded up the basic structure of my Classed system, I then go into files such as Player.cs and others to assemble to system calls, or "hooks" into the new system. From there, I can test it much more easily because instead of debugging %player.dump(); which as you know (or may not know), will print a load of information to the console, whereas if I need to debug my system $Game::KillstreakCore.dump(); will function in a much more smooth way. Just by assembling your key gameplay mechanics in systems rather than in blocks of code that function only because they function can prove to be a very useful debugging tool. Now, this doesn't mean jump right into the engine and convert everything into a system model, that would likely take up a chunk of your time, and then you'd have to run through the debugging process.

Sometimes, you just need to know when and where it's ideal to deploy the system mechanic, I've given you plenty of examples above. Some good places to use this are:

- Game Mode Variations
- Weapon Attachment Systems
- Superweapon Systems
- Medals/Rewards
- Game Scoring (Could hook into Game Modes)
- Player / Game Customizations
- GUIs and Client->Server GUI Interactions
- EX: Pre-Game Lobbies

And here's some examples of places not do use it:

- Weapons
- Vehicles
- Players
- In-Game Objects
- Map Events
- Static Items (And Variables)

Usually the common sense of things is to avoid deploying unnecessary systems for things that already run on their own, Datablocks and their respective objects usually fall into this category. Now, while the objects themselves might not qualify, you can see that operations involving the objects are usually the place to deploy this approach, and that should be the general rule of thumb here.

With all of that in mind you should generally follow this kind of route when facing tough scripting challenges:

1. Identify the task, what needs to be done in the function?
2. What variables are available for use?

3. Do I need to make any external calls (IE: Mathematical calculation functions)
4. What conditions do I need to check for in my function?
5. What kind of result does my function need to end with (return, void, etc)?
6. Can this operation be broken into multiple smaller functions for easier coding and feasibility?

After you ask yourselves these questions, go ahead and actually write a “scratch” function in terms of what the operation of the function needs to be. Once you finish it, hop in the engine and actually test it, be sure to ask these questions when testing functions:

1. Is the function running through completely without any errors or crashing?
2. Am I getting the result I need?
3. Is the result of the function repeatable (IE: function works every time when you use it?)

This is the place where the echo() and error() functions will come in very handy. Use them and use them often to determine what kind of numerical or string results you are getting from your functions, and I cannot stress the importance of using error checking enough for you, it’s imperative that you’re checking for errors in all of your functions. I’ll go over a few of those tools later on to help you out.

10. Programming Strategies: Game Mechanics

Now let’s move a bit away from the systems and do some in-game mechanics. Game Mechanics are basically defined as the interlinking of systems and functions to perform tasks in the game itself, in more simple terms; it’s the combination of your logical systems into the actual gameplay.

Programming individual gameplay elements will usually involve one of three things.

1. Datablocks a-plenty.
2. System Programming (IE: Game & Variants)
3. Math, Math, and more Math.
4. Object Programming

And let’s break down each. Starting with the Datablocks a-plenty category. The reason this will be extremely prevalent is due to the nature that all objects in game that are constant in terms of all players will be in the form of a datablock, now aside from the individual map objects which can be handled as Static references, you’ll be using datablocks for everything in terms of players, weapons, projectiles, and the like. A very simple definition appears as so:

```
datablock ItemData(Petrov) {
  category = "Weapon";
  className = "Weapon";

  // Basic Item properties
  shapeFile = "art/shapes/weapons/Ryder/TP_Ryder.DAE";
  mass = 1;
  elasticity = 0.2;
  friction = 0.6;
  emap = true;
  PreviewImage = "petrov.png";

  // Dynamic properties defined by the scripts
  pickUpName = "Petrov pistol";
  description = "Petrov";
  image = PetrovWeaponImage;
  reticle = "crossHair";
  heldWeaponName = "Petrov Pistol";
};
```

There are a few things going on here. You can see this is an Item datablock from the definition line (ItemData). Each is defined in the engine source code to be used in the engine itself, and each has its own unique properties. From here you may also define your own properties of the datablock (these are pretty much constants to all players since datablocks are constant). The best way to envision datablock properties is to assume they are a global variable that doesn't change once loaded.

You can also copy a datablock into another datablock and simply adjust properties you need to. This allows for things such as "BaseItem", "BaseProjectile", to be made where you can simply code up default datablock methods for each. However at that point, you're likely in a better position to overwrite the direct datablock function. Here's how you copy an existing block:

```
datablock ItemData(Petrov2 : Petrov) {
   pickUpName = "Petrov pistol 2";
   description = "Petrov 2";
   heldWeaponName = "Petrov Pistol 2.0";
};
```

And it's pretty much straight forward coding at that point using the same properties of object inheritance I discussed back in the systematic programming section. Next up, we'll talk about how games implement system programming.

As I briefly went over back in the same section, it's fairly efficient to base game modes off of a single game object, and in fact, Torque 3D provides this by default by means of their individual game types inheriting from a GameCore class, which provides the generic methods present in all games such as scoring, time limits, and callbacks for player deaths. You can use the approach and knowledge gained from the control system programming section above to easily craft inherited systems that follow the basic game approach and run with it, similar to how I did this:

```
function SixShooterShowdownGame::startGame(%game) {
   parent::startGame(%game);
}

function SixShooterShowdownGame::endGame(%game) {
   parent::endGame(%game);
}

function SixShooterShowdownGame::onGameDurationEnd(%game) {
   parent::onGameDurationEnd(%game);
}
```

Basically, I'm just stubbing out functions (functions with no code, or simply functions that point to other functions) and instead allowing GameCore to handle the task of what I have presented. You can even expand the function calls at this point to instead, call additional code. Parent::Code() basically executes the entire function call of the inherited function, so you could for example write custom code that runs before, or after the code you load normally functions.

The next thing, you'll need a lot of when programming gameplay, is your friendly neighborhood mathematics, and I'm talking mathematics of many different levels here. Some methods will require different variations of mathematical calculations which generally run along these lines:

1. General Mathematics
2. Position Calculations
3. Vector Mathematics
4. Trigonometric Calculations
5. General Geometric Operations

And like above, I've got some examples of each to demonstrate. So we'll start off with general mathematics. And you've seen this pretty much everywhere in programming. One way or another, you always seem to "tweak" the numbers in a sense, and sometimes you even need to mess around with the numerical operations to accomplish your results. A perfect example is for totalization operations:

```
$Numerics::Total = 0;
function growNumber(%to) {
  $Numerics::Total += %to;
  echo("Added "@%to@". We now have: "@$Numerics::Total);
}
```

The same can be done for subtraction, multiplication, and division. You've also got another operation to use for general mathematics called the modulus operator, which in simple terms returns the remainder of a division. So, if I did 3 / 2, I'd get 1.5, but if I did 3 % 2 (% is the modulus operator), I'd get 1, which is the remainder of 3 / 2.

Next up, we've got our position calculations section. You people who have loads of experience working with matrices and vectors will be right at home here. A position in terms of Torque3D is a 3 dimensional Vector: <X, Y, Z> but is stored in string format in a space delimitated style: "X Y Z". That makes methods such as getWord and setWord extremely useful for getting individual position elements and working with them:

```
function getIndPosElement(%position, %element) {
  switch$(strlwr(%element)) {
    case "x":
      %word = 0;
    case "y":
      %word = 1;
    case "z":
      %word = 2;
    default:
      error("element is not 'x', 'y', or 'z', defaulting to 'x'");
      %word = 0;
  }
  return getWord(%position, %word);
}
```

Here’s my example for getting individual x, y, z coordinates from a position. You could use the above to easily pull a coordinate out of the position and then do your standard mathematics on it, or you could instead opt for the next level of mathematics which I’m going to show you, which is instead to work directly with the vector that the position is.

There are plenty of vector operations readily available in Torque3D to handle all forms of mathematics you need to do. Let’s revisit my position example above, but instead let’s get the result of adding height to my position:

```
function addZTo(%position, %amount) {
  %newVec = “0 0 “@%amount;
  return vectorAdd(%position, %newVec);
}
```

And just like that you could code up a simple vector function to add one vector to another. We can also use operations such as subtract to get the direction an object needs to travel to move towards another object:

```
%dir = vectorsub(%target.getPosition(), %source.getPosition());
```

This would only work for stationary objects; you’d need to do a little more in terms of math to calculate a proper direction in terms of this to consider the movement of the objects as well:

```
function getTargetingVector(%source, %target) {
  %vec = vectorsub(%target.getPosition(), %source.getPosition());
  %vec = vectorAdd(%vec, %target.getVelocity());
  return %vec;
}
```

This one’s a little more advanced, but it will be much more precise, and you can always tweak the factors to obtain the desired result. There is also built in capabilities for vectorScale, vectorDot, and vectorCross, for more advanced mathematical capabilities when dealing with vector operations, I would recommend at least going through some vector operation examples since most objects that will interact with things in your world will be using vectors in some way, shape, or form.

The next one is also almost unavoidable in computer programming, and that’s your lovely trigonometry problems that involve angles and determining them from different variations of information. Thankfully, the engine is loaded with all of your necessary trigonometric functions and methods to use: mSin, mCos, mTan, mASin, mACos, mATan, mRadToDeg, mDegToRad, and a few other useful tools to help you out along the way in terms of these mathematics. No examples this time around since most of your operations will be standard enough to complete with single lines of code or will be handled within functions that need no documentation. Just be sure to keep your trigonometric and angle rules handy when doing these however, you may run into some frustrating moments when math is telling you that your answer isn't possible and you try to blame it on the computer. And always be sure to perform the trigonometric functions in radian form (mDegToRad is very helpful here) or you'll get the result you don't want.

Lastly, you'll probably run across some geometry problems a few times along the way, mainly in the form of area checking or region testing (such as if a player is located in a certain zone), most of these can be handled by the engine itself, although don't be afraid to pull off the position and extent/area variables where needed to obtain necessary mathematical results for your functions.

So that covers the math side of game programming, the last thing you'll need to worry about in your game logic is dealing with the objects in your game directly. While most objects will spawn from a datablock (so they share the properties via %object.getDatablock()), there will be static objects as well. While I want to leave most of the important stuff to the error checking section below, I'll be brief regarding some object methods of importance here. One of the big and useful ones is the isObject() method, which allows you to test if an object parameter is an object currently in the world. Whenever you work directly with %object as a variable, the first thing you always want is that test (again, I'll cover below).

Spawning objects is very easy in Torque:

```
%myObject = new ObjectClass() {
   //Parameters Here…
};
SomeGroup.add(%myObject);  //usually someGroup is MissionGroup / MissionCleanup
```

You could be fancier about it, but this is the quickest and easiest way to go about it. Obviously the mathematics would play a role in objects that need to move or be placed in certain positions, all objects have a %object.getPosition() / %object.setPosition() method to use. And they even have a more advanced method called getTransform() and setTransform() where appended to the position is the Torque rotation coordinates. The only other important thing is to delete the object when done via %object.delete() and that should cover all you need to know regarding those.

11. Programming Strategies: Error Checking

Now I've got one last big and important topic to discuss regarding programming strategies for FPS games, and that's error checking. Error checking is the art of designing paths in your functions that point to stopping a function call when it would lead to an invalid result or a crash in the game/engine. This is one of those big rules in terms of programming and you should always incorporate it where necessary.

First off, let's talk about what causes most crashes in Torque. The big ones are "access violations", or where the program tries to read memory that is out of bounds. This will almost always happen whenever a function parameter that makes an internal engine call is blank or completely invalid. For example, deleting a non-existent variable will lead to a crash. So, let's say I have something along the lines of this:

```
$Important = new ScriptObject();
function killTheImportantOne() {
  $Important.delete();
  echo("Important One Slayed…");
}
```

Now, if I run the function the first time, everything is happy, and $Important dies. Now run it again… (This is actually fixed in T3D, but older Torque Games would crash) You'll get an error which states the object doesn't exist. This is the result of an internal error check, and you could easily do the same:

```
$Important = new ScriptObject();
function killTheImportantOne() {
  if(!isObject($Important)) {
    echo("Sorry, the Important One is dead…");
    return;
  }
  $Important.delete();
  echo("Important One Slayed…");
}
```

Now that's better. The other big thing you have to watch out for is division by zero, which will cause the numeric system to go haywire with NaN (Not a Number), which will cause any mathematical operation you've got in the function to basically explode.

```
function divideItNAO(%one, %two) {
  return %one / %two;
}
echo(divideItNAO(1, 0));
```

And there you have it, instant NaN generator, which is bad. Instead, add an error checking clause before the operation and break out of the function right at that point to prevent the invalid operation:

```
function divideItNAO(%one, %two) {
   if(%two == 0) {
      error("divideItNAO() – Cannot divide by zero.");
      return 0;
   }
   return %one / %two;
}
echo(divideItNAO(1, 0));
```

You'll notice I returned 0 at the end. This is done so if the function is used in mathematical operations, a division by zero will simply return 0 allowing the function to partially continue functioning. You usually want to avoid deadlocking a math function without a result. This would be very bad in the actual gameplay if that deadlock is triggered. Whenever you do operations in your gameplay code, you need to ensure that you have proper error checking.

Again, I refer you to the Torque3D documentation regarding the function list, however, a great tool to explore functions is %object.dump(); which will return a list of usable and codable functions for your objects.

General rule of thumb is to ensure every function has proper error testing to allow the code to run smoothly and without deadlocking concerns. Now, the last bit on coding I've got for this document regards external code.

12. Programming Strategies: Getting C++ into Torque

This is one of those less discussed topics, but there's quite a few easy ways to get your own custom C++ code into the engine. Adding "external" code allows you to perform costly operations a much quicker C++ environment. The easiest C++ -> Torque pipeline is by means of the C++ Macro Definition: DefineEngineFunction(), which adds a global function to TorqueScript. Here's an example for a basic printing method:

```
   DefineEngineFunction(sendAuthData, void, (),, "(Disabled)") {
      Con::printf("This executable is running on a non-auth based game, this
function has been disabled for that purpose.");
   }
```

Now, a brief explanation of the parameters of DefineEngineFunction. The first parameter is the actual name of the TorqueScript code you will be calling. The second piece is the return type (TS has: void, S32 (int), F32 (float), bool, and const char *). The third is the argument list enclosed in parenthesis. The fourth is the default value of arguments, again in parenthesis, and the final argument is the documentation of the function.

Here's a full sample method:

```
DefineEngineFunction(ConvertNumber, const char *, (const char * input, const
char * newBase),, "converts a number") {
   int number = atoi(input);
   char * final = (char *)malloc(strlen(input));
   _itoa(number, final, atoi(newBase));

   return (const char *)final;
}
```

When working with string results (which a good deal of coding will be about in TorqueScript) you'll want to make use of the Return Buffer:

```
DefineEngineFunction(getAuthData, const char *, (),, "(void)") {
   String output;
   output = cryptoPackage->getAccountData();

   char *str = Con::getReturnBuffer(output.size() +1);
   dStrcpy(str, output.c_str());

   return str;
}
```

Here's an example piece from my MAP pack which returns a string containing the player's account information locally. Now let's briefly discuss how to code object specific C++ methods, ie: %object.code();

Essentially the process is the same, however the macro is slightly different, and you've got access to a unique local variable for the macro codeblock. Here's a quick sample piece:

```
DefineEngineMethod( GameConnection, setFirstPerson, void, (bool
firstPerson),,
   "@brief On the server, sets this connection into or out of first person
mode.\n\n"

   "@param firstPerson Set to true to put the connection into first person
mode.\n\n")
{
   object->setFirstPerson(firstPerson);
}
```

Now some documentation. The first change is that the macro is DefineEngineMethod instead of DefineEngineFunction. Now for the parameters, you've got six instead of five and the only change is that we're inserting the Class name of the object in question at the front of the list before the method name.

The special thing regarding these is a direct reference to the object variable when called: IE: for the above: GameConnection::setFirstPerson(%this, %bool) is the TorqueScript equivalent of the function. You should see right away that the TS has the extra (%this) variable. In C++ under DefineEngineMethod, %this is the object in question, and you can use object as %this. Object being a pointer reference of the class (first variable).

And that's all you need to know about adding custom methods to the engine. From here you can easily handle the big game tasks using customized engine code and simply call in necessary function "hooks" to TorqueScript using the relevant calls here!

13. Concluding Notes

And that's all I've got for now. I hope you've found this document enlightening regarding why it's important to "stick with a plan" per say, and how to design a proper plan when it comes to making shooter games. Hopefully, some of the methodology I've provided here can help push someone in the right direction, or maybe if I'm lucky spawn the game that finally kills Call of Duty. Who knows, maybe I'll be the one to do it, but in all honesty, just following the six rules should put you in a position to do more than just that in reaching grand success! ☺

Thanks again for grabbing this tutorial and giving it a read, and please, I want feedback on my forums!!! Don't be a document leecher, I spent days writing this thing, even when I was sick…. >:C