Mod Authoring for Unreal Tournament

Brandon Reinhart
Epic Games, Inc.
brandon@epicgames.com
http://unreal.epicgames.com/

Last Updated: 12/08/99

Contents

Introduction

> "Winds and storms, embrace us now,
> Lay waste the light of day,
> Open gates to darker lands
> We spread our wings and fly away."
> - Emperor, Alsvartr (The Oath)

Writing mods for the Unreal engine can be an extremely rewarding task. In the current state of the games industry, there is little better a way for a skilled programmer to show the world what he is capable of. This document is intended to give interested mod authors the information they need to write a successful mod for the Unreal engine. I will include technical information as well as pointers on mod development. If you work hard, you can get some fun stuff done through mod authoring.

The Lazy Man's Guide

So you want to get into UnrealScript hacking NOW NOW NOW. This is what you should do:

1. Export UnrealScript source files. Open UnrealED, set the Browser bar to Classes. Hit Export All. This dumps the UnrealScript source files to disk instantly creating your own SDK.

2. Create a package directory and hack some code. Like this:

```
mkdir MyPackage
cd MyPackage
mkdir Classes
cd Classes
edit MyScriptFile.uc
```

3. Add your package to the EditPackages list in UnrealTournament.ini

4. Build your files with ucc. First delete your "MyPackage.u" if it exists. **ucc make only rebuilds missing code packages.** Then type "ucc make" from the System directory of your Unreal Tournamnt install.

Some Advice to Start With

When you begin writing a mod you should start small. Don't plan to write a Total Conversion (TC) from the very start. If you set goals that are too hard to reach, you'll get frustrated in the process of working towards them. It is much better to set a series of small goals and work to each one in turn. Start with a simple idea that could be expanded into a larger game. Always work on small, managable chunks that could each be released in their own right. If you do undertake a large project, organize your features into a release schedule. If your game is going to have 5 new weapons, make a release with 3 while your work on the others. Pace yourself and think about the long term.

Everyone is an idea man. Everybody thinks they have a revolutionary new game concept that no one else has ever thought of. Having cool ideas will rarely get you anywhere in the games industry. You have to be able to implement your ideas or provide some useful skill. This also applies to mod authoring. If you become a skilled or notable mod author, you will find people propositioning you to implement their ideas. Never join a project whose idea man or leader has no obvious development skills. Never join a project that only has a web designer. You have your own ideas. Focus on them carefully and in small chunks and you will be able to develop cool projects.

Remember that developing a mod doesn't mean much if you never release it. Scale your task list so that you can release something quickly, adding and improving

features as your mod matures. If you hold back your mod until everything is perfect, you'll find yourself never releasing anything.

Now that you have your idea, you need to choose what kind of Unreal Tournament mod type is right for you. There are basically three types of mods. We'll go over each one in brief and then focus on them all individually.

The Three Mod Types

Mutators

Mutators are mini-mods. They have limited functionality as defined by the Mutator class. Mutators should follow certain rules. If you can't follow these rules, you should probably work on a **GameType** mod.

The first rule is that Mutators should be able to work with any other Mutator. If you write a "Vampire" mutator that allows the player to drain life from an enemy he shoots, the mutator should work well if combined with one of the Arena mutators or the No Powerups mutator. This is one of the beneficial features of the Mutator system. They slightly change (or mutate) gameplay, allowing for interesting combinations.

The second rule is that Mutators should only change gameplay in a slight fashion. Although that's a vague way of putting it, you need to try and restrict your Mutator behavior. Careful mutator design will increase the chances of your mutator working with other mods and will decrease your support effort.

The third rule is that Mutators should share resources with other Mutators. If your Mutator implements the "ModifyPlayer" function, you need to call "NextMutator.ModifyPlayer" somewhere inside your version of the function. This ensures that any Mutator on the Mutator list after your mod gets a chance to deal with the function call. Failing to do this is poor programming style.

GameTypes

GameTypes are a much larger class of mod. They do everything the Mutator can't and allow you access to a much larger range of functionality. If your idea can't be implemented within a Mutator, you should work on a GameType.

The drawback of a GameType is that it cannot be mixed with other GameTypes. For example, Capture the Flag is a GameType in Unreal Tournament. It is a wholly different style of gameplay from Assault (another GameType).

GameTypes are implemented as subclasses of the "TournamentGameInfo" class. There aren't any specific rules for GameTypes, other than some client-server issues that you should be aware of (and that we will discuss later).

Everything Else

It is possible to write a mod that doesn't change gameplay through the GameInfo or Mutator classes. These would include Player Plugin Models (PPM) or a new weapon. We'll talk about a few prime examples like Weapons and Pickups later. GameTypes will often include many new weapons, pickups, AI features, or special actors that are separate from the game rules themselves.

A Few Things to Watch Out For

This is where I'm going to put all the information that I wish someone had told me when I started writing mods for Unreal. A lot of this information may not be relevant to you until you have more experience with the engine. I spent a lot of time out on my front porch with a buddy (Sel Tremble) talking about things like replication trying to figure out exactly how it all worked. That was definitely one of the most satisfying things I have ever done. Cracking open a new game can be a very cool experience, but also a very frustrating one. Here I'll give you a couple pointers to ease your exploration.

Accessed Nones

Sooner or later these will start showing up in your log files. UnrealScript treats Accessed Nones as warnings but you should treat them as errors. Accessed Nones are easy to fix and always signal that something is wrong with your code. If you are familiar with C++ or Java, it's easy to figure out what an Accessed None is. I'll briefly explain them to people who aren't so familiar.

UnrealScript is an object oriented programming language. When you write a program in UnrealScript, you define a set of behavior for these objects to obey and how they will interact. An object has a set of properties: member variables and member functions. In order to access an object property, you need a reference to that object. Let's look at some sample code:

```
class MyObject expands Info;

var PlayerReplicationInfo PlayerInfo;

function PlayerReplicationInfo TestFunction()
{
        return PlayerInfo;
}
```

Here we have a simple object called "MyObject" that is a subclass of Info. It has two properties: a variable called PlayerInfo and a function called TestFunction. You might want to interact with this object from inside your mod. Let's say you have a reference to a MyObject inside your mod and you want to get some information from inside the PlayerInfo property. You might write code that looks like this:

```
class MyMod expands TournamentGameInfo;

function string GetPlayerName()
{
        local MyObject Object1;
        local string PlayerName;

        Object1 = GetMyObject();
```

```
        PlayerName = Object1.PlayerInfo.PlayerName;
        Log("The player's name is"@PlayerName);
}
```

In this example we call a function called GetMyObject() to get a reference to a MyObject. We then access that reference to resolve PlayerInfo ("**Object1.PlayerInfo**") and then access the PlayerInfo reference to resolve PlayerName ("**PlayerInfo.PlayerName**"). But what if there isn't a MyObject available, or a bug in GetMyObject() causes it to fail to return a MyObject? In that case, the function would return "None." None is an empty reference...a lot like a NULL pointer in C++.

If, in our example, GetMyObject() returns None, then the variable Object1 is assigned None. In the next line, we try and access Object1 to resolve the PlayerInfo reference. Uh oh....Object1 is None...it doesn't refer to anything. We can't access it, so the Unreal engine logs a warning saying the code broke: Accessed None in MyMod.GetPlayerName!

Its very easy to avoid buggy code like this. Just add a couple checks to your code and define special behavior in the case of a mistake:

```
class MyMod expands TournamentGameInfo;

function string GetPlayerName()
{
        local MyObject Object1;
        local string PlayerName;

        Object1 = GetMyObject();
        if ((Object1 != None) && (Object1.PlayerInfo != None))
                PlayerName = Object1.PlayerInfo.PlayerName;
        else
                PlayerName = "Unknown";
        Log("The player's name is"@PlayerName);
}
```

Now we are checking to see if Object1 is none and then checking to see if the PlayerInfo reference is none. "If" statements in UnrealScript use short circuit logic. That is, "If" statements are evaluated from left to right. As soon as the code encounters a statement that negates the "If", it stops evaluating. That means that if Object1 is None, the code will never evaluate the (Object1.PlayerInfo != None) statement. It knows that it doesn't matter what the rest of the statement says, the first part is false so the entire statement is false.

Accessed Nones can be especially dangerous in time critical functions like Timer and Tick. It takes a lot of time to write out an error message to the log and if your code is dumping 3000 error messages a second it can really kill performance (not to mention disk space).

Dangerous Iterators

UnrealScript implements a very useful programming tool called Iterators. An iterator is a datatype that encapsulates a list. (UnrealScript only supports list iterators, our next language will support user defined iterators). You can get an iterator and loop on it, performing an operation on every object inside the iterator. Here is an example:

```
local Ammo A;
foreach AllActors(class'Ammo', A)
{
        A.AmmoAmount = 999;
}
```

In this example we are using the AllActors function to get an Actor List iterator. We then use the foreach iterator loop to perform some behavior on every object the AllActors function returns. AllActors takes the class of the type of actor you want and a variable to put it in. AllActors will search through **every actor in the current game** for the objects you want. Here we are saying "set the AmmoAmount of every actor in the game to 999."

Sounds pretty cool, but lets think about it. We are searching through a list of hundreds of Actors for a small few. This isn't exactly a fast operation.

Iterators can be extremely useful if used carefully. Because they tend to be slow, you'll want to avoid performing iterations faster than a couple times a second. Never perform an AllActors iteration inside of Tick() and never perform AllActors iterations inside of other loops. (Okay, so saying NEVER is a little strict. Let's say...USE YOUR BEST JUDGEMENT...)

The most common type of AllActors search you'll work with will probably be a search for all of the PlayerReplicationInfo actors. PlayerReplicationInfo contains important information about Players that the server sends to each client. It allows each client to have an idea of the status of other playes without sending too much information. Its used to show the scores on the scoreboard and other common things.

Usually, there will only be a handful of PlayerReplicationInfo actors in the global Actor List. It doesn't really make sense to do a time consuming search for so few results. In order to simplify this common iteration, we've added a PRI array to GameReplicationInfo. Every tenth of a second, the PRIArray is updated to contain the current set of PlayerReplicationInfos. You can then do your operation of the PRIArray without having to do an AllActors call.

Other iterators are also available. Look in the Actor class definition for information. They do exactly what they sound like: TouchingActors returns touching actors, RadiusActors returns all the actors in the given radius, etc. Intelligent use of these iterators will help you keep your code fast.

The Foibles of Tracing

Wahaha. I just wanted to use the word foible.

Because the Unreal engine does not use a potentially visible set, if you want to find something in the world in a spacial sense, you'll need to perform a trace. Most of the time you'll have a good idea of where you are tracing, you just want to know whats on the other end of the line. Other times, you'll use a series of traces to get an idea of what surrounds the object in question.

My first advice is to avoid traces wherever possible. Think very hard about what you are using the trace for and try to come up with an alternate way of doing it. Traces are expensive operations that can introduce subtle slowdowns into your mod. You might have a player doing a couple traces every tick and during your testing everything is fine. What you don't realize, is that as soon as you are playing online with 15 of your buddies, those traces start to add up.

If you have to perform traces, limit their size. Shorter traces are faster than long traces. If you are designing a new Shotgun weapon for UT, for example, you might want to perform 12 traces when the weapon is fired to figure out the scatter of the gun. 12 traces is perfectly reasonable.... it's not like the player is going to be firing his shotgun 30 times a second. However, those 12 traces could get expensive if your mod uses large open levels. Its highly unlikely your shotgun is going to be very useful as a long-range weapon, so you might as well cut off its range at a certain point. It saves the engine from having to trace from one end of the map to the other in the worst case.

Using traces is ultimately a judgment call. It really only becomes a big problem when you perform a lot of traces in a single frame. Nonetheless, it's definitely something to keep your eyes on. **Always think about the performance implications of code you write.**

Decrypting Replication

Understanding replication is one of the most difficult aspects of writing a mod, but its utterly necessary if you plan on doing any netplay at all. Unfortunately, Tim's replication docs are not easy to understand and make some assumptions about the reader's knowledge that you may not possess. I'll try to point out the things that I learned only through trial and error.

Simulated functions are called on both the client and the server. **But only if called from a simulated function.** As soon as a function call breaks the simulation chain, the simulation stops. Be very aware of what you are simulating and what you are doing in simulated functions. **Never add a function modifier like simulated just because you saw it in the Unreal source code somewhere else.** Understand why you are adding it, know what it does. You can't possibly expect to write quality mods if you don't know what your code is doing.

Because a simulated function is called on both the client and the server you have to be particularly aware of what data you are accessing. Some object references that are available on the server might not be available on the client. For example, every Actor has a reference to the current level. Inside the level reference is a reference to the current game. You might write code that looks like this:

```
simulated function bool CheckTeamGame()
{
        return Level.Game.bTeamGame
}
```

This is a simple simulated function that returns true or false depending on whether or not the current game is a Team Game. It does this by checking the bTeamGame property of the current level's GameInfo reference. What's wrong with this picture?

The Game property of the Level reference is only valid on the server. The client doesn't know anything about the server's game object so the client will log an Accessed None. Yuck!

If you open up the script for LevelInfo, you can find a section that looks like this:

```
//-----------------------------------------------------------------------------
// Network replication.

replication
{
        reliable if( Role==ROLE_Authority )
                Pauser, TimeDilation, bNoCheating, bAllowFOV;
}
```

The replication block is a special statement that tells the Unreal engine how to deal with the properties of this object. Lets look at it closely.

First, we have a replication condition: **reliable if(Role == ROLE_Authority)**. The first part of the condition will either be reliable or unreliable. If it says reliable, that means the engine will make sure the replicated information gets to each client safely. Because of the way the UDP protocol works, its possible for packets to get lost in transmission. Unreliable replication won't check to see if the packet arrived safely. Reliable replication has a slightly higher network overhead than unreliable replication.

The second part of the condition (Role == ROLE_Authority) tells the engine when to send the data. In this situation we are going to send the data whenever the current LevelInfo object is an Authority. To really decypher what this means you have to understand the specific role of the object in question. With a LevelInfo, the server is going to maintain the authoritative version of the object. The server tells the clients how the level is behaving, not the other way around. For our example replication block, this means that the data will be sent from the server to each client.

The other common type of condition is (Role < ROLE_Authority). This means that the engine should send the data when the current object is not an authority. Or rather, that the client should tell the server the correct information.

Finally, we see four variables listed beneath the condition. These are the variables that the statement applies to. In this situation, we have a statement saying, "If we are the server and the client has an outdated copy of these variables, then send to the client new information about Pauser, TimeDilation, bNoCheating, and bAllowFOV. Always make sure the data arrives safely."

The replication statement doesn't cover the rest of the variables in the LevelInfo. This can mean two things. Either the information is filled in by the client in C++ (in the case of TimeSeconds) or the information is never updated on the client and is completely unreliable (in the case of Game).

You don't have access to the C++ code, but you can make a couple inferences about an object's properties to help you determine whether or not a class has non-replicated properties that are filled in my C++. Look at the class declaration for LevelInfo:

```
class LevelInfo extends ZoneInfo
  native;
```

Native means "This object is declared in C++ and in UnrealScript." Native classes probably have behavior in C++ that you can't see. Only a few special classes are

native.

Finally, watch out for classes that say "nativereplication" in the class declaration. This means that the "replication" block inside UnrealScript doesn't do anything and that replication is entirely defined in C++. Some network heavy objects use native replication to help with network performance.

So now you have an idea of how to avoid problems with simulated functions. Now lets look at replicated functions.

A replicated function is a function that is called from the client or the server but executed on the other side. An example of a replicated function is the "Say" function. When you hit the T key to talk to everyone in a game, you are actually executing the Say function along with whatever you said. The client takes the function and its parameters and sends it to the server for execution. The server then broadcasts your message to all the other clients.

Replicated functions are very easy to use if you remember one thing: they can't return a value. A replicated function is transmitted over the network to the other side...that takes time (approximately equal to your ping). If replicated functions were blocking (i.e.: they waited for a return value) network communication would halt.

This is obvious for anyone who thinks about it, but when you are working on your mod you might not think about it. Replicated functions return immediately. Use them to trigger behavior on the client (like special effects) or send a message (a weapon fire message to the server).

Finally, replicated functions are restricted to only a few classes. A function call on an actor can only be replicated to the player who owns that actor. A function call can only be replicated to one actor (the player who owns it); they cannot be multicast. You might use them with weapons or inventory items you make (where the function is replicated to the player who owns the item).

Okay, so that should help you get into replication....let's move on.

Don't use UnrealEd

UnrealEd is a great editor for developing levels, but probably not the best place to work on code. This is a judgment call. I use Microsoft Developer Studio as my editor and "ucc make" to compile the package files. I find the Find In Files option in Dev Studio to be very useful and the editor to be very powerful.

In addition, UnrealEd hides the default properties blocks of source files, making them only accessible through the Show Defaults option. This just sucks! To export the script files to disk, go to the script browser and hit the "Export All" button. The files will be exported to their package directories ready for you to browse.

If UnrealEd crashes with a DLL or OCX error of some sort, go to unreal.epicgames.com and click on Downloads. Download the latest Unreal Editor Fix. The current fix level is 4.

Getting Dirty: Setting Up Your Project.

Now its time to set up Unreal Tournament to build your project. First things first, you need to understand how UnrealScript uses packages.

Packages are collections of game resources. The resources can be anything, textures, sounds, music, or compiled game code. The package format is the same for all resources and multiple resource types can be mixed in a package. For the sake of sanity, Unreal Tournament splits up packages into resources. The textures directory contains packages with textures, the sounds directory contains packages with sounds and so forth. Even though these packages have different suffixes (.utx, .uax, etc) they are still the same kind of file.

You are going to be dealing with .u files, or code packages. Code packages primarily contain compiled UnrealScript, but may also contain textures and sounds that the code depends on.

UnrealScript is an object oriented language. If you aren't familiar with OO, now is a good time to take a detour and read my guide to object oriented programming. Here is the link: http://www.orangesmoothie.org/tuts/GM-OOtutorial.html. This document is fairly old, but still a good resource.

Since UnrealScript (lets call it US for short) is object oriented, you won't be editing any of the original source. This is different from Quake, where you edit the original source and then distribute a new DLL. In US, you will subclass the classes that shipped with Unreal Tournament, modifying them to suit your needs.

So lets make a package. I'm going to refer to our test package as "MyPackage" but you will want to call it the name of your mod. Where I say "MyPackage" you'll want to replace with your own package name. Make a directory in your Unreal Tournament directory called MyPackage. Underneath this directory, make a directory called Classes. The UnrealScript compiler looks in the Classes directory for source files to build.

Now, edit UnrealTournament.ini and search for EditPackages=. You'll see a list of all the Unreal Tournament packages. Add your package to the list:

```
EditPackages=Core
EditPackages=Engine
EditPackages=Editor
EditPackages=UWindow
EditPackages=Fire
EditPackages=IpDrv
EditPackages=UWeb
EditPackages=UBrowser
EditPackages=UnrealShare
EditPackages=UnrealI
EditPackages=UMenu
EditPackages=IpServer
EditPackages=Botpack
EditPackages=UTServerAdmin
EditPackages=UTMenu
EditPackages=UTBrowser
EditPackages=MyPackage
```

Lets take a break and figure out what all those packages are for!

Core contains fundamental unrealscript classes. You won't need to look at the stuff in here much at all. Notice that Core, like many .u files, has a related DLL. The DLL contains the C++ part of the package.

Engine is where things get interesting. You'll soon become very familiar with the classes in engine. It contains the core definitions of many classes that will be central to your mod. GameInfo describes basic game rules. PlayerPawn describes basic player behavior. Actor describes the basic behavior of UnrealScript objects.

Editor contains classes relevant to the editor. You'll never need to mess with this, unless you become a totally elite hacker.

UWindow contains the basic classes relevant to the Unreal Tournament windowing system. This is a good place to research how the system works if you want to add complex windows and menus to your mod.

Fire contains the UnrealScript interface to the "Fire Engine." The fire engine is the code that makes all the cool water and fire effects in UT.

IpDrv contains classes for putting a UDP or TCP interface into your mod. We use this for the IRC interface in the game, among other things.

UWeb contains classes for remote web administration.

UBrowser contains the core classes for the in game server browser.

UnrealShare contains all the code from the shareware version of Unreal. Nalis galore!

UnrealI contains all the code from the full version of Unreal. UnrealShare and UnrealI are included in UT because some UT code is based on classes in these packages. There is a LOT of content here you could use for your mod.

UMenu contains any menus for UT that don't depend on Botpack.

IpServer contains the GameSpy querying interface.

Botpack the soul of the new machine. Contains all of the game logic for Unreal Tournament. Tons of kick ass toys to play with. This is where a lot of your research time will be spent.

UTServerAdmin contains Unreal Tournamnt specific web admin code.

UTMenu contains UT menus that require content from Botpack.

UTBrowser contains browser code that requires content from Botpack.

Notice that the order matters here. This is the order in which the compiler will load the packages. "TournamentGameInfo" in Botpack is a GameInfo, so in order for the compiler to build that code, it needs to have Engine loaded. Your mod should go at the end of the list to benefit from all the code in the previous packages.

How You Build Your Package

Now that your package is set up, you are ready to build it. You don't have any code written yet, so lets make a very simple useless mod. This will serve as a good example of making a new GameType.

In the MyPackage/Classes directory, create a file called MyGame.uc. Put the following code inside of it:

```
class MyGame expands DeathMatchPlus;

defaultproperties
{
        GameName="My Game"
}
```

This creates a class called "MyGame" that is a subclass of DeathMatchPlus. All we are doing is changing the name of the game...pretty simple. Notice the inheritance in action. All the code that makes up DeathMatchPlus is now a part of your game type.

Save the file and change directory to System. Type 'ucc make' and watch the code burn. Pretty soon, your package will be compiled and a new MyPackage.u will be sitting in the System directory.

In order to make our new gametype show up in the menus, we have to give it a metaclass definition. Create a file in the System directory called MyPackage.int. INT files primarily contain localization information for foreign versions of the game, but they also contain extra information about the classes inside a package.

Add the following lines to your int file:

```
[Public]
Object=(Name=MyPackage.MyGame,Class=Class,MetaClass=Botpack.TournamentGameInfo)
```

Save it and exit. When UT's menus search for games to list in the Game selection window, they search all the .int files in the System directory for classes that have a MetaClass of Botapck.TournamentGameInfo. Now your game will show up on that list. Start Unreal Tournament and go to Start Practice Session. Find your game on the list. If you start it, "My Game" will show up as the game's name in the scoreboard. Cool!

So that's how you build a basic project. Obviously writing a mod is a lot more complicated than that. Now we'll get down and REALLY dirty by looking at the specifics of different types of mod.

Making a Mutator

Mutators are a great place to cut your teeth on UnrealScript because you are exposed to a limited, but powerful subset of the engine. As I said above, Mutators should only modify the game code in a relatively slight way. This increases the chances your mutator will work well when mixed with other mutators. (For example, you can play FatBoy, InstaGib, No Powerups deathmatch. A mix of 3 mutators).

All mutators descend from the Mutator base class either directly or indirectly. Let's make a Vampire mutator and see how it all works. Create a new file in your package classes directory called Vampire.uc. Drop the following code in there:

```
class Vampire expands Mutator;

var bool Initialized;

function PostBeginPlay()
{
        if (Initialized)
                return;
        Initialized = True;

        Level.Game.RegisterDamageMutator( Self );
}

function MutatorTakeDamage( out int ActualDamage, Pawn Victim, Pawn InstigatedBy, out Vector HitLocation,
        out Vector Momentum, name DamageType)
{
        if (InstigatedBy.IsA('Bot') || InstigatedBy.IsA('PlayerPawn'))
        {
                InstigatedBy.Health += ActualDamage;
                if (InstigatedBy.Health > 199)
                        InstigatedBy.Health = 199;
        }
        if ( NextDamageMutator != None )
                NextDamageMutator.MutatorTakeDamage( ActualDamage, Victim, InstigatedBy, HitLocation, Momentum, DamageType );
}
```

The first line declares the class contained in this file. US is like Java in that each file contains a separate class definition. We are saying that our class, Vampire, is a Mutator. It "inherits" all the properties of Mutator.

Next, we declare a class member. In UnrealScript, all class member variables must be declared before any functions (also called methods) are declared. The var keyword tells the compiler what we are doing. Here we have a Boolean (true/false) value called Intialized.

Next we have a function called **PostBeginPlay**. To someone who isn't experienced with object oriented programming, we have a bit of a puzzle. This object just declares functions, it doesn't seem to have any entry point! The "entry point" is inherited. Vampire is a Mutator, so it does everything Mutators can. Mutator is an Info, Info is an Actor, and an Actor is managed by the engine. In our case, we are overriding the function PostBeginPlay. PreBeginPlay, BeginPlay, and PostBeginPlay are called on every Actor in the level when the game starts. They are used to initialize the world.

Our PostBeginPlay function starts by setting initialized to true. Notice it'll return if initialized is already true. What's the point of that? Well, the problem is that the BeginPlay suite of functions get called twice on Mutators. Unfortunately, this is somewhat unavoidable. BeginPlay and its friends are called on Actors when they are spawned and when the game starts. Mutators, unlike other actors, are spawned before the game starts...so you get two calls. The Initialized check is to prevent the rest of PostBeginPlay from being executed twice.

The second part of PostBeginPlay is called RegisterDamageMutator, a method located in the GameInfo class. Here we are accessing the "Level" property that all Actors inherit. Then we access the "Game" property of the LevelInfo class that Level points to. Finally, we call the function from that reference, passing our "Self" as the parameter.

RegisterDamageMutator is a special method that tells the GameInfo to call MutatorTakeDamage on this mutator whenever a pawn takes damage. Because pawns take a lot of damage during the course of a normal game, we don't want to call this function on every mutator, that would be slow. RegisterDamageMutator allows us to limit the calls to only a subset of mutators. (By the way, in UT pawns consist primarily of Bots and PlayerPawns).

Next we have our implementation of **MutatorTakeDamage**. This is the heart of our Mutator. We are making a Vampire mutator. The idea is simple: if a Pawn A does damage to another Pawn B, give the Pawn A health equal to the amount of damage. Players and bots will effectively suck life off of other players or bots.

As I mentioned above, RegisterDamageMutator is called on our mutator whenever a player takes damage from another one. The pawns in question are passed to us in the variables.

We start by making sure we are only dealing with bots and playerpawns. There are pawns that are neither bots nor players and we don't want to deal with them. InstigatedBy refers to the player who dealt the damage. We do our core logic by adding to that pawn's health life equal to the damage dealt. RegisterDamageMutator always passes in an amount of damage AFTER armor, so this is raw final damage. Finally, we don't want a player gaining so much life he becomes unkillable, so we limit the total life gain to 199 points.

To finish the function off, we call RegisterDamageMutator on the next damage mutator in the list. It is **critical** that you pass along method calls like this. If you fail to, damage mutators loaded after your own won't work right. There are other functions that need to be passed along, which we'll look at below.

So now you can save this file and rebuild your package. We aren't done yet, though, because the mutator won't show up in the menus without some more work. Open your package's int file and add the following line to the [Public] section:

```
Object=(Name=SemperFi.Vampire,Class=Class,MetaClass=Engine.Mutator,Description="Vampire,You gain life equal to the amount of d
```

When the game looks for mutators to list in the Add Mutators window, it searches all .int files for objects with a declared MetaClass of Engine.Mutator. We've also added a Description for the help bar. Now we are ready to run Unreal Tournament and load the mutator. Play around with it for a bit and you'll probably get ideas for

your own mutators or ways of expanding Vampire.

The Anatomy of Mutator

So now you've had your first exposure to writing a simple UT mod. Clearly this isn't enough to shake the world or get a job in the industry. Let's take a close look at the methods inside the Mutator base class. This will give you a better idea of what you can do with them. It only scratches the surface, however, because you have the power to use a multitude of inherited functions as well as interact with other objects.

We'll skip the PostRender function for now and look at ModifyPlayer. This is called by the game whenever a pawn is respawned. It gives you a chance to modify the pawn's variables or perform some game logic on the pawn. Remember to call Super.ModifyPlayer() if you override this function. That will call the parent class' version of the function.

ScoreKill is called whenever a pawn kills another pawn. This lets you influence the score rules of the game, preventing point gains in certain situations or awarding more points in others. Remember to call Super.ScoreKill() if you override this function.

MutatedDefaultWeapon gives you an opportunity to give a different default weapon to a player that enters a game or respawns. In UT, the default weapon is the Enforcer. If you just want to change the default weapon, you don't need to override this function. Instead, just add a DefaultWeapon definition to the defaultproperties of your mutator. (See the bottom of PulseArena for an example).

You don't need to mess with MyDefaultWeapon or AddMutator.

ReplaceWith and AlwaysKeep allow you to interdict objects that the game wants to add to the world. You can replace objects on the fly with other objects as they appear. The Botpack.Arena mutators are a great example of this. They take all the weapons in a game and replace them with one other weapon. If you are adding a new weapon to the game, you might want to add an Arena mutator for it.

IsRelevant is called when the game wants to add an object to the world. You can override it with special code and return true, to keep the object, or false, to reject it. If you say false, the object will be destroyed.

Mutate is cool. It lets your mutator define new commands that player's can bind to keys. If a player binds "mutate givehealth" to a key and then uses that key, every mutator will get a mutate call with a "givehealth" parameter. Your mutator might look for this string and give the player who sent the message some extra health.

MutatorTakeDamage, as described above, is called on DamageMutators. It lets damage mutators do some kind of game logic based on a pawn taking damage. It also tells you where the pawn was hit, the type of damage, and how much force the damage imparted.

RegisterHUDMutator is used to tell the game that this mutator should get PostRender calls. PostRender passes you a Canvas every frame. You can use the Canvas to draw stuff on the HUD. Look in the Botpack.ChallengeHUD class for extensive examples of abusing the Canvas. Hehe.

The best way to learn mutators is to read the code in the mutators that ship with UT. In fact, you'll probably want to spend a lot of time just pooring over the massive amount of code that comes with the game. Trace execution paths and look at how the various classes override and interact with each other. It can be **very intimidating** at first, but with time you'll get more experienced with where things are and what things do. Don't be afraid to go online and ask questions either or read other mod authors code. If you spend the time it takes to learn, you will be rewarded with the ability to take on larger, more difficult projects.

Introduction to GameTypes

Where to start, where to start? This is the meat. The big bone. Now we start getting into the hard stuff. Mutators can do some cool stuff. They are pretty easy to understand and they can do a lot of things by interacting with the game. They can be mixed and matched to get even cooler effects...but they are NOT very powerful. If you want to make a new type of game (say a Jailbreak style mod) you can't do it with mutators. You need to have complete control over the game rules. That's where the GameInfo series of classes come into play.

GameInfo is a class located in Engine. It is created by the game engine and is the core of the game play rules. Unreal Tournament makes use of a series of GameInfo subclasses located in the Botpack package. TournamentGameInfo contains code that is universal to all of Unreal Tournament's game types. DeathMatchPlus contains the code for running a normal death match. TeamGamePlus contains code for team deathmatch as well as general team management code. Domination and Assault, which are subclasses of TeamGamePlus, implement those particular game types.

The first step in writing your new game type is to determine which class to subclass. If you are writing a team game, you'll want to subclass TeamGamePlus. If you are writing a game without teams, use DeathMatchPlus. If you are writing a game that departs significantly from any previously styled game type, use TournamentGameInfo. Subclassing is very beneficial...you immediately inherit all of the code in your parent classes.

Lets look at a very simple Game Type mod:

```
class MyGame expands DeathMatchPlus;

defaultproperties
{
   GameName="My Game"
}
```

The above code, when saved in a file called "MyGame.uc" will build a new game type. The only difference here is that we've changed the name to "My Game." This new name will be reflected in many places: the Practice Session selection window, the Scoreboard header, and so forth. If you play this game, it'll play just like DeathMatchPlus...we haven't actually added any new behavior.

Like Mutators, we need to do a little INT file hacking in order to get the new game type to show up in the menus. Edit your package's INT file and add the following lines to the [Public] section:

```
Object=(Name=MyPackage.MyGame,Class=Class,MetaClass=Botpack.TournamentGameInfo)
Preferences=(Caption="My Game",Parent="Game Types",Class=MyPackage.MyGame,Immediate=True)
```

The practice session and start server menus look in all INT files for declared objects that have a MetaClass of Botpack.TournamentGameInfo. If you add these line to your package's INT file, you'll get an entry called "My Game" in the list of games. The name is taken from the GameName variable of your GameInfo class. The Preferences line gives your game a configuration entry in the Advanced Options menu. You probably don't need to worry about that right now.

So now we have a simple game to start messing with. What do we do? Well lets look at a few of the methods available in GameInfo. Remember, you'll need to do a lot of research on your own. You'll only become a strong UnrealScript hacker if you spend time to acquaint yourself with the code at your fingertips.

A First Look at GameInfo

Open the Engine package file "GameInfo.uc." This is the definition of the basic game logic. At the top of the file you'll see a long list of variable declarations. Many of these variables have comments that describe their purpose. Below the variable declarations come the functions (or methods) that do the work.

The first thing to look at is the Timer function. In GameInfo its pretty short, but in DeathMatchPlus its very long. Timer is a special UnrealScript event. If you call the function SetTimer(int Time, bool bLoop) you can set up a repeating timer on your actor. The Time parameter describes when the Timer function should be called. The bLoop parameter describes whether or not Timer should be called in a loop after the first call. **All TournamentGameInfo classes use a Timer loop of one second.** This means that the Timer function is called every second. You can use Timer for events that have to happen at certain times. By declaring watch variables that count up seconds, you can perform events at any time up to a second's resolution. DeathMatchPlus uses this to check and see if the TimeLimit has been hit in a game.

Another important time function to get to know is Tick. Tick isn't used in GameInfo, but any Actor can use it. The declaration for Tick is: Tick(float DeltaTime). Tick is called on **every** Actor in the game each frame. DeltaTime contains the amount of time that has passed since the last Tick. Using Tick, you can perform behavior that has to be done at less-than-a-second resolution. You must be careful not to perform CPU heavy behavior in Tick, because it is called so often.

Scroll down in GameInfo until you find the Login function. This function is called by the engine whenever a player logs in to the game. GameInfo's version of login does important setup stuff like assigning the player a name, a skin, a mesh and so forth. It also spawns the intial teleport effect and finds a spawn point to stick the player at. A little ways below Login is Logout. It is called whenever a player leaves the game. You can use logout to clean up after a player exits.

Another interesting function in GameInfo is AddDefaultInventory. This function assigns a player his initial weapon and equipment. In UnrealTournment's DeathMatchPlus, the player is given an ImpactHammer and an Enforcer. LastManStanding has a great example of doing cool things with AddDefaultInventory. It gives the player every weapon in the game (except for the Redeemer) as well as some armor and a lot of ammo. You can use AddDefaultInventory to add custom inventory to players that join your mod (for example, you might want to give them a grenade and some money).

The FindPlayerStart method searches the actors in a level for NavigationPoints suitable for spawning. The "PlayerStart" actor that a map designer adds to their map is one such location. In TeamGamePlus, FindPlayerStart spawns players and bots depending on their Team. It checks the Team of each playerstart and the Team of the pawn to be spawned. You can use FindPlayerStart to write custom spawn code (for example, you might want to spawn Terrorists in one location and Snipers in another).

The RestartPlayer method is called whenever a player respawns. The basic GameInfo version calls FindPlayerStart to find a starting spot, moves the player to that spot and spawns a teleport effect. It also restores the players health, sets the player's collision, and gives the player his default inventory.

The Killed method is very useful. It is called whenever a player kills another player. It looks at the cirumstances of the death (whether a player suicided or killed successfully) and the type of damage and prints a message. It also logs the event to the ngStats log. Finally, it calls ScoreKill.

ScoreKill awards points for a kill. DeathMatchPlus assigns a frag for a successful kill and subtracts one for a suicide. TeamGamePlus also adds a point to the TeamInfo of the Killer's team, or subtracts one in the case of a suicide.

DiscardInventory is called whenever a player dies or is removed from the game. DiscardInventory goes through a pawn's inventory, tossing out weapons and destroying others as appropriate. You might override this function if you wanted to toss out a backpack or a trap.

Finally, the EndGame function is called with a reason whenever the game ends. You might want to perform special logging or clean up here.

So thats a quick look at the more important GameInfo functions. The advanced GameInfo classes like DeathMatchPlus add important new behavior for controlling bots and single player games, as well as refining the GameInfo methods into specific rules. We'll look at DeathMatchPlus next.

DeathMatchPlus, a specific game type.

Adding a Heads Up Display

Notice the GameInfo variable HUDType. This is used to specify the type of HUD the player will be given if they play your game. DeathMatchPlus uses a HUDType of Botpack.ChallengeHUD. The ChallengeHUD class is the primary HUD for Unreal Tournament. Let's take a look at adding custom HUD elements.

First, create a subclass of ChallengeHUD. Lets call it MyHUD:

```
class MyHUD extends ChallengeHUD;
```

Now add MyHUD to your gametype's HUDType. In the defaultproperties set MyHUD equal to class'MyPackage.MyHUD' Remember, you can't see defaultproperties if you are editing UnrealScript from UnrealED. Make sure you've exported the source classes and are editing using your own text editor like CoolEdit or MS Dev Studio.

A HUD does all of its drawing in the PostRender function. PostRender is called after the world has been drawn and all the models in the world have been drawn. The function passes you a canvas, which is an object that is used as an interface to the player's screen. Add a PostRender function to MyHUD:

```
function PostRender(canvas C)
{
        Super.PostRender(C);
```

```
}
```

What does the Super call do? It calls the parent class version of PostRender. MyHUD's parent class is ChallengeHUD, so that version of PostRender is called. If you add your custom code after the call to Super.PostRender, you'll be able to add elements to the HUD that will draw on top of all the other HUD elements. If you don't call Super.PostRender all of the basic HUD elements like weapon readouts and so forth will not be drawn.

The Canvas class is defined in the Engine package. You might want to open it up and get a look at its member functions. The ChallengeHUD class is full of good examples on how to draw stuff to the HUD. As an example, lets just draw the player's name on the HUD:

```
function PostRender(canvas C)
{
        Super.PostRender(C);

        C.SetPos( 0, Canvas.ClipY/2 );
        C.DrawColor.R = 255;
        C.DrawColor.G = 0;
        C.DrawColor.B = 0;
        C.Font = MyFonts.GetBigFont( C.ClipX );
        C.DrawText( PlayerOwner.PlayerReplicationInfo.PlayerName, False );
}
```

This code sets the canvas drawing position to halfway down the screen and all the way to the left. Next, it sets the drawing color to be a deep red. It then asks MyFonts (the ChallengeHUD font info object) to return an appropriate big font. The font size returned depends on the screen's resolution, so we have to tell the FontInfo class what the X length of the screen is. Finally, we draw the player's name.

Your HUD can be much more complex...adding scrolling features and new types of information readouts. You'll want to look over ChallengeHUD's PostRender function and see how it gets information about the world from PlayerOwner and other related objects. Skillfully changing the HUD can add a whole new look and feel to your modification.

Adding a Scoreboard

Scoreboards work just like HUDs. If you create a new scoreboard class that extends TournamentScoreboard and put the class in your gametype's ScoreBoardType variable, players will be given a scoreboard of that kind. A scoreboard draws its information whenever the Player's bShowScores variable is true. It draws through the PostRender function and is given a Canvas parameter. Look at TournamentScoreboard for an example of player sorting and score listing.

Pickups

Coming soon.

Weapons

Coming soon.