Index of Unreal Script

U n r e a l S c r i p t

I n t r o d u c t i o n

So, I'm guessing the first question floating through your mind would be, What the hell is UnrealScript? Well, UnrealScript is the mini programming language that Unreal mods are written in. If you've had experience coding with C++ or JavaScript, you'll probably catch on quickly. UnrealScript syntax is almost identical to JavaScript, so JavaScript books and tutorials are good resources for learning your UnrealScript vocabulary.
If you've never written a line of code in your life, though, don't give up. Everyone has to start somewhere, and UnrealScript is as good a place as any. I've tried to make this tutorial as simple and basic as possible, so it can be understood by just about anyone.
Like anything else, UnrealScript takes practice to become good at, but that doesn't mean it can't be fun along the way.

L e t ' s g e t s t a r t e d . . .

There are two methods for writing UnrealScript. The first, and simplest, is to use UnrealEd, which comes fully featured with everything you'll need to get started in UScript. The second method involves writing code in plain text .uc files, and compiling them using Unreal -make. This is usually the preferred method for most experienced UScripters, because it gets rid of complications and bugginess caused by UnrealEd. It also allows for easier mixing and matching of new models. Information on how to use this method is contained in the Using Unreal -make tutorial. If you're just starting out, though, I would have to recommend that you stick to UnrealEd for now. As a result, this is the method I'll talk about most during this tutorial. If you've never run UnrealEd before, you'll need to grab a couple of bug fixes to make sure it works properly. Download and install the Visual Basic 5.0 Runtime, and the RichText Control Update, and you should be set. Alright. It's time to start your career in UnrealScript. Open up UnrealEd, and take a look around. The first thing you'll probably notice are the grided viewports in the center. These are meant for level design and you won't be using them for writing UnrealScript. Now take a look on the right. This is the browser window. By default, it displays textures for level design, but this isn't what you want. Click on the "Browse" pull-down menu, and select “Classes". This will bring up the class tree.

W h a t i n t a r n a t i o n i s a c l a s s ?

You may have heard the term "Object Oriented Programming" before. C++, Java, JavaScript, and UnrealScript are all object-oriented languages. OOP is a fairly new concept, and it's one that can make the task of programming quite a bit easier. Especially when you're writing code for an FPS, where it's easy to think of things as actual "objects" in the game. Everything you see and interact with in Unreal (as well as quite a few things you can't see) is an object. Your eightball gun is an object. The rockets and grenades it fires are objects. The unfortunate krall at the other end of these rockets and grenades is an object. All of these things are controlled by written code, which is contained in a class. So, there's the answer to that question. A class is simply a collection of code which is used to control some object in the game. Each object has its own class.
A popular analogy is to think of a class as a mold that is used to create objects in the game. You might have more than one skaarj in a game at the same time, but that doesn't mean that these skaarj are identical. One of them could be patrolling peacefully at its post, and the other might be fighting for its life against a blood-crazed player with an attitude and a big gun. They are both created from the same class, or "mold", but they are controlled separately. In case you're new to 3D game development, there's something I should probably explain at this point. I've said an object is something in the game that can (usually) be seen and interacted with. What you "see" however is not dictated by the code you write for it. What you see is a 3D model which is created in a separate program entirely, such as 3D Studio Max or Lightwave. The code you write controls what the object does. A model without code will just it there and do nothing in the game. Code makes your eightball fire when you click the mouse button, makes the rocket appear in front of you, and makes it speed off to explode between your enemy's eyes.

M o v i n g O n

Now that you have some concept of what a class is, it's time to look at them in a little more depth. Go back to to the class browser in UnrealEd, and look it over a bit. Classes in Unreal are arranged in a hierarchy, with the

"Actor" class at the top. Actually, Actor is not the highest class, but it's as high as you'll need to go for now. Just so you know, "Object" is at the true top of the tree, and it can be displayed in the class browser by deselecting the Only show actor classes box. The idea behind having the classes arranged in a hierarchy is that each class will inherit code from the classes above it. Code that will be used for every class in the game is put in the top-most class, so it will be inherited by all the classes below it. This is very useful, since it means that you don't have to re-invent the wheel for each new class you create. If you want to create a new weapon, for instance, you can simply expand upon the existing Unreal weapon class, and add only the functionality that is specific to your weapon, instead of unnecessarily re-writing code that is already written in the base weapon class. Now, click on the little minus sign by the word "Inventory" to display its child classes. After that, expand "Weapon", and look at what appears. All the Unreal weapons are child classes of Weapon, which is a child class of Inventory, which is a child class of Actor. There is quite a bit of code in Inventory and Weapon which controls the basics of how a weapon should act, but the specific code that controls how each individual weapon works is contained in that weapon's class. To get your first look at UnrealScript in all its glory, double-click on the FlakCannon class. A window with a dark blue background will appear, containing all the code for the FlakCannon. If you've written C++ or JavaScript before, you'll probably recognize quite a bit of the syntax. If you're new to programming, though, don't panic. Code may look complicated at first, but once you break it down, it's really very simple. Code in UnealEd is color coded, as you've probably noticed already. Comments (text which is ignored by the compiler, and used to explain and document your code) are bright green, keywords are aqua blue, labels are yellow, exec commands are gray, and everything else is white. The first line of aqua blue and white when you first open the class is called the class declaration. Under this are the gray exec commands. These are used to import the models, sounds, and textures used by the class, and can be ignored for now. Scroll down till you get to some more colorful code. This code contains the variables, functions, and states of the class, and is what actually controls what the FlakCannon does.

The Class Declaration

The class declaration is a line of code in a class which states the name of the class, and its parent class. The class declaration for the FlakCannon looks like this:

FlakCannon Class Declaration

```
class FlakCannon expands Weapon;
```

Not too difficult, is it? All it consists of is the word "class", followed by the name of the class, then the word "expands" followed by the name of the parent class and a semicolon. The semicolon is just a way of telling the compiler that the statement is finished. Just about everything you write in UnrealScript will need a semicolon at the end, so get used to it. Now, when writing code in UnrealEd, you won't have to worry about the class declaration much, since UnrealEd will automatically generate this line of code when you create a new class. However, if you write code in text-based .uc files outside of UnrealEd, you will need to write the class declaration manually.

Introducing: Variables

If you've ever done any programming before, I'm sure you have a firm concept of what a variable is, and what they're used for. If this is the case, you should probably skip down to the "Types of Variables" section below. If the question What the hell is a variable? is floating around in the back of your mind (or the front of it, for that matter), though, you'll want to keep reading. Technically speaking, a variable is a location in your computer's memory that stores a piece of information. This information can be of many different types, such as numbers or words. Variables come in handy all the time while writing code. For example, let's say you're making a new weapon, and you want it to charge up in alt-fire. To accomplish this, you could use a variable. When the player presses alt-fire, have Unreal add to this variable. Then, when the player presses fire, have Unreal fire a projectile that does a varying amount of damage according to the value that was stored in your charge variable.Damn, I didn't do a very good job explaining that, did I? Well, hopefully you're able to grasp the concept of variables without much help from me. I've found that most people don't have much trouble with it. It's just one of those things that naturally makes sense.

Types of Variables

If you've done any programming in BASIC, or a similar language, you've probably become accustomed to using variables a certain way. Namely, not having to declare them. Declare them, you ask? Yes, declare them.

Variables in UnrealScript, just as in C++, Java, and JavaScript, must be declared before you can use them. Basically, you have to let Unreal know that you are going to use a new variable. The basic variable declaration syntax in UnrealScript looks like this:

The Variable Declaration

```
var [vartype] [varname];
```

Pretty straight-forward. First comes the keyword "var", then the type of variable you are declaring, and finally the name of the variable. Variables must be declared at the beginning of a class, after the class declaration and exec commands, but before any functions. There are many different types of variables, ranging from numbers, to letters and words, to "true" or "false" values. The types available in UnrealScript are as follows:

I n t e g e r   Keyword: int

An integer number value. For those of you not familiar with what an integer is, it's a whole number that can also be negative. In other words, anything without a decimal. 37 is an integer. 2 is an integer. -674 is an integer. 6.3432 is not an integer.

Integer Example

```
var int myInt;
myInt = 3; //Assign a value
myInt++; //Increment
myInt--; //Decrement
```

Note the special syntax you can use to add or subtract one from an integer. Saying "myInt++;" does the same thing as saying "myInt = myInt + 1;", and saying "myInt--;" does the same thing as saying "myInt = myInt - 1;".

F l o a t i n g P o i n t   Keyword: float

A number value that, unlike an integer, can include decimals. 6.3432, 4534243.2, and -0.98 are all floating point numbers.

Floating Point Example

```
var float myFloat;
myFloat = 3.2453; //Assign a value
```

You cannot increment and decrement a float using the same syntax as an int. To add one to a float, you would have to say myFloat = myFloat + 1;", not "myFloat++;". It's also important to keep in mind one other thing when working with integers and floats. Take a look at these examples:

Integers and Floats Don't Mix

```
var int myInt;
var float myFloat, Result;
//Example 1
myInt = 5;
myFloat = 0.5;
Result = myFloat * myInt;
//Example 2
myInt = 5;
myFloat = 2.0;
Result = myInt / myFloat;
```

In example one, Result will not equal 2.5. Because you are multiplying an integer by a float, you will always get an integer back. If you wanted to get 2.5 back, you would have to declare myInt as a float, not an int. The same is true for example two. Even though you are dividing 5 by 2, and assigning the result to a float, you will not get 2.5 back, because the 5 is an integer. Also, note the way I've declared the two floats in this example. You can declare multiple variables in the same statement by simply separating the different variable names with commas.

B o o l e a n V a l u e Keyword: bool

A value which is either "true" or "false". These have not always been around, since it's possible to simply use an integer, and set it to either 0 or 1. This would give the same effect. However, it's easier to see and nderstand the words true and false than it is to understand a 0 or 1, so the bool was introduced.

Boolean Example

```
var bool bMyBool;
bMyBool = true; //Assign a value
```

Note that the prefix "b" is often used in boolean variable names. This is just a naming convention, though, and it's not a required.

B y t e Keyword: byte

An integer value in the range of 0 to 255. The use of these may not be apparent at first glance. You're probably saying to yourself, Why not just use an integer? Well, I honestly can't answer that. I very rarely (if ever) use these, so I'm not extremely clear on their advantages. I can give you an example of one thing they are used for, though, and that is RGB color values, which fall in the range of 0 to 255.

Byte Example

```
var byte bMyByte;
bMyByte = 255; //Assign a value
```

Again, note the prefix "b". For some reason, Epic chose to use the same naming convention for both bytes and bools. So, be careful not to confuse the two when looking through existing scripts.

S t r i n g Keyword: string

A string is simply a bunch of alpha-numeric characters. In other words... well, words. Strings of letters and numbers that make up words and sentences.

String Example

```
var string[32] String1; //Declare string
var string[32] String2;
var string[32] Result;
String1 = "Blah"; //Assign a value
String2 = "Gah";
//Combine two strings
Result = String1 $ String2;
//Find left 2 characters of String1
Result = Left(String1, 2);
//Find right 2 characters of String1
Result = Right(String1, 2);
//Find the number of characters in String1
Result = Len(String1);
//Return String1 in all caps
Result = Caps(String1);
```

Note that strings are declared in a special way. The number in the square brackets after the word "string" is the maximum number of characters the string can be. In this example, the maximum length of String1, string2, or Result would be 32. There are also many special operations which can be performed on strings, such as "concantations" (or combining two strings into one), finding left or right characters, or finding the length of the string.

N a m e Keyword: name

Names are a tough one. They're hardly ever used, and understood by few. I'll do my best to explain them, though. The only application I can think of for them is in tags. If you've done any level design, you've probably used tags. They're used to associate one object with another in order to trigger certain events. Anyway, tags are simply name variables. It can be easy to confuse names with strings, but names are not strings. A string can be modified dynamically, but a name is simply a label for an object.

Name Example

```
var name MyName;
MyName = 'Windex'; //Assign a value
```

E n u m e r a t i o n Keyword: enum

Enumerations are simply a way of defining a type of variable that can be one of a certain pre-defined set of values. Like bools, they're not absolutely necessary, since integers could be used to get the same effect. However, it's easier to see understand a set of descriptive words instead of a bunch of numbers.

Enumeration Example

```
//Declare a new enumeration
enum EColor;
{
    CO_Red,
    CO_Blue,
    CO_Green
};

//Declare variable of type EColor
var EColor MyColor;

//Assign a value
MyColor = CO_Blue;
```

Note that the "CO_" preceding each of the color values is simply a naming convention, and is not required. You can name your enumeration values anything you want.

A c t o r R e f e r e n c e Keyword: n/a

Actor references are a special type of variable that references an actual object in the game. It will be difficult to fully grasp them at this point, but I'll give you the basics, at any rate. Later on, once I've introduced functions, I'll go into more detail about them.

Actor Reference Example

```
//Declare a reference to any actor
var Actor MyActor;
//Declare a reference to a pawn
var Pawn MyPawn;
//Declare a reference to a weapon
var Weapon MyWeapon;
```

You'll notice I didn't include any examples explaining how you assign a value to an actor reference. This is because you can only set an actor reference equal to another actor reference, or to a newly spawned actor using the Spawn() function. As I said, actor references are difficult to explain at this point, but I'll go into more detail later, once you've learned a bit more.

C l a s s R e f e r e n c e Keyword: class

Class references, like actor references, are a special type of variable. Also like actor references, they're difficult to explain at this point. I'll do my best, though. A class reference, unlike an actor reference, doesn't reference an actual object in the world. It references a class itself, or the mold. It references the thing that creates objects, instead of the objects themselves.

Class Reference Example

```
//Declare a class reference
var class<Actor> MyClass;
//Assign a value
MyClass = Class'Pawn';
```

The word "Actor" in angle brackets after the word "class" in the declaration means that Actor is the upper limit of this variable. What this means is that MyClass cannot be set equal to anything higher than Actor in the class tree. Class references are assigned values by using the keyword Class, followed by the name of a class in single quotes.

S t r u c t Keyword: struct

A structure is a way of defining a new type of "super-variable" that is made up of individual components. A struct is actually similar to a class, although a very simple class that can only contain variables. Structs define a new type of variable that can be declared and used just as any other variable. Take this example:

Struct Example

```
//Define a new struct
struct Box
{
    var float Length;
    var float Width;
    var float Height
};

//Declare a couple variables of type Box
var Box MyBox, YourBox;

//Assign values to individual components
YourBox.Length = 3.5;
YourBox.Width = 5.43;
YourBox.Height = 2.8;

//Set MyBox equal to YourBox
MyBox = YourBox;
```

The struct defines a new type of variable, called "Box", which has three sub-variables to it: length, width, and height. Once you define a variable of the new type, you can assign values to its individual components with the syntax "VarName.ComponentName". One very common struct in UnrealScript is the vector, which is made up of X, Y, and Z components. The techniques of working with vectors are somewhat complex, and you can learn more about them in the Vectors tutorial.

C o n d i t i o n a l s

If you've ever done any programming before, you're almost sure to be familiar with the If/Then/Else statement. They exist in UnrealScript as well, although the syntax might be slightly different than what you're used to if you program in a BASIC language. If you've never programmed before, then allow me to explain. A conditional is a way of having Unreal perform certain operations only if a certain condition is met. For instance, do one thing if a bool is true, and do something else if it's false. Conditionals are key to accomplishing all sorts of things in any programming language, and UnrealScript is no exception. The basic syntax for a conditional in UnrealScript is:

Conditional Syntax

```
if ([expression1] [operator] [expression2])
```

```
{
    Do some stuff here;
}
else if ([expression1] [operator] [expression2])
{
    Do more stuff here;
}
else
{
    Hey, look, more stuff;
}
```

First, Unreal checks to see if the first condition is true by comparing expression1 to expression2 using the operator. If that condition checks out, then the first set of commands are executed, and the conditional is finished. If the first condition isn't true, though, Unreal will check the second condition, and if it's true, it'll execute the second set of commands. If it goes through all the conditions, and none of them are true, it will execute the "else" set of commands. When writing a conditional, you don't have to have else if's and else's. They're just available should you need to be more specific with what you want Unreal to do. All you have to have when writing a conditional is the first "if" statement. There are many different operators that can be used in conditionals, as you can see in this table:

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
~=	Approximately equal to

Not every operator will work with every variable type. For instance, you can't really say that one actor reference is "greater than" another actor reference, so the four greater than/less than operators aren't applicable to actor references. Just use common sense to determine what will work with what, and you should be just fine.

Basic Conditional

```
var bool bSomeBool, bSomeOtherBool;
var int SomeInt, SomeOtherInt;

if (SomeInt > 3)
{
    SomeInt is greater than 3, so do
    something;
}
else if (SomeOtherInt <= SomeInt)
{
    SomeInt is not greater than 3,
    but SomeOtherInt is less than or
    equal to SomeInt, so do something
    else;
}
else
{
    All the conditionals failed, so
    do this;
}
```

```
if (bSomeBool)
{
    bSomeBool is true, so do this;
}
else if (!bSomeOtherBool)
{
    The first conditional failed, but
    bSomeOtherBool is false, so do this
    instead;
}
```

Note the way I used the bools in the second conditional. Because bools can only be one of two values (true or false), they don't need to be compared using two expressions. Saying "if (bSomeBool)" is the same as saying "if (bSomeBool == true)", and saying "if (!bSomeOtherBool)" is the same as saying "if (bSomeOtherBool == false)". Now, moving on, what if you wanted to do something only if two conditions were true? Or what if you wanted to do something if only one of two different conditions were true? That's where these operators come in:

Operator	Description
&&	And
||	Or

These are used to link conditions together in the same statement.
Take a look at this example:

Conditional with && and ||

```
var bool bSomeBool;
var int SomeInt, SomeOtherInt;

if (SomeInt > 3 && SomeOtherInt < 3)
{
    SomeInt is greater than three,
    and SomeOtherInt is less than
    three, so do something;
}
else if (SomeOtherInt == SomeInt || !bSomeBool)
{
    The first condition failed, but
    either SomeOtherInt equals SomeInt,
    or bSomeBool is false, so do this
    instead;
}
```

In the first one, && links the two statements together, so the condition is only true if both statements are true. In the else if, || links the two expressions together, so the condition will be true if either of the statements is true.

Other Flow Control Devices

In addition to "if" statements, there are other ways to control how code flows. Things such as loops and switch statements will allow you to fine-tune your code, and get the results you want. To be honest, I've never used a switch statement in UnrealScript, but I'll explain them anyway, since everyone's coding style is different. Loops, however, I use all the time. They can be extremely useful to do certain things. There are three types of loops in UnrealScript, which I will explain below.

For Loops

For loops are the type I use the most, since I've found that they usually fit my needs just as well or better than the other two types. The basic concept of a for loop is to execute a certain block of code over and over again, until a certain condition is met.

For Loop Example

```
var int i;

for ( i=0; i<5; i++ )
{
    Do stuff;
}
```

The first statement in the parenthesis, "i=0", sets the initial value of i as the loop starts. The second statement, "i<5", is the condition that must be met for the loop to continue executing. As soon as i is greater than or equal to 5, the loop will terminate. The final statement, "i++", is what is done to i each time the loop executes. So, the first time this loop executes, i will equal 0. The next time, i will be incremented by one, making it equal 1. This is still less than 5, so the loop executes again. Next time, i will be 2, then 3, then 4, and finally 5. The loop will terminate once it gets to 5, since 5 is not less than 5.

Do Loops

Do loops, unlike for loops, have no built-in expressions for incrementing counters or setting initial values. They simply execute over and over until a condition at the end of the loop is met. Because they have no built-in expression for incrementing a variable, you will have to include a line within the loop that somehow increments or changes your counter variable, so that the ending condition will eventually be met. Otherwise, you get an infinite loop. Not a good thing.

Do Loop Example

```
var int i;

do
{
    Do stuff;
    i++;
} until (i == 5);
```

You'll notice I included the line "i++;" within the loop. This will increment i each time the loop executes, so it will terminate when i gets to 5. The main distinction of the do loop is the fact that it executes until some condition is true. Both for and while loops execute while some condition is true.

While Loops

While loops are basically do loops, except for the fact that they execute while their condition is true, whereas do loops execute until their condition is true. Again, you'll have to include a line in the loop to somehow increment your counter variable, since there is no built-in expression for this in the loop declaration.

While Loop Example

```
var int i;

while ( i < 5 )
{
    Do stuff;
    i++;
}
```

Switch Statements

A switch statement is basically like a complicated if statement. It allows you to execute different blocks of code depending on the value of a certain variable. Take a look at this example:

Switch Statement Example

```
var string[32] Developer;

switch ( Developer )
{
    case "Tim":
            Hey, it's Tim. Do something;
            break;

    case "Cliff":
            Look, there's Cliffy.
            Do something else;
            break;

    case "Myscha":
            Where'd Myscha come from?
            Better do something else;
            break;

    default:
            No one here;
            break;
}
```

Not too complicated. You just supply the variable you want to use for the switch in the first parameter, then write different "cases" depending on the different values of the variable. The final "default" label is optional, and will be executed if none of the other cases are true. Note the break statements marking the end of each case. Like I said before, switch statements are basically just complicated if statements. I rarely have use for them, since the same effects can be achieved simply by using an if/else if/else.

Functions

I have a little confession to make. You know all the examples I've been giving so far, in which I declare a variable or two, then jump right into some code, such as assigning values to these variables, or writing an if statement or a loop? Well, that was illegal. In actual UnrealScript, you cannot just write code by itself. The only parts of a class that can be completely on their own are the class declaration, variable declarations, and exec commands. Everything else must be part of a function or state. So, what's a function, you ask? A function is just a block of code that performs some action. Once they're defined, they can be called in other parts of the code, to do whatever it is they're supposed to do. They can be given, or passed, variables when they're called, and they can return values. I know this all probably sounds very complicated (assuming you've never done any programming before), but it's really fairly simple once you understand it. Take a look at this example:

Simple Function Example

```
var int SomeInt, Result;

//Take an integer, and return its square
function int Sqr( int Num )
{
    return Num * Num;
}

//Test the Sqr() function
```

```
function PostBeginPlay()
{
    SomeInt = 3;
    Result = Sqr(SomeInt);
}
```

There are two functions here, Sqr(), and PostBeginPlay(). Sqr() takes a number, Num, multiplies it by itself, and returns it. You'll notice that I didn't declare Num up with SomeInt and Result. This is because it is "declared" as a parameter to a function. When I call Sqr() down in the PostBeginPlay() function, I supply SomeInt as the value in parenthesis, or the parameter. The call to Sqr() causes the code contained in the Sqr() function to be executed, with SomeInt plugged in for Num. You'll also notice that I put the call to Sqr() after "Result =". This is because I am assigning the value which is returned by Sqr() to Result. When all this code is done executing, Result will be equal to 9: the square of 3. You may also have noticed the keyword "int" before the function name in Sqr()'s definition. This "int" means that Sqr() returns an integer value. Anyway, you may be wondering by now, Where is PostBeginPlay() being called from? The answer is, the engine. There are a wide variety of functions in UnrealScript which are called by the engine in certain places and under certain circumstances. The PostBeginPlay() function is called when an object is first created, so it makes a good place to put code that you want to be executed before any other code. For a list of common functions which are called by the engine (as well as other useful functions which aren't called by the engine), refer to the Function Reference at the side of this page. So, are you thoroughly confused yet? If not, then you're doing good. I know I was scratching my head quite a bit when I first learned this stuff. Well, keep reading, it gets better (or worse, depending on your viewpoint). You know the way I've been declaring variables all along? At the beginning of a class, using the syntax "var [vartype] [varname]"? Well, that's not the only way you can declare a variable. That type of variable, declared outside of any functions, is called a global variable. Global variables can be accessed anywhere in a class, and even outside of a class (as we'll see a little later). But, there are also local variables. Local variables are declared at the beginning of a function, and can only be accessed within that function. They're useful for doing short-term operations that won't need to be "seen" outside of a particular function. You see, one of the key elements of a function, and of object-oriented program as a whole, is the fact that a class or function can share useful data and important information with other classes and functions, but hide how they got that useful data and information. They show only the result, but not how they found the result. In any case, where was I going with this? Oh, yes. Local variables. Local variables are declared just like global variables, only they use the local keyword instead of the var keyword.

Local Variables

```
function PostBeginPlay()
{
    //Declare a local integer
    local int SomeInt;

    //Assign a value
    SomeInt = 3;
}
```

So, that's a local variable. Not too complicated, is it? Just like a global variable, except for the fact that it can only be accessed inside a particular function.

I n h e r i t a n c e

Inheritance, you ask? What could inheritance possibly have to do with programming? Well, it has a lot to with programming. At least when you're talking about classes. If you'll remember, I told you earlier that one of the reasons classes are arranged in a hierarchy is that child classes inherit code from their parent classes. Well, I wasn't just saying that to watch myself type. A class will inherit all variables, functions, states, and default properties (I'll talk about states and default properties a bit later) from every class above it in the hierarchy. For example, the Weapon class has in it all the code written in the Inventory class, the Actor class, and the Object class, since these are the classes above it in the hierarchy. Any new code you write in a class is simply added on to the code inherited from parent classes. But what if you wanted to change a certain inherited function? Well, you can. It's called overriding a function. All you have to do is copy the function definition into the new class (the name, parameters, and return value type), and write new code for it. The ability to do this is extremely useful in UScript, since it allows you to add or change functionality in things without having to copy over all the code. For instance, say you wanted to make an ASMD that launched grenades in alt-fire instead of the little blue energy ball thingy. All you would have to do is copy the one function that controls what happens when the player presses alt-fire, and make a few little changes. Nothing to it.

States

No, not the United kind. We're talking about UnrealScript here, remember? Anyway, a state is simply a section of code that is executed only the class is in that state. For instance, what are the different states that a weapon could be in? It could be firing, alt-firing, reloading, or just sitting there looking pretty. Each of these conditions could have their own state defined for them, which would contain code that's only used when the weapon is in that condition. For instance, if you wanted a weapon to play an idle animation every 30 seconds, you could put a looping timer in the idle state, so it would only run when the weapon was not doing anything. To give you an idea of how they're defined, here's the actual Idle state from the Weapon class:

The Weapon Idle State

```
state Idle
{
    function AnimEnd()
    {
          PlayIdleAnim();
    }

    function bool PutDown()
    {
          GotoState('DownWeapon');
          return True;
    }

Begin:
    bPointing=False;
    if ( (AmmoType != None)
       && (AmmoType.AmmoAmount<=0) )
          Pawn(Owner).SwitchToBestWeapon();
    if ( Pawn(Owner).bFire!=0 ) Fire(0.0);
    if ( Pawn(Owner).bAltFire!=0 ) AltFire(0.0);
    Disable('AnimEnd');
    PlayIdleAnim();
}
```

Code in states can be written either within functions, or under labels. Begin is by far the most common label, and any code written under it is executed as soon as the class enters that state. Another cool thing about states is that you can use them to override functions within a particular class. For instance, say you had a Timer() function defined outside of a state. If you defined another Timer() function within a state, then that Timer() function would override the global one if the class was in that state. Another useful thing you can do with a state is to stop certain functions from executing while the class is in that state. For example, if you wanted to make it so the player couldn't fire while his gun was reloading (usually a good idea), you could add this line just after the definition of your reload state:

The Ignore Statement

```
ignores Fire, AltFire;
```

This makes it so neither the Fire() or AltFire() function can be executed while the class is in this state. To make an object enter a state, use the syntax: "GoToState('State');".

Default Properties

Default properties are simply a means by which you, as the programmer, or someone else, such as a mapper, can set default values for certain variables in a class. Default properties are used to control many things, such as how to display a class, what mesh or texture to use, and what sounds to use. If you want a variable to be displayed in the default properties of a class, you have to declare it in a special way:

Default Properties Variable

```
var([defaultgroup]) [vartype] [varname];
```

The part in the parenthesis, defaultgroup, tells Unreal what section of the default properties you want the variable to be displayed in. If you don't supply anything for this parameter, it will be displayed in a section with the same name as the class. You can look at and change the default properties of a class by selecting it in the class browser, and clicking the "Defaults" button.

Your First Class

Here it is. The moment you've been waiting for since... since... well, since you started reading this sentence, I suppose. It's time to create your first new UnrealScript class. To be specific, you're going to make a new type of FlakCannon that randomly alternates between firing a flakshell, a grenade, or an energy ball in alt-fire. Not the most exciting weapon there ever was, but hey, this is a tutorial for beginners. To start off, I suppose I should explain a little something about the way Unreal is organized. All classes, sounds, textures, and models are stored in special files called packages. Most of the code and models, and some of the textures and sounds for Unreal are stored in two files: unreali.u, and unrealshare.u (as of 220, anyway). These files are found in your Unreal\System directory, as are all .u files. When you create a new class in UnrealScript, you store that class in a new .u file. It's not a good idea to store a new class in an existing .u file, since you would then have to pass around the entire file if you wanted to distribute your class. With that out of the way, let's get ready to rumble. Open up UnrealEd (if you don't already have it open), and get to the class browser. Open up the Inventory and Weapon threads, and select the FlakCannon class. Now, hit the "New..." button, which can be found down below the browser. You'll get a window asking you to enter a class name, and a package name. Enter "MultiCannon" in both fields, and press the "Create this actor class" button. You'll see that your MultiCannon class will appear under FlakCannon in the class tree, and an editor window will appear, complete with the class declaration. The next thing to do is copy the AltFire() function from the FlakCannon class, since we want to modify what the weapon shoots in alt-fire. AltFire() is simply a function which is called by the engine when the player presses the alt-fire button. It's used to control what happens when a weapon alt-fires. The same goes for the Fire() function, but we're not modifying primary fire here, so we don't need Fire(). Anyway, double-click on FlakCannon in the class browser to open it up. Scroll through the code until you find the AltFire() function, and copy it into your MultiCannon class. You'll notice that the new code appears as all green when you first copy it. This is because UnrealEd doesn't apply the proper coloring to code until you compile it. So, let's compile it. Compiling in UnrealEd couldn't be easier. Simply hit the F7 key to compile all modified classes. Now, add the following local variable declarations to your new AltFire() function:

New Variable Declarations

```
local projectile p;
local class<projectile> Proj;
local float Selection;
```

Next, find the line "Spawn(class'FlakShell',,, Start,AdjustedAim);", and replace it with the following lines:

New Code

```
Selection = FRand();
if ( Selection < 0.4 )
    Proj = Class'FlakShell';
else if ( Selection < 0.7 )
    Proj = Class'Grenade';
else
    Proj = Class'TazerProj';
```

```
p = Spawn(Proj,,, Start,AdjustedAim);
if ( Proj == Class'Grenade' )
    p.DrawScale *= 1.5;
```

That's it. Those are all the modifications that need to be made. Hit F7 again to compile the changes, and you're set. So, what does all this code do, you ask? Well, it's simple, really. Selection is set equal to FRand(), which returns a random number between 0.0 and 1.0. Then, an if/else if/else is used to set Proj to either FlakShell, Grenade, or TazerProj, depending on the value of Selection. Whatever Proj is set to is then spawned (Spawn() is a function that brings a class into existence in the Unreal world), and at the same time, p is set equal to this new projectile. Now, here's something I haven't explained quite yet. The line "p.DrawScale *= 1.5;" is used to reference a variable called "DrawScale" in p, and assign it as itself times 1.5. This is useful notation. If you want to reference a variable or function in another class, all you have to do is specify which class it's in by putting the name of the class, followed by a dot and the name of the variable or function. So, that's it. Your first new UnrealScript class. Before the MultiCannon package will actually be written to your hard disk for use in-game, though, you'll have to save it. Hit the "Save" button at the bottom of the class browser, and select "MultiCannon" in the pull down menu of the window that appears. You'll need to do this every time you make any changes to one of your classes. Now, to try out your work in a game, start up Unreal, go to the console, and type "summon multicannon.multicannon" Good luck, and I hope to see some kick ass weapons from you :)

UnrealScript Language Reference

Introduction

Purpose of this document

This is a technical document describing the UnrealScript language. It's not a tutorial, nor does it provide detailed examples of useful UnrealScript code. For examples of UnrealScript prior to release of Unreal, the reader is referred to the source code to the Unreal scripts, which provides tens of thousands of lines of working UnrealScript code which solves many problems such as AI, movement, inventory, and triggers. A good way to get started is by printing out the "Actor", "Object", "Pawn", "Inventory", and "Weapon" scripts.

This document assumes that the reader has a working knowledge of C/C++, is familiar with object-oriented programming, has played Unreal and has used the UnrealEd editing environment.

For programmers who are new to OOP, I highly recommend going to Amazon.com or a bookstore and buying an introductory book on Java programming. Java is very similar to UnrealScript, and is an excellent language to learn about due to its clean and simple approach.

Design goals of UnrealScript

UnrealScript was created to provide the development team and the third-party Unreal developers with a powerful, built-in programming language that maps naturally onto the needs and nuances of game programming.

The major design goals of UnrealScript are:

- To support the major concepts of time, state, properties, and networking which traditional programming languages don't address. This greatly simplifies UnrealScript code. The major complication in C/C++ based AI and game logic programming lies in dealing with events that take a certain amount of game time to complete, and with events which are dependent on aspects of the object's state. In C/C++, this results in spaghetti-code that is hard to write, comprehend, maintain, and debug. UnrealScript includes native support for time, state, and network replication which greatly simplify game programming.
- To provide Java-style programming simplicity, object-orientation, and compile-time error checking. Much as Java brings a clean programming platform to Web programmers, UnrealScript provides an equally clean, simple, and robust programming language to 3D gaming. The major programming concepts which UnrealScript derives from Java are: a pointerless environment with automatic garbage collection; a simple single-inheretance class graph; strong compile-time type checking; a safe client-side execution "sandbox"; and the familiar look and feel of C/C++/Java code.
- To enable rich, high level programming in terms of game objects and interactions rather than bits and pixels. Where design tradeoffs had to be made in UnrealScript, I sacrificed execution speed for development simplicity and power. After all, the low-level, performance-critical code in Unreal is written in C/C++ where the performance gain outweighs the added complexity. UnrealScript operates at a level above that, at the object and interaction level, rather than the bits and pixels level.

During the early development of UnrealScript, several major different programming paradigms were explored and discarded before arriving at the current incarnation. First, I researched using the Sun and Microsoft Java VM's for Windows as the basis of Unreal's scripting language. It turned out that Java offered no programming benefits over C/C++ in the Unreal context, added frustraging restrictions due to the lack of needed language features (such as operator overloading), and turned out to be unfathomably slow due to both the overhead of the VM task switch and the inefficiencies of the Java garbage collector in the case of a large object graph. Second, I based an early implementation of UnrealScript on a Visual Basic variant, which worked fine, but was less friendly to programmers accustomed to C/C++. The final decision to base UnrealScript on a C++/Java variant was based on the desire to map game-specific concepts onto the language definition itself, and the need for speed and familiarity. This turned out to be a good decision, as it has greatly simplified many aspects of the Unreal codebase.

Example program structure

This example illustrates a typical, simple UnrealScript class, and it highlights the syntax and features of UnrealScript. Note that this code may differ from that which appears in the current Unreal source, as this documentation is not synced with the code.

```
//=============================================================================
// TriggerLight.
// A lightsource which can be triggered on or off.
//=============================================================================
class TriggerLight expands Light;

//-----------------------------------------------------------------------------
// Variables.

var() float ChangeTime; // Time light takes to change from on to off.
var() bool bInitiallyOn; // Whether it's initially on.
var() bool bDelayFullOn; // Delay then go full-on.

var ELightType InitialType; // Initial type of light.
var float InitialBrightness; // Initial brightness.
var float Alpha, Direction;
var actor Trigger;

//-----------------------------------------------------------------------------
// Engine functions.

// Called at start of gameplay.
function BeginPlay()
{
    // Remember initial light type and set new one.
    Disable( 'Tick' );
    InitialType = LightType;
    InitialBrightness = LightBrightness;
    if( bInitiallyOn )
    {
            Alpha = 1.0;
            Direction = 1.0;
    }
    else
    {
            LightType = LT_None;
            Alpha = 0.0;
            Direction = -1.0;
    }
}

// Called whenever time passes.
function Tick( float DeltaTime )
{
    LightType = InitialType;
    Alpha += Direction * DeltaTime / ChangeTime;
    if( Alpha > 1.0 )
    {
            Alpha = 1.0;
            Disable( 'Tick' );
            if( Trigger != None )
                    Trigger.ResetTrigger();
    }
    else if( Alpha < 0.0 )
    {
            Alpha = 0.0;
```

```
			Disable( 'Tick' );
			LightType = LT_None;
			if( Trigger != None )
				Trigger.ResetTrigger();
	}
	if( !bDelayFullOn )
			LightBrightness = Alpha * InitialBrightness;
	else if( (Direction>0 && Alpha!=1) || Alpha==0 )
			LightBrightness = 0;
	else
			LightBrightness = InitialBrightness;
}

//-----------------------------------------------------------------
// Public states.

// Trigger turns the light on.
state() TriggerTurnsOn
{
	function Trigger( actor Other, pawn EventInstigator )
	{
			Trigger = None;
			Direction = 1.0;
			Enable( 'Tick' );
	}
}

// Trigger turns the light off.
state() TriggerTurnsOff
{
	function Trigger( actor Other, pawn EventInstigator )
	{
			Trigger = None;
			Direction = -1.0;
			Enable( 'Tick' );
	}
}

// Trigger toggles the light.
state() TriggerToggle
{
	function Trigger( actor Other, pawn EventInstigator )
	{
			log("Toggle");
			Trigger = Other;
			Direction *= -1;
			Enable( 'Tick' );
	}
}

// Trigger controls the light.
state() TriggerControl
{
	function Trigger( actor Other, pawn EventInstigator )
	{
			Trigger = Other;
```

```
            if( bInitiallyOn ) Direction = -1.0;
            else Direction = 1.0;
            Enable( 'Tick' );
    }
    function UnTrigger( actor Other, pawn EventInstigator )
    {
            Trigger = Other;
            if( bInitiallyOn ) Direction = 1.0;
            else Direction = -1.0;
            Enable( 'Tick' );
    }
}
```

The key elements to look at in this script are:

- The class declaration. Each class "expands" (derives from) one parent class, and each class belongs to a "package", a collection of objects that are distributed together. All functions and variables belong to a class, and are only accessible through an actor that belongs to that class. There are no system-wide global functions or variables.
- The variable declarations. UnrealScript supports a very diverse set of variable types including most base C/Java types, object references, structs, and arrays. In addition, variables can be made into editable properties which designers can access in UnrealEd without any programming.
- The functions. Functions can take a list of parameters, and they optionally return a value. Functions can have local variables. Some functions are called by the Unreal engine itself (such as BeginPlay), and some functions are called from other script code elsewhere (such as Trigger).
- The code. All of the standard C and Java keywords are supported, like "for", "while", "break", "switch", "if", and so on. Braces and semicolons are used in UnrealScript as in C, C++, and Java.
- Actor and object references. Here you see several cases where a function is called within another object, using an object reference.
- The "state" keyword. This script defines several "states", which are groupings of functions, variables, and code which are executed only when the actor is in that state.
- Note that all keywords, variable names, functions, and object names in UnrealScript are case-insensitive. To UnrealScript, "Demon", "demON", and "demon" are the same thing.

The Unreal Virtual Machine

The Unreal Virtual Machine consists of several components: The server, the client, the rendering engine, and the engine support code.

The Unreal server controls all gameplay and interaction between players and actors. In a single-player game, both the Unreal client and the Unreal server are run on the same machine; in an Internet game, there is a dedicated server running on one machine; all players connect to this machine and are clients.

All gameplay takes place inside a "level", a self-contained environment containing geometry and actors. Though UnrealServer may be capable of running more than one level simultaneously, each level operates independently, and are shielded from each other: actors cannot travel between levels, and actors on one level cannot communicate with actors on another level.

Each actor in a map can either be under player control (there can be many players in a network game) or under script control. When an actor is under script control, its script completely defines how the actor moves and interacts with other actors.

With all of those actors running around, scripts executing, and events occuring in the world, you're probably asking how one can understand the flow of execution in an UnrealScript. The answer is as follows:

To manage time, Unreal divides each second of gameplay into "Ticks". A tick is the smallest unit of time in which all actors in a level are updated. A tick typically takes between 1/100th to 1/10th of a second. The tick time is limited only by CPU power; the faster machine, the lower the tick duration is.

Some commands in UnrealScript take zero ticks to execute (i.e. they execute without any game-time passing), and others take many ticks. Functions which require game-time to pass are called "latent functions". Some examples of latent functions include "Sleep", "FinishAnim", and "MoveTo". Latent functions in UnrealScript may only be called from code within a state, not from code within a function.

While an actor is executing a latent function, that actor's state execution doesn't continue until the latent function completes. However, other actors, or the VM, may call functions within the actor. The net result is that all UnrealScript functions can be called at any time, even while latent functions are pending.
In traditional programming terms, UnrealScript acts as if each actor in a level has its own "thread" of execution. Internally, Unreal does not use Windows threads, because that would be very inefficient (Windows 95 and Windows NT do not handle thousands of simultaneous threads efficiently). Instead, UnrealScript simulates threads. This fact is transparent to UnrealScript code, but becomes very apparent when you write C++ code which interacts with UnrealScript.
All UnrealScripts execute in parallel. If there are 100 monsters walking around in a level, all 100 of those monsters' scripts are executing simultaneously and independently.

Class overview

Before beginning work with UnrealScript, it’s important to understand the high-level relationships of objects within Unreal. The architecture of Unreal is a major departure from that of most other games: Unreal is purely object-oriented (much like COM/ActiveX), in that it has a well-defined object model with support for high-level object oriented concepts such as the object graph, serialization, object lifetime, and polymorphism. Historically, most games have been designed monolithically, with their major functionality hardcoded and unexpandable at the object level, though many games, such as Doom and Quake, have proven to be very expandable at the content level. There is a major benefit to Unreal’s form of object-orientation: major new functionality and object types can be added to Unreal at runtime, and this expansion can take the form of subclassing, rather than (for example) by modifying a bunch of existing code. This form of extensibility is extremely powerful, as it encourages the Unreal community to create Unreal enhancements that all interoperate.
Object is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. Object is an abstract base class, in that it doesn’t do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).
Actor (expands Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment, and do other useful game-related things.
Pawn (expands Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.
Class (expands Object) is a special kind of object which describes a class of object. This may seem confusing at first: a class is an object, and a class describes certain objects. But, the concept is sound, and there are many cases where you will deal with Class objects. For example, when you spawn a new actor in UnrealScript, you can specify the new actor’s class with a Class object.
With UnrealScript, you can write code for any Object class, but 99% of the time, you will be writing code for a class derived from Actor. Most of the useful UnrealScript functionality is game-related and deals with actors.

The class declaration

Each script corresponds to exactly one class, and the script begins by declaring the class, the class’s parent, and any additional information that is relevent to the class. The simplest form is:
class MyClass expands MyParentClass;
Here I am declaring a new class named "MyClass", which inherets the functionality of "MyParentClass". Additionally, the class resides in the package named "MyPackage".
Each class inherets all of the variables, functions, and states from its parent class. It can then add new variable declarations, add new functions (or override the existing functions), add new states (or add functionality to the existing states).
The typical approach to class design in UnrealScript is to make a new class (for example a Minotaur monster) which expands an existing class that has most of the functionality you need (for example the Pawn class, the base class of all monsters). With this approach, you never need to reinvent the wheel – you can simply add the new functionality you want to customize, while keeping all of the existing functionality you don’t need to customize. This approach is especially powerful for implementing AI in Unreal, where the built-in AI system provides a tremendous amount of base functionality which you can use as building blocks for your custom creatures.

The class declaration can take several optional specifiers that affect the class:

- native: Says "this class uses behind-the-scenes C++ support". Unreal expects native classes to contain a C++ implementation in the DLL corresponding to the class's package. For example, if your package is named "Robots", Unreal looks in the "Robots.dll" for the C++ implementation of the native class, which is generated by the C++ IMPLEMENT_CLASS macro.
- Abstract: Declares the class as an "abstract base class". This prevents the user from adding actors of this class to the world in UnrealEd, because the class isn't meaningful on its own. For example, the "Pawn
  base class is abstract, while the "Brute" subclass is not abstract – you can place a Brute in the world, but you can't place a Pawn in the world.
- guid(a,b,c,d): Associates a globally unique identifier (a 128-bit number) with the class. This Guid is currently unused, but will be relevent when native COM support is later added to Unreal.
- transient: Says "objects belonging to this class should never be saved on disk". Only useful in conjunction with certain kinds of native classes which are non-persistent by nature, such as players or windows.
- config(section_name): If there are any configurable variables in the class (declared with "config" or "globalconfig"), causes those variables to be stored in a particular configuration file:
  - config(system): Uses the system configuration file, Unreal.ini for Unreal.
  - config(user): Uses the user configuration file, currently User.ini.
  - config(whatever): Uses the specified configuration file, for example "whatever.ini".

Config(configname)

<u>Variables</u>

<u>Simple Variables</u>

Here are some examples of instance variable declarations in UnrealScript:
var int a; // Declare an integer variable named "A".
var byte Table[64]; // Declare an array of 64 bytes named "Table".
var string[32] PlayerName; // Declare a max 32-character string.
var actor Other; // Declare a variable referencing an actor.
Variables can appear in two kinds of places in UnrealScript: instance variables, which apply to an entire object, appear immediately after the class declarations. Local variables appear within a function, and are only active while that function executes. Instance variables are declared with the "var" keyword. Local variables are declard with the "local" keyword.
Here are the basic variable types supported in UnrealScript:

- byte: A single-byte value ranging from 0 to 255.
- int: A 32-bit integer value.
- bool: A boolean value: either "true" or "false".
- float: A 32-bit floating point number.
- string: A string of characters.
- name: The name of an item in Unreal (such as the name of a function, state, class, etc). Names are stored as a 16-bit index into the global name table. Names correspond to simple strings of 1-31 characters. Names are not like strings: strings can be modified dynamically, but names can only take on predefined name values.
- Enumeration: A variable that can take on one of several predefined name values. For example, the ELightType enumeration defined in the Actor script describes a dynamic light and takes on a value like LT_None, LT_Pulse, LT_Strobe, and so on.
- Object and actor references: A variable that refers to another object or actor in the world. For example, the Pawn class has an "Enemy" actor reference that specifies which actor the pawn should be trying to attack. Object and actor references are very powerful tools, because they enable you to access the variables and functions of another actor. For example, in the Pawn script, you can write "Enemy.Damage(123)" to call your enemy's Damage function – resulting in the enemy taking damage. Object references may also contain a special value called "None", which is the equivalent of the C "NULL" pointer: it says "this variable doesn't refer to any object".

- Structs: Similar to C structures, UnrealScript structs let you create new variable types that contain sub-variables. For example, two commonly-used structs are "vector", which consists of an X, Y, and Z component; and "rotator", which consists of a pitch, yaw, and roll component.

Variables may also contain additional specifiers such as "const" that further describe the variable. Actually, there are quite a lot of specifiers which you wouldn't expect to see in a general-purpose programming language, mainly as a result of wanting UnrealScript to natively support many game- and environment-specific concepts:

- const: Advanced. Treats the contents of the variable as a constant. In UnrealScript, you can read the value of const variables, but you can't write to them. "Const" is only used for variables which the engine is responsible for updating, and which can't be safely updated from UnrealScript, such as an actor's Location (which can only be set by calling the MoveActor function).
- input: Advanced. Makes the variable accessible to Unreal's input system, so that input (such as button presses and joystick movements) can be directly mapped onto it. Only relevent with variables of type "byte" and "float".
- transient: Advanced. Declares that the variable is for temporary use, and isn't part of the object's persistent state. Transient variables are not saved to disk. Transient variables are initialized to zero when an actor is loaded.
- native: Advanced. Declares that the variable is loaded and saved by C++ code, rather than by UnrealScript.
- private: The variable is private, and may only be accessed by the class's script; no other classes (including subclasses) may access it.

Arrays are declared using the following syntax:

```
var int MyArray[20]; // Declares an array of 20 ints.
```

UnrealScript supports only single-dimensional arrays, though you can simulate multidimensional arrays by carrying out the row/column math yourself.

In UnrealScript, you can make an instance variable "editable", so that users can edit the variable's value in UnrealEd. This mechanism is responsible for the entire contents of the "Actor Properties" dialog in UnrealEd: everything you see there is simply an UnrealScript variable, which has been declared editable.

The syntax for declaring an editable variable is as follows:

```
var() int MyInteger; // Declare an editable integer in the default category.
var(MyCategory) bool MyBool; // Declare an editable integer in "MyCategory".
```

Object and actor reference variables

You can declare a variable that refers to an actor or object like this:

```
var actor A; // An actor reference.
var pawn P; // A reference to an actor in the Pawn class.
var texture T; // A reference to a texture object.
```

The variable "P" above is a reference to an actor in the Pawn class. Such a variable can refer to any actor that belongs to a subclass of Pawn. For example, P might refer to a Brute, or a Skaarj, or a Manta. It can be any kind of Pawn. However, P can never refer to a Trigger actor (because Trigger is not a subclass of Pawn).

One example of where it's handy to have a variable refering to an actor is the Enemy variable in the Pawn class, which refers to the actor which the Pawn is trying to attack.

When you have a variable that refers to an actor, you can access that actor's variables, and call its functions. For example:

```
// Declare two variables that refer to a pawns.
var pawn P, Q;

// Here is a function that makes use of P.
// It displays some information about P.
function MyFunction()
{
    // Set P's enemy to Q.
    P.Enemy = Q;

    // Tell P to play his running animation.
    P.PlayRunning();
}
```

Variables that refer to actors always either refer to a valid actor (any actor that actually exists in the level), or they contain the value "None". None is equivalent to the C/C++ "NULL" pointer. However, in UnrealScript, it is safe to access variables and call functions with a "None" reference; the result is always zero.
Note that an object or actor reference "points to" another actor or object, it doesn't "contain" an actor or object. The C equivalent of an actor reference is a pointer to an object in the AActor class (in C, you'd say an AActor*). For example, you could have two monsters in the world, Bob and Fred, who are fighting each other. Bob's "Enemy" variable would "point to" Fred, and Fred's "Enemy" variable would "point to" Bob.
Unlike C pointers, UnrealScript object references are always safe and infallible. It is impossible for an object reference to refer to an object that doesn't exist or is invalid (other than the special-case "None" value). In UnrealScript, when an actor or object is destroyed, all references to it are automatically set to "None".

Class Reference Variables

In Unreal, classes are objects just like actors, textures, and sounds are objects. Class objects belong to the class named "class". Now, there will often be cases where you'll want to store a reference to a class object, so that you can spawn an actor belonging to that class (without knowing what the class is at compile-time). For example:

```
var() class C;
var actor A;
A = Spawn( C ); // Spawn an actor belonging to some arbitrary class C.
```

Now, be sure not to confuse the roles of a class C, and an object O belonging to class C. To give a really shaky analogy, a class is like a pepper grinder, and an object is like pepper. You can use the pepper grinder (the class) to create pepper (objects of that class) by turning the crank (calling the Spawn function)...BUT, a pepper grinder (a class) is not pepper (an object belonging to the class), so you MUST NOT TRY TO EAT IT!
When declaring variables that reference class objects, you can optionally use the special class<classlimitor> syntax to limit the variable to only containing references to classes which expand a given superclass. For example, in the declaration:

```
var class<actor> ActorClass;
```

The variable ActorClass may only reference a class that expands the "actor" class. This is useful for improving compile-time type checking. For example, the Spawn function takes a class as a parameter, but only makes sense when the given class is a subclass of Actor, and the class<classlimitor> syntax causes the compiler to enforce that requirement.
As with dynamic object casting, you can dynamically cast classes like this:

```
class<actor>( SomeFunctionCall() )
```

Enumerations

Enumerations exist in UnrealScript as a convenient way to declare variables that can contain "one of" a bunch of keywords. For example, the actor class contains the enumeration EPhysics which describes the physics which Unreal should apply to the actor. This can be set to one of the predefined values like PHYS_None, PHYS_Walking, PHYS_Falling, and so on.
Internally, enumerations are stored as byte variables. In designing UnrealScript, enumerations were not seen as a necessity, but it makes code so much easier to read to see that an actor's physics mode is being set to "PHYS_Swimming" than (for example) "3".
Here is sample code that declares enumerations.

```
// Declare the EColor enumeration, with three values.
enum EColor
{
    CO_Red,
    CO_Green,
    CO_Blue
};

// Now, declare two variables of type EColor.
var EColor ShirtColor, HatColor;

// Alternatively, you can declare variables and
// enumerations together like this:
var enum EFruit
{
    FRUIT_Apple,
    FRUIT_Orange,
    FRUIT_Bannana
} FirstFruit, SecondFruit;
```

In the Unreal source, we always declare enumeration values like LT_Steady, PHYS_Falling, and so on, rather than as simply "Steady" or "Falling". This is just a matter of programming style, and is not a requirement of the language.

UnrealScript only recognizes unqualified enum tags (like FRUIT_Apple) in classes where the enumeration was defined, and in its subclasses. If you need to refer to an enumeration tag defined somewhere else in the class hierarchy, you must "qualify it":

```
FRUIT_Apple // If Unreal can't find this enum tag...
EFruit.FRUIT_Apple // Then qualify it like this.
```

Structs

An UnrealScript struct is a way of cramming a bunch of variables together into a new kind of super-variable called a struct. UnrealScript structs are just like C structs, in that they can contain any simple variables or arrays.

You can declare a struct as follows:

```
// A point or direction vector in 3D space.
struct Vector
{
    var float X;
    var float Y;
    var float Z
};
```

Once you declare a struct, you are ready to start declaring specific variables of that struct type:

```
// Declare a bunch of variables of type Vector.
var Vector Position;
var Vector Destination;
```

To access a component of a struct, use code like the following.

```
function MyFunction()
{
    Local Vector A, B, C;

    // Add some vectors.
    C = A + B;

    // Add just the x components of the vectors.
    C.X = A.X + B.X;

    // Pass vector C to a function.
    SomeFunction( C );

    // Pass certain vector components to a function.
    OtherFunction( A.X, C.Z );
}
```

You can do anything with Struct variables that you can do with other variables: you can assign variables to them, you can pass them to functions, and you can access their components.

There are several Structs defined in the Object class which are used throughout Unreal. You should become familiar with their operation, as they are fundamental building blocks of scripts:

- Vector: A unique 3D point or vector in space, with an X, Y, and Z component.

- Plane: Defines a unique plane in 3D space. A plane is defined by its X, Y, and Z components (which are assumed to be normalized) plus its W component, which represents the distance of the plane from the origin, along the plane's normal (which is the shortest line from the plane to the origin).
- Rotation: A rotation defining a unique orthogonal coordinate system. A rotation contains Pitch, Yaw, and Roll components.
- Coords: An arbitrary coordinate system in 3D space.
- Color: An RGB color value.
- Region: Defines a unique convex region within a level.

Expressions

Constants

In UnrealScript, you can specify constant values of nearly all data types:

- Integer and byte constants are specified with simple numbers, for example: 123
- If you must specify an integer or byte constant in hexidecimal format, use i.e.: 0x123
- Floating point constants are specified with decimal numbers like: 456.789
- String constants must be enclosed in double quotes, for example: "MyString"
- Name constants must be enclosed in single quotes, for example 'MyName'
- Vector constants contain X, Y, and Z values like this: Vect(1.0,2.0,4.0)
- Rotation constants contant Pitch, Yaw, and Roll values like this: Rot(0x8000,0x4000,0)
- The "None" constant refers to "no object" (or equivalantly, "no actor").
- The "Self" constant refers to "this object" (or equivalantly, "this actor"), i.e. the object whose script is executing.
- General object constants are specified by the object type followed by the object name in single quotes, for example: texture 'Default'
- EnumCount gives you the number of elements in an enumeration, for example: EnumCount(ELightType)
- ArrayCount gives you the number of elements in an array, for example: ArrayCount(Touching)

You can use the "const" keyword to declare constants which you can later refer to by name. For example:

```
const LargeNumber=123456;
const PI=3.14159;
const MyName="Tim";
const Northeast=Vect(1.0,1.0,0.0);
```

Constants can be defined within classes or within structs.

To access a constant which was declared in another class, use the "classname.constname" syntax, for example:

Pawn.LargeNumber

Expressions

To assign a value to a variable, use "=" like this:

```
function Test()
{
    local int i;
    local string[80] s;
    local vector v, q;

    i = 10; // Assign a value to integer variable i.
    s = "Hello!"; // Assign a value to string variable s.
    v = q; // Copy value of vector q to v.
}
```

In UnrealScript, whenever a function or other expression requires a certain type of data (for example, an "float"), and you specify a different type of data (for example, an "int), the compiler will try to convert the value you give to the proper type. Conversions among all the numerical data types (byte, int, and float) happen automatically, without any work on your part.

UnrealScript is also able to many other built-in data types to other types, if you explicitly convert them in code. The syntax for this is:

```
function Test()
{
    local int i;
    local string[80] s;
    local vector v, q;
    local rotation r;

    s = string(i); // Convert integer i to a string, and assign it to s.
    s = string(v); // Convert vector v to a string, and assign it to s.
    v = q + vector(r); // Convert rotation r to a vector, and add q.
}
```

Here is the complete set of non-automatic conversions you can use in UnrealScript:

- String to Byte, Int, Float: Tries to convert a string like "123" to a value like 123. If the string doesn’t represent a value, the result is 0.
- Byte, Int, Float, Vector, Rotation to String: Converts the number to its textual representation.
- String to Vector, Rotation: Tries to parse the vector or rotation’s textual representation.
- String to Bool: Converts the case-insensitive words "True" or "False" to True and False; converts any non-zero value to True; everything else is False.
- Bool to String: Result is either "True" or "False".
- Byte, Int, Float, Vector, Rotation to Bool: Converts nonzero values to True; zero values to False.
- Bool to Byte, Int, Float: Converts True to 1; False to 0.
- Name to String: Converts the name to the text equivalant.
- Rotation to Vector: Returns a vector facing "forward" according to the rotation.
- Vector to Rotation: Returns a rotation pitching and yawing in the direction of the vector; roll is zero.
- Object (or Actor) to Int: Returns an integer that is guaranteed unique for that object.
- Object (or Actor) to Bool: Returns False if the object is None; False otherwise.
- Object (or Actor) to String: Returns a textual representation of the object.

Converting object references among classes

Just like the conversion functions above, which convert among simple datatypes, in UnrealScript you can convert actor and object references among various types. For example, all actors have a variable named "Target", which is a reference to another actor. Say you are writing a script where you need to check and see if your Target belongs to the "Pawn" actor class, and you need to do something special with your target that only makes sense when it’s a pawn -- for example, you need to call one of the Pawn functions. The actor cast operators let you do this. Here’s an example:

```
var actor Target;
//...

function TestActorConversions()
{
    local Pawn P;

    // See if my target is a Pawn.
    P = Pawn(Target);
    if( P != None )
    {
            // Target is a pawn, so set its Enemy to Self.
            P.Enemy = Self;
    }
    else
    {
            // Target is not a pawn.
    }
}
```

To perform an actor conversion, type the class name followed by the actor expression you wish to convert, in parenthesis. Such a conversion will either succeed or fail based on whether the conversion is sensible. In the above example, if your Target is referencing a Trigger object rather than a pawn, the expression Pawn(Target) will return "None", since a Trigger can't be converted to a Pawn. However, if your Target is referencing a Brute object, the conversion will successfully return the Brute, because Brute is a subclass of Pawn.

Thus, actor conversions have two purposes: First, you can use them to see if a certain actor reference belongs to a certain class. Second, you can use them to convert an actor reference from one class to a more specific class. Note that these conversions don't affect the actor you're converting at all -- they just enable UnrealScript to treat the actor reference as if it were a more specific type.

Another example of conversions lies in the Inventory script. Each Inventory actor is owned by a Pawn, even though its Owner variable can refer to any Actor. So a common theme in the Inventory code is to cast Owner to a Pawn, for example:

```
// Called by engine when destroyed.
function Destroyed()
{
    // Remove from owner's inventory.
    if( Pawn(Owner)!=None )
            Pawn(Owner).DeleteInventory( Self );
}
```

<u>Functions</u>

<u>Declaring Functions</u>

In UnrealScript, you can declare new functions and write new versions of existing functions. Functions can take one or more parameters (of any variable type UnrealScript supports), and can optionally return a value. Though most functions are written directly in UnrealScript, you can also declare functions that can be called from UnrealScript, but which are implemented in C++ and reside in a DLL. The Unreal technology supports all possible combinations of function calling: The C++ engine can call script functions; script can call C++ functions; and script can call script.

Here is a simple function declaration. This function takes a vector as a parameter, and returns a floating point number:

```
// Function to compute the size of a vector.
function float VectorSize( vector V )
{
    return sqrt( V.X * V.X + V.Y * V.Y + V.Z * V.Z );
}
```

The word "function" always precedes a function declaration. It is followed by the optional return type of the function (in this case, "float"), then the function name, and then the list of function parameters enclosed in parenthesis.

When a function is called, the code within the brackets is executed. Inside the function, you can declare local variables (using the "local" keyword"), and execute any UnrealScript code. The optional "return" keyword causes the function to immediately return a value.

You can pass any UnrealScript types to a function (including arrays), and a function can return any type.

By default, any local variables you declare in a function are initialized to zero.

Function calls can be recursive. For example, the following function computes the factorial of a number:

```
// Function to compute the factorial of a number.
function int Factorial( int Number )
{
    if( Number <= 0 )
            return 1;
    else
            return Number * Factorial( Number – 1 );
}
```

Some UnrealScript functions are called by the engine whenever certain events occur. For example, when an actor is touched by another actor, the engine calls its "Touch" function to tell it who is touching it. By writing a custom "Touch" function, you can take special actions as a result of the touch occuring:

```
// Called when something touches this actor.
function Touch( actor Other )
{
    Log( "I was touched!")
    Other.Message( "You touched me!" );
}
```
The above function illustrates several things. First of all, the function writes a message to the log file using the "Log" command (which is the equivalent of Basic's "print" command and C's "printf"). Second, it calls the "Message" function residing in the actor Other. Calling functions in other actors is a common action in UnrealScript, and in object-oriented languages like Java in general, because it provides a simple means for actors to communicate with each other.

<u>Function parameter specifiers</u>

When you normally call a function, UnrealScript makes a local copy of the parameters you pass the function. If the function modifies some of the parameters, those don't have any effect on the variables you passed in. For example, the following program:
```
function int DoSomething( int x )
{
    x = x * 2;
    return x;
}
function int DoSomethingElse()
{
    local int a, b;

    a = 2;
    log( "The value of a is " $ a );

    b = DoSomething( a );
    log( "The value of a is " $ a );
    log( "The value of b is " $ b );
}
```
Produces the following output when DoSomethingElse is called:
```
The value of a is 2
The value of a is 2
The value of b is 4
```
In other words, the function DoSomething was futzing with a local copy of the variable "a" which was passed to it, and it was not affecting the real variable "a".

The "out" specified lets you tell a function that it should actually modify the variable that is passed to it, rather than making a local copy. This is useful, for example, if you have a function that needs to return several values to the caller. You can juse have the caller pass several variables to the function which are "out" values. For example:
```
// Compute the minimum and maximum components of a vector.
function VectorRange( vector V, out float Min, out float Max )
{
    // Compute the minimum value.
    if ( V.X<V.Y && V.X<V.Z ) Min = V.X;
    else if( V.Y<V.Z ) Min = V.Y;
    else Min = V.Z;

    // Compute the maximum value.
    if ( V.X>V.Y && V.X>V.Z ) Max = V.X;
    else if( V.Y>V.Z ) Max = V.Y;
    else Max = V.Z;
}
```

Without the "out" keyword, it would be painful to try to write functions that had to return more than one value.
With the "optional" keyword, you can make certain function parameters optional, as a convenience to the caller. For UnrealScript functions, optional parameters which the caller doesn't specify are set to zero. For native functions, the default values of optional parameters depends on the function. For example, the Spawn function takes an optional location and rotation, which default to the spawning actor's location and rotation.
The "coerce" keyword forces the caller's parameters to be converted to the specified type (even if UnrealScript normally would not perform the conversion automatically). This is useful for functions that deal with strings, so that the parameters are automatically converted to strings for you.

Function overriding

"Function overriding" refers to writing a new version of a function in a subclass. For example, say you're writing a script for a new kind of monster called a Demon. The Demon class, which you just created, expands the Pawn class. Now, when a pawn sees a player for the first time, the pawn's "SeePlayer" function is called, so that the pawn can start attacking the player. This is a nice concept, but say you wanted to handle "SeePlayer" differently in your new Demon class. How do you do this? Function overriding is the answer.
To override a function, just cut and paste the function definition from the parent class into your new class. For example, for SeePlayer, you could add this to your Demon class.

```
// New Demon class version of the Touch function.
function SeePlayer( actor SeenPlayer )
{
    log( "The demon saw a player" );
    // Add new custom functionality here…
}
```

Function overriding is the key to creating new UnrealScript classes efficiently. You can create a new class that expands on an existing class. Then, all you need to do is override the functions which you want to be handled differently. This enables you to create new kinds of objects without writing gigantic amounts of code.
Several functions in UnrealScript are declared as "final". The "final" keyword (which appears immediately before the word "function") says "this function cannot be overridden by child classes". This should be used in functions which you know nobody would want to override, because it results in faster script code. For example, say you have a "VectorSize" function that computes the size of a vector. There's absolutely no reason anyone would ever override that, so declare it as "final". On the other hand, a function like "Touch" is very context-dependent and should not be final.

Advanced function specifiers

Static: A static function acts like a C global function, in that it can be called without having a reference to an object of the class. Static functions can call other static functions, and can access the default values of variables. Static functions cannot call non-static functions and they cannot access instance variables (since they are not executed with respect to an instance of an object). Unlike languages like C++, static functions are virtual and can be overridden in child classes. This is useful in cases where you wish to call a static function in a variable class (a class not known at compile time, but referred to by a variable or an expression).
Singular: The "singular" keyword, which appears immediately before a function declaration, prevents a function from calling itself recursively. The rule is this: If a certain actor is already in the middle of a singular function, any subsequent calls to singular functions will be skipped over. This is useful in avoiding infinite-recursive bugs in some cases. For example, if you try to move an actor inside of your "Bump" function, there is a good chance that the actor will bump into another actor during its move, resulting in another call to the "Bump" function, and so on. You should be very careful in avoiding such behaviour, but if you can't write code with complete confidence that you're avoiding such potential recursive situations, use the "singular" keyword.
Native: You can declare UnrealScript functions as "native", which means that the function is callable from UnrealScript, but is actually written (elsewhere) in C++. For example, the Actor class contains a lot of native function definitions, such as:

```
native(266) final function bool Move( vector Delta );
```

The number inside the parenthesis after the "native" keyword corresponds to the number of the function as it was declared in C++ (using the AUTOREGISTER_NATIVE macro). The native function is expected to reside in the DLL named identically to the package of the class containing the UnrealScript definition.

Latent: Declares that an native function is latent, meaning that it can only be called from state code, and it may return after some game-time has passed.
Iterator: Declares that an native function is an iterator, which can be used to loop through a list of actors using the "foreach" command.
Simulated: Declares that a function may execute on the client-side when an actor is either a simulated proxy or an autonomous proxy. All functions that are both native and final are automatically simulated as well.
Operator, PreOperator, PostOperator: These keywords are for declaring a special kind of function called an operator (equivalent to C++ operators). This is how UnrealScript knows about all of the built-in operators like "+", "-", "==", and "||". I'm not going into detail on how operators work in this document, but the concept of operators is similar to C++, and you can declare new operator functions and keywords as UnrealScript functions or native functions.
Event: The "event" keyword has the same meaning to UnrealScript as "function". However, when you export a C++ header file from Unreal using "unreal -make -h", UnrealEd automatically generates a C++ -> UnrealScript calling stub for each "event". This is a much cleaner replacement for the old "PMessageParms" struct, because it automatically keeps C++ code synched up with UnrealScript functions and eliminates the possibility of passing invalid parameters to an UnrealScript function. For example, this bit of UnrealScript code:

```
event Touch( Actor Other )
{ ... }
```

Generates this piece of code in EngineClasses.h:

```
void Touch(class AActor* Other)
{
   FName N("Touch",FNAME_Intrinsic);
   struct {class AActor* Other; } Parms;
   Parms.Other=Other;
   ProcessEvent(N,&Parms);
}
```

Thus enabling you to call the UnrealScript function from C++ like this:

```
AActor *SomeActor, *OtherActor;
Actor->Touch(OtherActor);
```

Program Structure

UnrealScript supports all the standard flow-control statements of C/C++/Java:

For Loops

"For" loops let you cycle through a loop as long as some condition is met. For example:

```
// Example of "for" loop.
function ForExample()
{
    local int i;
    log( "Demonstrating the for loop" );
    for( i=0; i<4; i++ )
    {
            log( "The value of i is " $ i );
    }
    log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the for loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

In a for loop, you must specify three expressions separated by semicolons. The first expression is for initializing a variable to its starting value. The second expression gives a condition which is checked before

each iteration of the loop executes; if this expression is true, the loop executes. If it's false, the loop terminates. The third condition gives an expression which increments the loop counter.
Though most "for" loop expressions just update a counter, you can also use "for" loops for more advanced things like traversing linked lists, by using the appropriate initialization, termination, and increment expressions.
In all of the flow control statements, you can either execute a single statement, without brackets, as follows:

```
for( i=0; i<4; i++ )
    log( "The value of i is " $ i );
```

Or you can execute multiple statements, surrounded by brackets, like this:

```
for( i=0; i<4; i++ )
{
    log( "The value of i is" );
    log( i );
}
```

<u>Do-While Loops</u>

"Do"-"Until" loops let you cycle through a loop while some ending expression is true. Note that Unreal's do-until syntax differs from C/Java (which use do-while).

```
// Example of "do" loop.
function DoExample()
{
    local int i;
    log( "Demonstrating the do loop" );
    do
    {
            log( "The value of i is " $ i );
            i = i + 1;
    } until( i == 4 );
    log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the do loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

<u>While Loops</u>

"While" loops let you cycle through a loop while some starting expression is true.

```
// Example of "while" loop.
function WhileExample()
{
    local int i;
    log( "Demonstrating the while loop" );
    while( i < 4 )
    {
            log( "The value of i is " $ i );
            i = i + 1;
    }
    log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the do loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

Break

The "break" command exits out of the nearest loop ("For", "Do", or "While").

```
// Example of "while" loop.
function WhileExample()
{
    local int i;
    log( "Demonstrating break" );
    for( i=0; i<10; i++ )
    {
        if( i == 3 )
            break;
        log( "The value of i is " $ i );
    }
    log( "Completed with i=" $ i );
}
```

The output of this loop is:

```
Demonstrating break
The value of i is 0
The value of i is 1
The value of i is 2
Completed with i=3
```

Goto

The "Goto" command goes to a label somewhere in the current function or state.

```
// Example of "goto".
function GotoExample()
{
    log( "Starting GotoExample" );
    goto Hither;
Yon:
    log( "At Yon" );
    goto Elsewhere;
Hither:
    log( "At Hither" );
    goto Yon;
Elsewhere:
    log( "At Elsewhere" );
}
```

The output is:

```
Starting GotoExample
At Hither
At Yon
At Elsewhere
```

Conditional Statements

"If", "Else If", and "Else" let you execute code if certain conditions are met.

```
// Example of simple "if".
if( LightBrightness < 20 )
    log( "My light is dim" );

// Example of "if-else".
if( LightBrightness < 20 )
```

```
    log( "My light is dim" );
else
    log( "My light is bright" );

// Example if "if-else if-else".
if( LightBrightness < 20 )
    log( ‘My light is dim" );
else if( LightBrightness < 40 )
    log( "My light is medium" );
else if( LightBrightness < 60 )
    log( "My light is kinda bright" );
else
    log( "My light is very bright" );

// Example if "if" with brackets.
if( LightType == LT_Steady )
{
    log( "Light is steady" );
}
else
{
    log( "Light is not steady" );
}
```

<u>Case Statements</u>

"Switch", "Case", "Default", and "Break" let you handle lists of conditions easily.

```
// Example of switch-case.
function TestSwitch()
{
    // Executed one of the case statements below, based on
    // the value in LightType.
    switch( LightType )
    {
            case LT_None:
                    log( "There is no lighting" );
                    break;
            case LT_Steady:
                    log( "There is steady lighting" );
                    break;
            case LT_Backdrop:
                    log( "There is backdrop lighting" );
                    break;
            default:
                    log( "There is dynamic" );
                    break;
    }
}
```

A "switch" statement consists of one or more "case" statements, and an optional "default" statement. After a switch statement, execution goes to the matching "case" statement if there is one; otherwise execution goes to the "default" statement; otherwise execution continues past the end of the "select" statement.

After you write code following a "case" label, you must use a "break" statement to cause execution to go past the end of the "switch" statement. If you don’t use a "break", execution "falls through" to the next "case" handler.

States

Overview of States

Historically, game programmers have been using the concept of states ever since games evolved past the "pong" phase. States (and what is known as "state machine programming") are a natural way of making complex object behaviour manageable. However, before UnrealScript, states have not been supported at the language level, requiring developers to create C/C++ "switch" statements based on the object's state. Such code was difficult to write and update.

UnrealScript supports states at the language level.

In UnrealScript, each actor in the world is always in one and only one state. Its state reflects the action it wants to perform. For example, moving brushes have several states like "StandOpenTimed" and "BumpOpenTimed". Pawns have several states such as "Dying", "Attacking", and "Wandering".

In UnrealScript, you can write functions and code which exist in a particular state. These functions are only called when the actor is in that state. For example, say you're writing a monster script, and you're contemplating how to handle the "SeePlayer" function. When you're wandering around, you want to attack the player you see. When you're already attacking the player, you want to continue on uninterrupted.

The easiest way to do this is by defining several states (Wandering and Attacking), and writing a different version of "Touch" in each state. UnrealScript supports this.

Before delving deeper into states, you need to understand that there are two major benefits to states, and one complication:

- Benefit: States provide a simple way to write state-specific functions, so that you can handle the same function in different ways, depending on what the actor is doing.
- Benefit: With a state, you can write special "state code", using all of the regular UnrealScript commands plus several special functions known as "latent functions". A latent function is a function which executes "slowly", and may return after a certain amount of "game time" has passed. This enables you to perform time-based programming – a major benefit which neither C, C++, nor Java offer. Namely, you can write code in the same way you conceptualize it; for example, you can write a script that says the equivalent of "open this door; pause 2 seconds; play this sound effect; open that door; release that monster and have it attack the player". You can do this with simple, linear code, and the Unreal engine takes care of the details of managing the time-based execution of the code.
- Complication: Now that you can have functions (like "Touch") overridden in multiple states as well as in child classes, you have the burden of figuring out exactly which "Touch" function is going to be called in a specific situation. UnrealScript provides rules which clearly delineate this process, but it is something you must be aware of if you create complex hierarchies of classes and states.

Here is an example of states from the TriggerLight script:

```
// Trigger turns the light on.
state() TriggerTurnsOn
{
    function Trigger( actor Other, pawn EventInstigator )
    {
            Trigger = None;
            Direction = 1.0;
            Enable( 'Tick' );
    }
}

// Trigger turns the light off.
state() TriggerTurnsOff
{
    function Trigger( actor Other, pawn EventInstigator )
    {
            Trigger = None;
            Direction = -1.0;
            Enable( 'Tick' );
    }
}
```

Here you are declaring two different states (TriggerTurnsOn and TriggerTurnsOff), and you're writing a different version of the Trigger function in each state. Though you could pull off this implementation without states, using states makes the code far more modular and expandable: in UnrealScript, you can easily subclass an existing class, add new states, and add new functions. If you had tried to do this without states, the resulting code would be more difficult to expand later.

A state can be declared as editable, meaning that the user can set an actor's state in UnrealEd, or not. To declare an editable state, do the following:

```
state() MyState
{
    //...
}
```

To declare a non-editable state, do this:

```
state MyState
{
    //...
}
```

You can also specify the automatic, or initial state that an actor should be in by using the "auto" keyword. This causes all new actors to be placed in that state when they first are activated:

```
auto state MyState
{
    //...
}
```

<u>State Labels and Latent Functions</u>

In addition to functions, a state can contain one or more labels followed by UnrealScript code. For example:

```
auto state MyState
{
Begin:
    Log( "MyState has just begun!" );
    Sleep( 2.0 );
    Log( "MyState has finished sleeping" );
    goto Begin;
}
```

The above state code prints the message "MyState has just begun!", then it pauses for two seconds, then it prints the message "MyState has finished sleeping". The interesting thing in this example is the call to the latent function "Sleep": this function call doesn't return immediately, but returns after a certain amount of game time has passed. Latent functions can only be called from within state code, and not from within functions. Latent functions let you manage complex chains of events which include the passage of time.

All state code begins with a label definition; in the above example the label is named "Begin". The label provides a convenient entry point into the state code. You can use any label name in state code, but the "Begin" label is special: it is the default starting point for code in that state.

There are three main latent functions available to all actors:

- Sleep(float Seconds) pauses the state execution for a certain amount of time, and then continues.
- FinishAnim() waits until the current animation sequence you're playing completes, and then continues. This function makes it easy to write animation-driven scripts, scripts whose execution is governed by mesh animations. For example, most of the AI scripts are animation-driven (as opposed to time-driven), because smooth animation is a key goal of the AI system.
- FinishInterpolation() waits for the current InterpolationPoint movement to complete, and then continues.

The Pawn class defines several important latent functions for actions such as navigating through the world and short-term movement. See the separate AI docs for descriptions of their usage.

Three native UnrealScript functions are particularly useful when writing state code:

- The "Goto" function (similar to the C/C++/Basic goto) within a state causes the state code to continue executing at a different label.
- The special Goto(") command within a state causes the state code execution to stop. State code execution doesn't continue until you go to a new state, or go to a new label within the current state.
- The "GotoState" function causes the actor to go to a new state, and optionally continue at a specified label (if you don't specify a label, the default is the "Begin" label). You can call GotoState from within state code, and it goes to the destination immediately. You can also call GotoState from within any function in the actor, but that does not take effect immediately: it doesn't take effect until execution returns back to the state code.

Here is an example of the state concepts discussed so far:

```
// This is the automatic state to execute.
auto state Idle
{
    // When touched by another actor…
    function Touch( actor Other )
    {
            log( "I was touched, so I'm going to Attacking" );
            GotoState( 'Attacking' );
            Log( "I have gone to the Attacking state" );
    }
Begin:
    log( "I am idle…" );
    sleep( 10 );
    goto 'Begin';
}

// Attacking state.
state Attacking
{
Begin:
    Log( "I am executing the attacking state code" );
    //...
}
```

When you run this program and then go touch the actor, you will see:

```
I am idle...
I am idle...
I am idle...
I was touched, so I'm going to Attacking
I have gone to the Attacking state
I am executing the attacking state code
```

Make sure you understand this important aspect of GotoState: When you call GotoState from within a function, it does not go to the destination immediately, rather it goes there once execution returns back to the state code.

State inheretance and scoping rules

In UnrealScript, when you subclass an existing class, your new class inherets all of the variables, functions and states from its parent class. This is well-understood.

However, the addition of the state abstraction to the UnrealScript programming model adds additional twists to the inheretance and scoping rules. The complete inheretance rules are:

- A new class inherets all of the variables from its parent class.
- A new class inherents all of its parent class's non-state functions. You can override any of those inhereted non-state functions. You can add entirely new non-state functions.
- A new class inherets all of its parent class's states, including the functions and labels within those states. You can override any of the inhereted state functions, and you can override any of the inhereted state labels, you can add new state functions, and you can add new state labels.

Here is an example of all the overriding rules:

```
// Here is an example parent class.
class MyParentClass expands Actor;

// A non-state function.
function MyInstanceFunction()
{
    log( "Executing MyInstanceFunction" );
}

// A state.
state MyState
{
    // A state function.
    function MyStateFunction()
    {
            Log( "Executing MyStateFunction" );
    }
// The "Begin" label.
Begin:
    Log("Beginning MyState");
}

// Here is an example child class.
class MyChildClass expands MyParentClass;

// Here I'm overriding a non-state function.
function MyInstanceFunction()
{
    Log( "Executing MyInstanceFunction in child class" );
}

// Here I'm redeclaring MyState so that I can override MyStateFunction.
state MyState
{
    // Here I'm overriding MyStateFunction.
    function MyStateFunction()
    {
            Log( "Executing MyStateFunction" );
    }
// Here I'm overriding the "Begin" label.
Begin:
    Log( "Beginning MyState in MyChildClass" );
}
```

When you have a function that is implemented globally, in one or more states, and in one or more parent classes, you need to understand which version of the function will be called in a given context. The scoping rules, which resolves these complex situations, are:

- If the object is in a state, and an implementation of the function exists somewhere in that state (either in the actor's class or in some parent class), the most-derived state version of the function is called.
- Otherwise, the most-derived non-state version of the function is called.

Advanced state programming

If a state doesn't override a state of the same name in the parent class, then you can optionally use the "expands" keyword to make the state expand on an existing state in the current class. This is useful, for example, in a situatio where you have a group of similar states (such as MeleeAttacking and RangeAttacking) which have a lot of functionality in common. In this case you could declare a base Attacking state as follows:

```
// Base Attacking state.
state Attacking
{
    // Stick base functions here...
}

// Attacking up-close.
state MeleeAttacking expands Attacking
{
    // Stick specialized functions here...
}

// Attacking from a distance.
state RangeAttacking expands Attacking
{
    // Stick specialized functions here...
}
```

A state can optionally use the "ignores" specifier to ignore functions while in a state. The syntax for this is:

```
// Declare a state.
state Retreating
{
    // Ignore the following messages...
    ignores Touch, UnTouch, MyFunction;

    // Stick functions here...
}
```

You can tell what specific state an actor is in from its "state" variable, a variable of type "name".

It is possible for an actor to be in "no state" by using GotoState(‘’). When an actor is in "no state", only its global (non-state) functions are called.

Whever you use the GotoState command to set an actor’s state, the engine can call two special notification functions, if you have defined them: EndState() and BeginState(). EndState is called in the current state immediately before the new state is begun, and BeginState is called immediately after the new state begins. These functions provide a convenient place to do any state-specific initialization and cleanup which your state may require.

Language Functionality

Built-in operators and their precedence

UnrealScript provides a wide variety of C/C++/Java-style operators for such operations as adding numbers together, comaring values, and incrementing variables. The complete set of operators is defined in Object.u, but here is a recap. Here are the standard operators, in order of precedence. Note that all of the C style operators have the same precedence as they do in C.

Operator	Types it applies to	Meaning
$	string	String concatenation
*=	byte, int, float, vector, rotation	Multiply and assign
/=	byte, int, float, vector, rotation	Divide and assign
+=	byte, int, float, vector	Add and assign
-=	byte, int, float, vector	Subtract and assign
||	bool	Logical or
&&	bool	Logical and
&	int	Bitwise and
|	int	Bitwise or

^	int	Bitwise exlusive or
!=	All	Compare for inequality
==	All	Compare for equality
<	byte, int, float, string	Less than
>	byte, int, float, string	Greater than
<=	byte, int, float, string	Less than or equal to
>=	byte, int, float, string	Greater than or equal to
~=	float, string	Approximate equality (within 0.0001), case-insensitive equality.
<<	int, vector	Left shift (int), Forward vector transformation (vector)
>>	int, vector	Right shift (int), Reverse vector transformation (vector)
+	byte, int, float, vector	Add
-	byte, int, float, vector	Subtract
%	float	Modulo (remainder after division)
*	byte, int, float, vector, rotation	Multiply
/	byte, int, float, vector, rotation	Divide
Dot	vector	Vector dot product
Cross	vector	Vector cross product
**	float	Exponentiation

The above table lists the operators in order of precedence (with operators of the same precedence grouped together). When you type in a complex expression like "1*2+3*4", UnrealScript automatically groups the operators by precedence. Since multiplication has a higher precedence than addition, the expression is evaluated as "(1*2)+(3*4)".

The "&&" (logical and) and "||" (logical or) operators are short-circuited: if the result of the expression can be determined solely from the first expression (for example, if the first argument of && is false), the second expression is not evaluated.

In addition, UnrealScript supports the following unary operators:

- ! (bool) Logical not.
- - (int, float) negation.
- ~ (int) bitwise negation.
- ++, -- Decrement (either before or after a variable).

General purpose functions

Integer functions:

- int Rand(int Max); Returns a random number from 0 to Max-1.
- int Min(int A, int B); Returns the minimum of the two numbers.
- int Max(int A, int B); Returns the maximum of the two numbers.
- int Clamp(int V, int A, int B); Returns the first number clamped to the interval from A to B.

Floating point functions:

- float Abs(float A); Returns the absolute value of the number.
- float Sin(float A); Returns the sine of the number expressed in radius.
- float Cos(float A); Returns the cosine of the number expressed in radians.
- float Tan(float A); Returns the tangent of the number expressed in radians.
- float Atan(float A); Returns the inverse tangent of the number expressed in radians.
- float Exp(float A); Returns the constant "e" raised to the power of A.
- float Loge(float A); Returns the log (to the base "e") of A.
- float Sqrt(float A); Returns the square root of A.
- float Square(float A); Returns the square of A = A*A.
- float FRand(); Returns a random number from 0.0 to 1.0.
- float FMin(float A, float B); Returns the minimum of two numbers.

- float FMax(float A, float B); Returns the maximum of two numbers.
- float FClamp(float V, float A, float B); Returns the first number clamped to the interval from A to B.
- float Lerp(float Alpha, float A, float B); Returns the linear interpolation between A and B.
- float Smerp(float Alpha, float A, float B); Returns an Alpha-smooth nonlinear interpolation between A and B.

Unreal's string functions have a distinct Basic look and feel:

- int Len(coerce string[255] S); Returns the length of a string.
- int InStr(coerce string[255] S, coerce string[255] t); Returns the offset into the first string of the second string if it exists, or -1 if not.
- string[255] Mid (coerce string[255] S, int i, optional int j); Returns the middle part of the string S, starting and character i and including j characters (or all of them if j is not specified).
- string[255] Left (coerce string[255] S, int i); Returns the i leftmost characters of s.
- string[255] Right (coerce string[255] S, int i); Returns the i rightmost characters of s.
- string[255] Caps (coerce string[255] S); Returns S converted to uppercase.

Vector functions:

- float Size(vector A); Returns the euclidean size of the vector (the square root of the sum of the components squared).
- vector Normal(vector A); Returns a vector of size 1.0, facing in the direction of the specified vector.
- Invert (out vector X, out vector Y, out vector Z); Inverts a coordinate system specified by three axis vectors.
- vector VRand ( ); Returns a uniformly distributed random vector.
- float Dist (vector A, vector B); Returns the euclidean distance between two points.
- vector MirrorVectorByNormal(vector Vect, vector Normal); Mirrors a vector about a specified normal vector.

Advanced Language Features

ForEach and iterator functions

UnrealScript's "foreach" command makes it easy to deal with large groups of actors, for example all of the actors in a level, or all of the actors within a certain distance of another actor. "foreach" works in conjunction with a special kind of function called an "iterator" function whose purpose is to iterate through a list of actors.
Here is a simple example of foreach:

```
// Display a list of all lights in the level.
function Something()
{
    local actor A;

    // Go through all actors in the level.
    log( "Lights:" );
    foreach AllActors( class 'Actor', A )
    {
            if( A.LightType != LT_None )
                    log( A );
    }
}
```

The first parameter in all "foreach" commands is a constant class, which specifies what kinds of actors to search. You can use this to limit the search to, for example, all Pawns only.
The second parameter in all "foreach" commands is a variable which is assigned an actor on each iteration through the "foreach" loop.
Here are all of the iterator functions which work with "foreach".

1. AllActors (class BaseClass, out actor Actor, optional name MatchTag); Iterates through all actors in the level. If you specify an optional MatchTag, only includes actors which have a "Tag" variable matching the tag you specified.

2. ChildActors(class BaseClass, out actor Actor); Iterates through all actors owned by this actor.
3. BasedActors(class BaseClass, out actor Actor); Iterates throgh all actors which are standing on this actor.
4. TouchingActors(class BaseClass, out actor Actor); Iterates through all actors which are touching (interpenetrating) this actor.
5. TraceActors(class BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent); Iterates through all actors which touch a line traced from the Start point to the End point, using a box of collision extent Extent. On each iteration, HitLoc is set to the hit location, and HitNorm is set to an outward-pointing hit normal.
6. RadiusActors(class BaseClass, out actor Actor, float Radius, optional vector Loc); Iterates through all actors within a specified radius of the specified location (or if none is specified, this actor's location).
7. VisibleActors(class BaseClass, out actor Actor, optional float Radius, optional vector Loc); Iterates through a list of all actors who are visible to the specified location (or if no location is specified, this actor's location).

Function Calling Specifiers

In complex programming situations, you will often need to call a specific version of a function, rather than the one that's in the current scope. To deal with these cases, UnrealScript provides the following keywords:

- Global: Calls the most-derived global (non-state) version of the function.
- Super: Calls the corresponding version of the function in the parent class. The function called may either be a state or non-state function depending on context.
- Super(classname): Calls the corresponding version of the function residing in (or above) the specified class. The function called may either be a state or non-state function depending on context.

It is not valid to combine multiple calling specifiers (i.e. Super(Actor).Global.Touch).
Here are some examples of calling specifiers:

```
class MyClass expands Pawn;

function MyExample( actor Other )
{
    Super(Pawn).Touch( Other );
    Global.Touch( Other );
    Super.Touch( Other );
}
```

As an additional example, the BeginPlay() function is called when an actor is about to enter into gameplay. The BeginPlay() function is implemented in the Actor class and it contains some important functionality that needs to be executed. Now, say you want to override BeginPlay() in your new class MyClass, to add some new functionality. To do that safely, you need to call the version of BeginPlay() in the parent class:

```
class MyClass expands Pawn;

function BeginPlay()
{
    // Call the version of BeginPlay in the parent class (important).
    Super.BeginPlay();

    // Now do custom BeginPlay stuff.
    //...
}
```

Default values of variables

Accessing default values of variables

UnrealEd enables level designers to edit the "default" variables of an object's class. When a new actor is spawned of the class, all of its variables are initialized to those defaults. Sometimes, it's useful to manually reset a variable to its default value. For example, when the player drops an inventory item, the inventory code needs to reset some of the actor's values to its defaults. In UnrealScript, you can access the default variables of a class with the "Default." keyword. For example:

```
var() float Health, Stamina;
//...

// Reset some variables to their defaults.
function ResetToDefaults()
{
    // Reset health, and stamina.
    Health = Default.Health;
    Stamina = Default.Stamina;
}
```

Accessing default values of variables in a variable class

If you have a class reference (a variable of "class" or "class<classlimitor>" type), you can access the default properties of the class it references, without having an object of that class. This syntax works with any expression that evaluates to class type.

```
var class C;
var class<Pawn> PC;
Health = class'Spotlight'.default.LightBrightness; // Access the default value of LightBrightness in the Spotlight class.
Health = PC.default.Health; // Access the default value of Health in a variable class identified by PC.
Health = class<Pawn>(C).default.Health; // Access the default value of Health in a casted class expression.
```

Accessing static functions in a variable class: Static functions in a variable class may be called using the following syntax.

```
var class C;
var class<Pawn> PC;
class'SkaarjTrooper'.static.SomeFunction(); // Call a static function in a specific class.
PC.static.SomeFunction(); // Call a static function in a variable class.
class<Pawn>(C).static.SomeFunction(); // Call a static function in a casted class expression.
```

Advanced Technical Issues

UnrealScript binary compatibility issues

UnrealScript is designed so that classes in package files may evolve over time without breaking binary compatibility. Here, binary compatibility means "dependent binary files may be loaded and linked without error"; whether your modified code functions as designed is a separate issue. on the Specifically, the kinds of modifications when may be made safely are as follows:

- The .uc script files in a package may be recompiled without breaking binary compatibility.
- Adding new classes to a package.
- Adding new functions to a class.
- Adding new states to a class.
- Adding new variables to a class.
- Removing private variables from a class.

Other transformations are generally unsafe, including (but not limited to):

- Adding new members to a struct.
- Removing a class from a package.
- Changing the type of any variable, function parameter, or return value.
- Changing the number of parameters in a function.

Technical notes

Garbage collection: All objects and actors in Unreal are garbage-collected using a tree-following garbage collector similar to that of the Java VM. The Unreal garbage collector uses the UObject class's serialization functionality to recursively figure out which other objects are referenced by each active object. As a result,

object need not be explicitly deleted, because the garbage collector will eventually hunt them down when they become unreferenced. This approach has the side-effect of latent deletion of unreferenced objects; however it is far more efficient than reference counting in the case of infrequent deletion.
Unreal COM integration: Unreal's base UObject class derives from IUnknown in anticipation of making Unreal interoperable with the Component Object Model without requiring binary changes to objects. However, Unreal is not COM-aware at the moment and the benefits of integrating Unreal with COM are not yet clear, so this project is on indefinite hold.
UnrealScript is bytecode based: UnrealScript code is compiled into a series of bytecodes similar to p-code or the Java bytecodes. This makes UnrealScript platform-neutral; this porting the client and server components of Unreal to other platforms, i.e. the Macintosh or Unix, is straightforward, and all versions can interoperate easily by executing the same scripts.
Unreal as a Virtual Machine: The Unreal engine can be regarded as a virtual machine for 3D gaming in the same way that the Java language and the built-in Java class hierarchy define a virtual machine for Web page scripting. The Unreal virtual machine is inherently portable (due to splitting out all platform-dependent code in separate modules) and expandable (due to the expandable class hierarchy). However, at this time, there are no plans to document the Unreal VM to the extent necessary for others to create independent but compatible implementations.
The UnrealScript compiler is two-pass: Unlike C++, UnrealScript is compiled in two distinct passes. In the first pass, variable, state and function definitions are parsed and remembered. In the second pass, the script code is compiled to byte codes. This enables complex script hierarchies with circular dependencies to be completely compiled and linked in two passes, without a separate link phase.
Persistent actor state: It is important to note that in Unreal, because the user can save the game at any time, the state of all actors, including their script execution state, can be saved at any time where all actors are at their lowest possible stack level. This persistence requirement is the reason behind the limitation that latent functions may only be called from state code: state code executes at the lowest possible stack level, and thus can be serialized easily. Function code may exist at any stack level, and could have (for example) C++ native functions below it on the stack, which is clearly not a situation which one could save on disk and later restore.
Unrealfiles: Unrealfiles are Unreal's native binary file format. Unrealfiles contain an index, serialized dump of the objects in a particular Unreal package. Unrealfiles are similar to DLL's, in that they can contain references to other objects stored in other Unrealfiles. This approach makes it possible to distribute Unreal content in predefined "packages" on the Internet, in order to reduce download time (by never downloading a particular package more than once).
Why UnrealScript does not support static variables: While C++ supports static (per class-process) variables for good reasons true to the language's low-level roots, and Java support static variables for reasons that appear to be not well thought out, such variables do not have a place in UnrealScript because of ambiguities over their scope with respect to serialization, derivation, and multiple levels: should static variables have "global" semantics, meaning that all static variables in all active Unreal levels have the same value? Should they be per package? Should they be per level? If so, how are they serialized -- with the class in its .u file, or with the level in its .unr file? Are they unique per base class, or do derived versions of classes have their own values of static variables? In UnrealScript, we sidestep the problem by not defining static variables as a language feature, and leaving it up to programmers to manage static-like and global-like variables by creating classes to contain them and exposing them in actual objects. If you want to have variables that are accessible per-level, you can create a new class to contain those variables and assure they are serialized with the level. This way, there is no ambiguity. For examples of classes that serve this kind of purpose, see LevelInfo and GameInfo.

UnrealScript programming strategy

Here I want to cover a few topics on how to write UnrealScript code effectively, and take advantage of UnrealScript's strengths while avoiding the pitfalls.

- UnrealScript is a slow language compared to C/C++. A typical C++ program runs at about 50 million base language instructions per second, while UnrealScript runs at about 2.5 million - a 20X performance hit. The programming philosophy behind all of our own script writing is this: Write scripts that are almost always idle. In other words, use UnrealScript only to handle the "interesting" events that you want to customize, not the rote tasks, like basic movement, which Unreal's physics code can handle for you. For example, when writing a projectile script, you typically write a HitWall(), Bounce(), and Touch() function describing what to do when key events happen. Thus

95% of the time, your projectile script isn't executing any code, and is just waiting for the physics code to notify it of an event. This is inherently very efficient. In our typical level, even though UnrealScript is comparably much slower than C++, UnrealScript execution time averages 5-10% of CPU time.

- Exploit latent functions (like FinishAnim and Sleep) as much as possible. By basing the flow of your script execution on them, you are creating animation-driven or time-driven code, which is fairy efficient in UnrealScript.
- Keep an eye on the Unreal log while you're testing your scripts. The UnrealScript runtime often generates useful warnings in the log that notify you of nonfatal problems that are occuring.
- Be wary of code that can cause infinite recursion. For example, the "Move" command moves the actor and calls your Bump() function if you hit something. Therefore, if you use a Move command within a Bump function, you run the risk of recursing forever. Be careful. Infinite recursion and infinite looping are the two error conditions which UnrealScript doesn't handle gracefully.
- Spawning and destroying actors are fairly expensive operations on the server side, and are even more expensive in network games, because spawns and destroys take up network bandwidth. Use them reasonably, and regard actors as "heavy weight" objects. For example, do not try to create a particle system by spawning 100 unique actors and sending them off on different trajectories using the physics code. That will be sloooow.
- Exploit UnrealScript's object-oriented capabilities as much as possible. Creating new functionality by overriding existing functions and states leads to clean code that is easy to modify and easy to integrate with other peoples' work. Avoid using traditional C techniques, like doing a switch() statement based on the class of an actor or the state, because code like this tends to break as you add new classes and modify things.

V e c t o r s

I n t r o d u c t i o n

Before you get too far into this, I should let you know that this is not a tutorial for someone just starting out in UnrealScript. If you're new to coding, you should check out the Beginner's Guide to UnrealScript. That ought to get you a good start. Alright, with that out of the way, let's get down to the good stuff. Vectors are a very important part of UnrealScript, since they are responsible for controlling how objects move in the 3D world. If you fire a grenade from your Eightball gun, its path is determined by a vector. Anything you see moving in some way is most likely controlled by a vector. So what is a vector? Keep reading.

T h e   M y s t i c a l   V e c t o r

Well, not exactly mystical, but pretty damned cool, anyway. If you get right down to it, a vector in UnrealScript is simply a struct with three float components: X, Y, and Z. Sounds simple, eh? Well, those three little numbers can do quite a bit. There are two different ways in which a vector can be used in UnrealScript. First, they can represent a position in the 3D world. X, Y, and Z: Just a point in 3D space. Second, a vector can be used to represent a direction and speed. A direction and speed, you ask? How in hell could 3 simple numbers tell all that? Well, understanding how they do tell all that is key to being successful when you work with vectors. Think of it like this: Say you have an object at position (1,1,1). You set that object's velocity to a vector of (0,0,1). After one unit of time the object's new position will be (1,1,2). How'd it get to (1,1,2)? Well, in that unit of time, the value of the movement vector was added on to the object's original position, giving a new position of (1,1,2). After another unit of time, the object's position will be (1,1,3), then (1,1,4), and so on. So, which direction is the object moving? Why, up, of course. It's Z value is continually getting bigger, while it's X and Y values stay the same. So, there's the direction part. But what about speed? Well, what if the object's velocity was set to (0,0,5) instead of (0,0,1). Then, after one unit of time, the object's position would be (1,1,6). After two units of time, it would be (1,1,11). So, it's still moving in the same direction (up), it's just moving much more quickly. There's the speed part. By supplying different values for these three numbers, you can make an object move in any direction at any speed. Conceptually, it's not too hard to see this. But how in the hell would you do the math to figure out what three numbers to use if you wanted an object to move at some obscure angle at some obscure speed? It's possible, obviously, but there must be an easier way. Fortunately, there is.

T h e   R o l e   o f   R o t a t o r s

Let's move away from the X, Y, Z view of vectors for a while, and take another look at them. When I picture a vector in my mind, I always think of an arrow. Where this arrow points indicates a direction, and the length of the arrow indicates a speed. The longer the arrow, the faster the object will move. I don't worry too much about the actual numbers of the vector, because I don't need to. There's an easier way to get to them. You may have encountered rotators before. Like vectors, they're also structs with three float components: pitch, yaw, and roll. These three values are angles that can describe any rotational direction in 3D space. Pitch is up and down, yaw is side to side, and roll is... well, roll. Roll is unimportant when you're working with vectors, though, so don't worry about it too much. There's one important thing to remember about UnrealScript rotators, though. They're not measured in degrees, or even radians. For some reason, Epic chose to make up their own little system of angle measure. In this system, 360 degrees is about equal to 65535 Unreal angle units. Quite a big difference there. So don't get the two mixed up, or you'll spend hours trying to figure out why in the hell your setting of 60 degrees still looks like 0 in the game. Now, say you're in a game, and the player is facing a certain direction. How would you find a vector that points 45 degrees to the player's right? Well, it's simple really.

Vectors and Rotators

```
function VectorRotator()
{
    local rotator VectRot;
    local vector SomeVector;

    VectRot = Player.ViewRotation;
    VectRot.Yaw = VectRot.Yaw + 8192;
    SomeVector = Vector(VectRot);
}
```

Note that this is all assuming "Player" is an actor reference variable set equal to some PlayerPawn. Anyway, what's going on here? Well, first, VectRot is set equal to the player's view rotation. Then, 8192 (45 degrees) is added to its yaw, making it point 45 degrees to the player's right. Finally, SomeVector is set equal to "Vector(VectRot)". The Vector() function simply takes a rotator, and returns a vector pointing in the same direction as the rotator, with a length (speed) of one. So, now you have an easy way of getting a vector that points in any direction. But what about its speed? What if you want a vector with a speed of five instead of one? This is easy, too. Simply multiply the one-length vector by five. You'll get a vector pointing in the same direction, but with a length of five instead of one.

Other Vector Operations

There are quite a few little tricks you can use to get the vector you're looking for. For instance, how the hell would you come up with a vector that points from one object to another? It's simple, really. You just have to know the trick. So, for your reading pleasure, here is the amazing and wonderful chart of useful UnrealScript vector operations.

Normal(Direction):Returns a vector pointing in the same direction as the vector you supply, but with a length of one.

VSize(Direction):Returns a floating point value equal to the length of the vector you supply.

Direction + Direction:Returns a direction vector which is equal to the bisector of the parallelogram formed by the original two direction vectors. What the bloody hell does this mean? Just take a look at the diagram.

As you can see, this is on a 2D plane, and we're talking about 3D space. If you think about it, though, two vectors in 3D space, no matter which direction they're pointing, can always be thought of as being on some 2D plane or another.

Location - Location:Gives you a direction vector which points from the second location to the first, and has a length equal to the distance between the two locations. This is not only useful for making one object move towards another, but also for determining the distance between things.

Location + Direction:Gives you a location equal to the new location an object would have after one unit of time with a velocity of the direction vector. If that made no sense whatsoever to you, think of it this way. Take the X component of the location vector, and add to it the X component of the direction vector. Do the same for the Y and Z components, and you'll end up with the location that's returned by this operation.

Mod Authoring for Unreal Tournament

Introduction

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 damage you do to an enemy.")
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.

Uwindows

Part 1

What is UWindows?

Probably the first thing about Unreal Tournament you'll notice is how much spiffier the interface is. Rather than having the Wolfenstein 3D throwback menu system, UT has implemented its own small Graphical User Interface (GUI). It has windows, menus, a mouse pointer, scrollbars, and a lot of other things which other GUIs have (such as Windows). GUI interfaces are famous for making information easier to convey, and harder to program.

Luckily, UT's interface isn't really all that hard to program. If you understand some basic object orientated principles, you shouldn't have much trouble with it. If you've programmed for Windows, Java, or the Macintosh, you'll find that UT's interface is much more simple and easier to use.

UWindows may or may not be the official title of the GUI interface. The core classes all have names with the prefix UWindow, like UWindowButton or UWindowTabControlLeft. For obvious reasons, I call the interface UWindows.

How does UWindows work?

UWindows makes up the core engine of the graphics interface. It does not include any game specific classes. This allows mod makers and liscencees to make their own games with the Unreal engine without having all that extra game specific code. UT's interface exists as a layer or two above the UWindows core. Right above UWindows is the UMenu interface. It provides the basic interface with the help system, menu bar, and configuration dialogs. Built on top of UMenu is UTMenu, which is the specific classes used in Unreal Tournament.

When working with UWindows, you really shouldn't find much need to mess around with the UMenu and UTMenu layers unless you are trying to specifically change a feature of Unreal Tournament. Working with classes outside of the standard UWindows core group could mean compatibility problems with other game engines. Creating a mutator configuration window could actually work between Unreal Tournament, Wheel of Time, and even Nerf Blast Arena if you never stray past the standard classes.

It all begins with UWindowBase?

The entire class tree for UWindows, UMenu, and UTMenu all start with a single node: UWindowBase. All classes in the UWindows system stem from this single point, and they all inherit what is in it. As such, UWindowBase doesn't really have a whole bunch of code. It provides several constants and enumerations that are used amonst the bunch.

After UWindowBase, the class tree splits three ways. Probably the least important path of the three is UWindowLookAndFeel. It provided the hooks for building in the look and feel of the application. Maybe you've noticed in UT, you can change your Look and Feel between Gold, Ice, and Metal? Well, each of those Look and Feels are objects under this class. They define how buttons are drawn, and what a menu looks like, and so on. Nerf Blast Arena even has its own Look and Feel. However, the provided L&Fs should be more than enough to satisfy most ideas.

The second branch past UWindowBase is the UWindowList tree. None of the classes under UWindowList are graphical objects. UWindowList provides a basic linked list functionality for its children classes. Most of the child classes are specific lists used in the GUI, such as a list of menu items, or the list of strings in a combo box. There really isn't a whole lot to monkey around with in this section, although you do need to subclass at least one thing here.

The third and final branch of the tree is UWindowWindow. While it says window, it really isn't. In this case, the window it is refering to is a rectangular graphic. This could be a window window, but this could also be a menu, or a button. All objects under UWindowWindow are visible objects that can be messed with by either the mouse or the keyboard. To distinguish between a UWindowWindow and a window with a title bar that can be moved around, I shall refer to the latter as a Framed Window.

The really cool thing about everything descending from UWindowWindow is that anything under this class can have anything else in this class inside it. You can have Framed Windows with menu bars, and menu bars with buttons, and even buttons that pop up menus. By careful construction of the class tree, Epic hsa made it very functional and very easy to use.

So everything is a window?

All visible graphical objects are descended from UWindowWindow. They all share the same properties and values defined in this class. Easily the most important of these are four variables: WinLeft, WinTop, WinWidth, and WinHeight. These variables form the rectangle that the window occupies. There are a few boolean variables of note, such as bAlwaysOnTop, bAlwaysBehind, and bWindowVisible.
There are several functions in UWindowWindow which don't do anything, but are worth pointing out. Functions like BeginPlay(), BeforeCreate(), Created(), AfterCreate(), Activated(), and Resized() allow their functionallity to be filled out by the children classes. You might override the Created() function of a window to add controls to it. Just be sure that when you override a function to call the parent function too (ie Super.Created()).

How do I start?

This tutorial assumes you are already a little familiar with UScript, and have a mutator or two done. The provided sample code assumes a mutator called sqFatMutator with two variables: fatness and bUseTwoGuns.
The first thing you need is a starting place. UT needs to know when and how to create a window for you. Luckily, UT makes this easy by providing a built in starting place: the MOD menu. Normally, this menu is invisible, but shows up when you add your first item to it. You create your own spot on the menu by subclassing UMenuModMenuItem (from the UWindowList tree). Notice that it is a UMenu class and not a UWindow one. The UMenuModMenuItem has only two functions: Setup() and Execute(). Here is the code for a sample mod menu item (remember to change the class names and what-not to your classnames first).
One the class is completed and compiled, add a line similar to the following to your .int file:

```
[Public]
Object=(Name=SquidiMods.sqFatModMenuItem,Class=Class, MetaClass=UMenu.UMenuModMenuItem)
```

```
-----------------------------------------------------------------
class sqFatModMenuItem expands UMenuModMenuItem;
function Setup()
{
	//The MenuCaption variable is used to store the menu item's name on
	// the menu. Unfortunately, the menu item is created before Setup()
	// is called, and thus uses the default property. Still, this is
	// provided as a reference.
	MenuCaption = "FatMutator Config";
	//MenuHelp is simply the string written in the status bar at the bottom
	// of the screen when the mouse hovers over this menu item. Luckily
	// for us, setting the help caption here DOES work.
	MenuHelp = "Sets up the FatMutator";
}
function Execute()
{
	//Create a window using the following function:
	// CreateWindow( class, top, left, width, height )
	//The window I will create will automatically resize itself to
	// the right proportions (I'll show you how when we get to the
	// Framed Window part), so it doesn't matter what the top, left,
	// width or height variables are. Just the class portion matters.
	MenuItem.Owner.Root.CreateWindow(class'sqFatConfigWindow',
	10,10,10,10);
}
-----------------------------------------------------------------
```

It didn't compile!

When UnrealEd tries to compile this code, it'll point out that you haven't written the sqFatConfigWindow class yet. So lets do that. What we are going to do is subclass UWindowFramedWindow to create a nifty little frame which we drag around the screen to our heart's content.

```
-----------------------------------------------------------------
class sqFatConfigWindow expands UWindowFramedWindow;
function BeginPlay()
{
	Super.BeginPlay();
	//Set the title of the Framed Window
	WindowTitle = "Configure Fat Mutator";

	//The class of the content
	ClientClass = class'sqFatClientWindow';
	//Make the Framed Window resizable
	bSizable = true;
}
-----------------------------------------------------------------
```
Most of this code is pretty self explanatory, with the exception of the ClientClass variable. If this variable were set to None, you'd see that when the window was created you would just get the framed part. The insides would show through to the background! A FramedWindow is just that, a frame. When ClientClass is set to a valid client window class, when the window is created, that class will be created along with it. You don't need to worry about making both of them. You'll find the class UWindowClientWindow just under WindowWindow, but that isn't the client window type we are going to use. A child class of ClientWindow, UWindowDialogClientWindow has a few extra important functions required to use controls. So we are going to subclass that one instead. It also has an extra ability to suggest a desired width and height for the content. Just subclassing UWindowDialogClientWindow is enough to fill in the empty space in the Framed Window.

Ooh. How do I add buttons and stuff?

Content is added in the form of controls. All controls are children of UWindowDialogControl (which in turn is a child of UWindowWindow). So all controls are also windows. There are dozens of controls to choose from. Some of the controls share functionality (for instance UWindowCheckbox and UWindowSmallButton are both children of UWindowButton), but most have individual options to tweak. Controls are created, and added to your client window via the UWindowDialogClientWindow.CreateControl() function that you'll have inherited. It has the same parameters as the CreateWindow() function. Here is a somewhat modified version of my sample client window class:

```
-----------------------------------------------------------------
class sqFatClientWindow expands UWindowDialogClientWindow;
var UWindowCheckBox checkbox;
var UWindowHSliderControl slider;
// called after object has been created...add content
function Created()
{
	Super.Created();
	//Add checkbox via the CreateControl() function.
	checkbox = UWindowCheckBox(
	CreateControl(class'UWindowCheckBox', 10, 10, 150, 1));
	checkbox.SetText("Two Guns");
	checkbox.bChecked = true;
	//create slider in same fashion
	slider = UWindowHSliderControl(
	CreateControl(class'UWindowHSliderControl', 10, 30, 150, 1));
	slider.SetText("Fatness");
	slider.SetRange(1,255,5);
	slider.SetValue(128,true);
}
//when a control changes, Notify is called with the changed control
// and the message sent.
function Notify(UWindowDialogControl C, byte E)
{
```

```
            switch(E)
            {
            case DE_Change: // the message sent by sliders and checkboxes
                        switch(C)
                        {
                        case slider:
                        //get slider value
                        break;
                        case checkbox:
                        //get checkbox info
                        break;
                        }
            break;
            }
}
```

The Notify() is how you know when one of your controls has changed in value. This allows you to update the variables the controls were supposed to reflect. The message sent in this case is the DE_Change one, which is the message most often used. You can also use Notify() to update the text on the controls. For instance, the fatness control can change from skinniness to fatness depending where the slider is.

Great! That's just about it. How do I interact with my mutator?

To make the config dialog more useful, those levers and buttons actually need to affect something in your mutator. This is accomplished by setting the default values. This can easily be accessed via a line similar to this:

```
class'SquidiMods.sqFatMutator'.default.fatness = slider.GetValue();
class'SquidiMods.sqFatMutator'.static.StaticSaveConfig();
```

The final version of my client window sample follows:

```
class sqFatClientWindow expands UWindowDialogClientWindow;
var UWindowCheckBox checkbox;
var UWindowHSliderControl slider;
// called after object has been created...add content
function Created()
{
            Super.Created();
            //Add checkbox via the CreateControl() function.
            checkbox = UWindowCheckBox(
            CreateControl(class'UWindowCheckBox', 10, 10, 150, 1));
            checkbox.SetText("Two Guns");
            checkbox.bChecked =
            class'SquidiMods.sqFatMutator'.default.bUseTwoGuns;
            //create slider in same fashion
            slider = UWindowHSliderControl(
            CreateControl(class'UWindowHSliderControl', 10, 30, 150, 1));
            slider.SetText("Fatness");
            slider.SetRange(1,255,5);
            slider.SetValue(
            class'SquidiMods.sqFatMutator'.default.fatness, true);
}
//when a control changes, Notify is called with the changed control
function Notify(UWindowDialogControl C, byte E)
{
            local int sval;

            switch(E)
            {
```

```
			case DE_Change: // the message sent by sliders and checkboxes
			switch(C)
			{
			case slider:
			sval = slider.GetValue();
			class'SquidiMods.sqFatMutator'.default.fatness = sval;
			class'SquidiMods.sqFatMutator'.static.StaticSaveConfig();
			if( sval > 0 ) slider.SetText("Fatness [Skinny]");
			if( sval > 50 ) slider.SetText("Fatness [Slim]");
			if( sval == 128) slider.SetText("Fatness [Normal]");
			if( sval > 175 ) slider.SetText("Fatness [Fat]");
			if( sval > 200 ) slider.SetText("Fatness [Fatboy]");
			break;
			case checkbox:
			class'SquidiMods.sqFatMutator'.default.bUseTwoGuns
			= SomeCheck.bChecked;
			class'SquidiMods.sqFatMutator'.static.StaticSaveConfig();
			break;
			}
			break;
	}
}
```

Part2

Before We Begin

This tutorial assumes you are already familiar with the basics of programming the UWindows system. This includes making windows and client windows, adding controls to windows, adding a spot on the modmenu, and generally understanding the terminology. When I talk about ClientWindows and FramedWindows, I am going to assume that you know the difference between them. As such, I will not walk you through the process of doing these things. Please see my introductory tutorial for that type of help.
All of my class names begin with 'sq' (for Squidi), and then an uppercase abbreviation of what that particular item is. ClientWindows are CW, Framed Windows are FW, Menu Bars are MB, Menus are M, and so on. Then is the mod type (in this case "Tutorial"). After that, if it is necessary, is a description of the object. sqFWTutorial and sqCWTutorial are sufficient enough to distinguish a difference. However, individual menus might be called sqMTutorialGame and sqMTutorialHelp.

Creating and Using Menus

To use menus in your configuartion windows, you need a way to get to the menu. This can be accomplished through a MenuBar, or through using Right Clicks to bring up a pop up menu. MenuBars are a little easier follow, so we'll start there. To create a Menu (and MenuBar), you must subclass the appropriate object (UWindowMenuBar and UWindowPulldownMenu). Since both MenuBars and Menus are descendents from UWindowDialogControl, they can both exist inside framed windows, and are created using the CreateControl() function.
In the following examples, I haven't implemented the tooltips helpbar that the main menu bar makes. It makes things a little more complicated. If you want to add in this help functionality, checkout UMenuGameMenu and the likes. Also, I won't go into Right Click menus at this time, as I'm unsure how Mac users do the right click. I don't want to promote an activity that I'm not sure everyone can participate in.

```
//=============================================================================
// sqMBTutorial.
//=============================================================================
class sqMBTutorial expands UWindowMenuBar;
var UWindowPulldownMenu Game, Help;
var UWindowMenuBarItem GameItem, HelpItem;
```

```
function Created()
{
            Super.Created();
            bAlwaysOnTop = False;
            GameItem = AddItem("Game");
            Game = GameItem.CreateMenu(class'sqPMTutorialGame');
            // For this menu, we are using the built in HelpMenu
            HelpItem = AddItem("Help");
            Help = HelpItem.CreateMenu(class'UMenuHelpMenu');
            Spacing = 12;
}

/**
** Adding in the following code will make the menuBar drawn in the
** current look and feel, as opposed to a generic gray one
**
** function DrawMenuBar(Canvas C)
** {
** LookAndFeel.Menu_DrawMenuBar(Self, C);
** }
**
**
**/

//=============================================================================
// sqPMTutorialGame.
//=============================================================================
class sqPMTutorialGame expands UWindowPulldownMenu;

var UWindowPulldownMenuItem NewGame, Load, Save, GameOptions, Quit;

function Created()
{
            Super.Created();
            // Add menu items.
            NewGame = AddMenuItem("New Game", None);
            Load = AddMenuItem("Load", None);
            Save = AddMenuItem("Save", None);
            AddMenuItem("-", None);
            Quit = AddMenuItem("Quit", None);
}

function ExecuteItem(UWindowPulldownMenuItem I)
{
            switch(I)
            {
            case NewGame: // new game menu item was selected
                        break;
            case Load: // Load menu item was selected
                        break;
            case Save: // Save Menu Item was selected
                        break;
            case Quit: // Quit menu item was selected
                        break;
            }
            Super.ExecuteItem(I);
}
```

Add the MenuBar to your window with the following code (in the clientwindow's Created() function):
MenuBar = sqMBTutorial(CreateWindow(class'sqMBTutorial', 0, 0, WinWidth, 16));
And that's about it. You may want to put some code in the client window's Resized() function to change the width of the menuBar when the window changes sizes. Otherwise, the user may make the window larger than the menuBar, and you'll see empty space. Another thing of note is that you don't need to add a menuBar to the top of a window. You can put it anywhere. Just remember that the pulldown menus will drop below the menuBar no matter where the location is.

Tabbed Pages

An example of Tabbed Pages can be seen in the preferences dialog. The window has several tabs at the top of the window which will select between the different pages of configurable options. Making a TabbedPage is incredibly easy. You just need to create a control of type UWindowPageControl, and then add the different pages to it. Each page is just a client window of type UWindowPageWindow. Since UWindowPageWindow is descended from UWindowDialogClientWindow, you create them in the same fashion as you would a normal client window. In fact, if you aren't really sure whether or not your client window will ever be used in a tabbed page, you can go ahead and make it a subclass of UWindowPageWindow anyway. The change is transparent to UWindows, and saves you from having to change parent classes if you decide later on to put it under a tab.
To create a tab control, just do the following in your client window's Created() function:
Pages = UWindowPageControl(CreateWindow(class'UWindowPageControl', 0, 0, WinWidth, WinHeight));
To add a page to the tab bar, just use the following code, where XXX is the class name of the Page Window you want to add. And that's it. The most important thing to remember is to subclass UWindowPageWindow for your client window needs.
Pages.AddPage("Test", class'XXX');

Scrolling Client Areas

Maybe you've noticed that when you resize windows, the controls and content don't change. They are obscured by the edges of the window when it gets too small. One way to change this is to resize each of the controls when the window changes in size. My client window has the following code to make sure the menu bar and tabbed controls stay with proportion to the window's size. The function Resized() is called whenever the client area changes size.

```
function Resized()
{
        Super.Resized();
        MenuBar.WinLeft = 0;
        MenuBar.WinTop = 0;
        MenuBar.WinWidth = WinWidth;
        MenuBar.WinHeight = 16;
        Pages.WinTop = 20;
        Pages.WinLeft = 10;
        Pages.WinWidth = WinWidth - 20;
        Pages.WinHeight = WinHeight - 30;
}
```

This works great because the menu bar and tab area always have a size relative to the window. But in cases where you have controls and other client windows, you can't always change their size small enough. The solution to this is to provide a scrollable area. The class UWindowScrollingDialogClient (which is a Page Window) provides scrollbars which allows its own client area to remain the same size and be scrolled around in.
They are really easy to use. Just like a framed window, they have a client class variable which will create the specified client area. This time the client window doesn't need to be a PageWindow. Here is some sample code (lifted straight out of UMenuHUDConfigScrollingClient)

```
class UMenuHUDConfigScrollClient extends UWindowScrollingDialogClient;

function Created()
{
	ClientClass = Class<UMenuPageWindow>(DynamicLoadObject(
	GetPlayerOwner().MyHUD.HUDConfigWindowType, class'Class'));
	Super.Created();
}
```

The scrolling client exists as an invisible layer between the client area, and the container in which the client area exists. Most of UT's dialogs use scrolling clients for the internal client areas. Then if the area is too large, or the user sets big fonts, the user can still get to every control. You should really use scrolling clients whenever you get the chance, as they add functionality which either is invisible to the user, or makes his life a lot easier. It does, however, add one more class to your user interface for each client area which uses it.

Object Oriented Logic Tutorial

Introduction

Talking on #UnrealED and #UnrealScript (both active EF-Net channels on IRC) it has come to my attention that a lot of interested UnrealScript hackers aren't very familiar with Object Oriented (OO) logic. In an attempt to do my bit o' public good, I'm writing this tutorial as a short guide to thinking in OO. Hopefully, by the time you are done reading this, you'll have a strong enough grasp of Object Oriented Programming (OOP) to work uninhibited with UnrealScript. From my experience, I have learned that just by approaching a mod idea or problem with OO design in mind, an answer is more easily found. I hope this tutorial is useful to you. I will continue to update it for as long as I see necessary. If you have any information or corrections than I urge you to email the address above. I will be more than happy to include such items with credit to the author. This is just the first in a series of useful tutorials (called tuts by the in-crowd hehe) that I plan on authoring. Keep your eyes peeled.

Legal

The Way Programmers Think

Programmers are a strange breed. They often forget to eat. They often forget to sleep. They even tend to neglect their girlfriend/boyfriend (if they are lucky enough to possess one *hint hint*) all the while attempting to make rather unintelligent machines more intelligent than any reasonable person would deem worthwhile. Programmers, you see, are not reasonable people. This lack of reason is in part due to the aforementioned lack of food, sleep, and sex, and also partly due to the way programmers think. Programmers like to solve problems. Specifically, programmers like to solve problems by changing the way they think about the problem. Object Oriented Programming, which is the focus of this tutorial, is one such way of solving problems. To understand the OO method of solving a problem, we first have to change the way we think about the problem.

Breaking It Up

Let's say you, as a programmer, have a problem. You want to write a program that will build a car. Sounds tough doesn't it? Well it is, building a car is certainly no trivial task, but half of the difficulty in thinking of a solution is in the way we think of the problem. If you say "I want to write a program that will build a car," you are probably mentally overwhelmed by the immensity of the task! A car is made up of thousands of parts! How can you possibly write a program that will make all those parts? Thousands of parts? Ah ha! We've already started to break it up. What if, instead of saying "I want to write a program that will build a car," you said "I want to write a program that will build a series of car parts and then assemble those parts." Still a daunting task, but certainly more organized. This is what we call "breaking it up" or "top-down programming." By breaking the core problem up into successively smaller pieces, you are faced with many small, easy tasks instead of one large, difficult task. If we were to leave the car analogy and move to an Unreal analogy, one might say "I want to write a bot that plays Unreal." A tough problem. However, if you break a bot down into successively smaller pieces, you end up with a path-finding project, a weapon-using project, and so on. This is the first step in taking an OO approach to programming.

Getting Organized

Now that we've got a bunch of little tasks that make up our big task, we've got to get organized. If you were to write your car program in the classical C "functional" style, you'd have a lot of functions and a big mess. You could clean it up by assigning certain parts of the car to their own files, but that would still be pretty wild. How to we organize the small parts of the big task? The solution is to change the way we think of the parts of the problem. Let's look at the car analogy. We want to build a car and we are going to do it by having one part of our program build car parts and another part of our program assemble those parts into a working car, right? Well what, exactly, is a car part? Its maybe a mix of metal, plastic, smaller parts, and it has a specific function...different car parts of the same type might be slightly different. Basically, we could say that a car part is a generic type of object. It isn't any specific object in particular, just a blueprint by which we could manufacture the actual thing. Let's call this a "class."

The Class

In OO logic, a "class" is a description of what a thing might be if it were created. A class is a generic template. It defines abstract properties (the car part has color) but usually does not define the specific nature of those properties (the car part is red). A class also defines the behavior of an object. A series of special functions that the class owns define the exact way in which an object you create works in your program. In programmer speak, the abstract properties are called "instance variables" and the behavior functions are called "methods." The process of taking a class and making an object from it is call "instantiation." It is _very_ important to realize that a class is not an object, merely a blueprint by which an object may be made. When you instantiate an object from a class, you are creating a model from that blueprint.

Class Mechanics

In UnrealScript, we define a class through the "class declaration:"

```
class MyClass expands MyParentClass;
```

The class specifier tells UnrealScript that you are starting to define a new class. MyClass is the name of this class (it can be whatever you want, preferably something meaningful, like CarPart). We'll get to the rest in a bit.
After you've made your class declaration in UnrealScript you are ready to define the class's instance variables. This is done by listing a series of property variables:

```
var int color;          // Part color index number
var byte manufacturer;  // Manufacturer reference value
```

You can find out the specific data types that UnrealScript supports after you've read this tutorial by reading Tim Sweeney's UnrealScript Language Reference at http://unreal.epicgames.com/UnrealScript.htm. It might be smart to separate your instance variables from the rest of the code with a comment line like this:

```
////////////////////////////////////////////////////
// Instance Variables for MyClass
```

This is just a matter of taste, but it won't hurt. Its usually a _good thing_ to make your code more readable, especially if you want to share it with others or come back to it later.
Defining methods for your new class are simular to defining instance variables, just make a list of functions:

```
function doAThing()
{
  // Do some stuff in UnrealScript
}

function doAnotherThing()
{
  // Another function that does something
}
```

Once again, you might separate the method area of your class from the rest of the code by using a comment line. It is also highly suggested that you name your methods after their functions. For example, a function that cleans your socks might be called cleanSocks().
In UnrealScript, we call object instantiation "spawning." As such, you use the Spawn() function to create a new object from a class template:

```
var actor MyObject;          // variable to hold the object
MyObject = Spawn(MyClass);   // spawning the object
```

A Brief Discussion of Methods

Objects are independent collections of "interactive" data that sit in memory. The keyword is _independent_. Each object does its own thing. If you have a class called MyBot and you spawn two objects from that class called BotA and BotB those two instantiations don't know about each other. This leads us to the next concept of Object Oriented Programming: objects edit themselves.

When an object is spawned it is usually stuffed into a hash table in memory, given a lookup key, and forgotten about. The system uses the lookup key to find the specific object in the hash table if you want to do something to it, but for all intents and purposes the object is just sort of unavailable. Unavailable in the since that, unlike a normal variable, you can't just change it. You cannot, for example, rip open an object and change the contents of the instance variables. (This is one of the ways in which objects are different from structs in C++.) Instead, you have to tell the object to change itself. This is done through the methods.

The methods of a class define the way an object will act when it is spawned. If you want to change an instance variable of an object, you have to have written a method in the class that allows this to behavior to take place. Our CarPart class might have a method that looks like this:

```
 function setColor(int newColor)
 {
   color = newColor;
 }
```

In this case, when the above method is called, the object takes the int supplied as an argument and sets the color instance variable to that value. The object alters itself. The syntax for calling an object's method in UnrealScript looks like this:

```
MyObject.setColor(15);  // tell the object to call it's setColor() method
```

Methods, Variables, and Object Security

Remember when I told you that an object alters itself? Well that isn't entirely true. I wanted you to believe that so you would start thinking about objects as independent entities in your programming environment. It is possible, in fact, to change the instance variables of an object directly:

```
MyObject.color = 15;
```

Notice the difference. In the method case, we tell the object to change itself and in the assignment case we change the nature of the object directly. This presents another one of those "How Programmers Think" issues. You're probably asking yourself if I can just alter an object directly, why would I ever use a method to do it? Well, think about it.

What if there were very special restrictions on the instance variable "color"? For example, you might want the color variable to only contain values from 1 to 10. Anything outside of that range would maybe cause your program to act unpredictably. In that case, it just wouldn't be a good idea to allow the object's user to edit the color variable directly. Right? Even though _you_ might know that 1 to 10 are the only correct values for color, someone else who uses your code might not. The solution is to make the instance variable private, so that only the class itself can change it:

```
var private int color;  // declare a private variable
```

The private specifier indicates that this instance variable is _only_ accessable by an object of this class. The object can change the color variable from inside its own methods, but an external assignment, like:

```
MyObject.color = 15;
```

Would become an error no matter what the right hand value. This allows you to control the input to your objects and more clearly define their behavior. Now you could have your object return an error code or take appropriate action if an invalid value was passed to it through a method.

Class Families

Whew. Getting tired yet? This might be a good time to grab a Dr. Pepper. We are just now getting the fundamental elements of objects!

As you can see from the above (if you are a creative individual and you must be if you are reading this), objects alone have a lot of potential. Objects make it easy to break down a problem into usuable parts. Nonetheless, it can still be difficult if you have to manage lots of objects. This brings us to the fundamentals of object oriented programming: The relationships between objects.
A good way to picture object relationships is through our car analogy. The CarPart class certainly doesn't go very far in describing what a CarPart is. Given what we know about objects so far, we'd probably not even use CarPart...we'd have to write classes like SteeringWheel that are more specific and useful.
Actually, this isn't quite the case. In our minds, CarPart has already created a relationship to SteeringWheel. A steering wheel is a kind of car part. Right? So what if the CarPart class defined very generic methods and instance variables that all car parts used and another class called SteeringWheel _expanded_ that functionality?
In programmer speak we call this "the parent-child relationship." CarPart is the "parent class" (or super class) of SteeringWheel. In UnrealScript, we define a child class like this:

```
  class SteeringWheel expands CarPart package(MyPackage);
```

See the expands specifier? It indicates that the class we are now declaring (SteeringWheel) is a child class of CarPart. As soon as Unreal sees this it forms a special relationship between the two classes.

Fundamental I: Inheritance

What, exactly, does this parent-child relationship do for us, as problem solvers? It simplifies the solution, that's what! By creating a parent-child relationship between two classes, the child class immediately "inherits" the properties and methods of the parent class. Without you even having to type a line of code, SteeringWheel contains all of the functionality of CarPart. If CarPart has a setColor() method defined (as discussed above) SteeringWheel has the same method. Inheritance applies to instance variables, methods, and states.
This allows us to create what programmers call an "Object Hierarchy" (or class family). Its a lot like a family tree:

```
     Object
       | expanded by
      Actor
       | expanded by
     CarPart
       | expanded by
   SteeringWheel
```

Object and Actor are special classes in Unreal described in Tim Sweeney's guide. The full class family tree for Unreal is a sprawling web of relationships as you can no doubt imagine. In our example, we have a basic "is-a" relationship:

A SteeringWheel is a CarPart.
A CarPart is an Actor.
An Actor is an Object.

Each successive layer of the family tree inherits and expands upon the functionality and detail of the previous layer. This allows us to easily describe a complex object in terms of its component objects. It is important to realize that the relationship is not commutative. A SteeringWheel is always a CarPart, but a CarPart isn't always a SteeringWheel. Moving up the tree you get more general and moving down the tree you get more specialized. Get it? Good!
But wait a second...if we are building the car and a car is made up of parts, where is the Car class? This brings us to an important distinction in relationships: is-a vs. has-a. Clearly, a CarPart is not a kind of Car. Therefore, the relationship "CarPart expands Car" would be invalid. Rather, you would have a tree structure that might look like this:

```
        Object
          |
        Actor
       /    \
     Car   CarPart
              |
    SteeringWheel
```

Car is a class derived from Actor, but it doesn't have a direct relationship to CarPart (you might say they are siblings). Instead, the internal definition of the Car class might include instance variables that are CarParts. In this case, we have a has-a relationship. A Car has a SteeringWheel, but a SteeringWheel is not a Car. If you are ever designing a class hierarchy like this and you get confused about object relationship, it is sometimes very useful to phrase in the relationship in the "is-a" or "has-a" style.
As you can see, the relationship hierarchy allows us to do some very interesting things. If we wanted to, for example, make a more liberal definition of Car, we could add Vehicle:

```
            Object
              |
             Actor
            /    \
         Part    Vehicle
        /   |    |    \
   CarPart AirPart  Car  Airplane
```

Pretty cool huh? Not only is it a great way to organize and visualize data, but the benefits of inheritance mean we save time that would normally be spent copying and rewriting code!

<u>Fundamental II: Polymorphism</u>

Poly what? Its more of that crazy programmer speak. (If you've understood everything up until now, you are more a programmer than you think.) Polymorphism is another one of the fundamentals of object oriented programming. In inheritance, the child class gains the instance variables, methods, and states of the parent class... but what if we want to change those inherited elements? In our car example, we might have a Pedal class that defines a method called pushPedal(). When pushPedal() is called, the method preforms a default behavior (maybe it activates the breaks.) If we expand the Pedal class with a new class called AcceleratorPedal, the pushPedal() method suddenly becomes incorrect. An accelerator certainly shouldn't turn on the breaks! (Or you're gonna have a lot of lawsuits when you release your program, believe you me).
In this situation, we have to replace the behavior we inherited from Pedal with something new. This is done through a process called "Polymorphism" or "Function Overloading." You'll run into this all the time when you write UnrealScript. To borrow an explanation from Tim Sweeney:
"[Function overloading] refers to writing a new version of a function in a subclass. For example, say you're writing a script for a new kind of monster called a Demon. The Demon class, which you just created, expands the Pawn class. Now, when a pawn sees a player for the first time, the pawn's [SeePlayer()] function is called, so that the pawn can start attacking the player. This is a nice concept, but say you wanted to handle [SeePlayer()] differently in your new Demon class."
To do this, just redefine the function in the child class. When the class is instantiated, the object will have the new behavior, and not the parent behavior. If you don't want anyone to redefine a function you have added to a class, add the "final" specifier to the function's declaration:

```
  function final SeePlayer()
```

This prevents the script from overloading the function in derived classes and can be very useful in maintaining a consistant behavior in code you write. Tim Sweeney notes that it also results in a speed increase inside Unreal.

Bringing It All Together

So now you know the fundamentals of Object Oriented design and have a good idea of how objects relate to one another. What do you do next?

The best advice is to get hacking. Dive into the code and don't come up for air even if the promise of food, sleep, or sex looms near. Seek the zone. Or...you can always read Tim Sweeney's guide to UnrealScript. It goes into much greater detail about the syntax surrounding OO in Unreal. In addition, I suggest you find other resources on the net discussing OO. As I develop this paper, I'll try to come up with some good link, which I will list below. There are a lot of subtle elements of OO that can only be learned. Some aren't really supported by Unreal, some are. Some are merely ways of thinking.

And that brings me to my closing point. OO is as much a way of thinking as it is a way of programming. As you walk to school or drive to work, try imagining the relationship between things you see (a tree has leaves, a rose is a flower). This will greatly enhance your understanding of OO. Create complex relationships in your mind and then find ways of representing them in code.

To those who really understand it and really enjoy it, programming is a mental, physical, and spiritual task. It might sound wierd, but programming touches the fundamental ways in which we think and solve problems. If you can think in OO, then your mind is unrestricted when it comes to solving problems in OO. The more you use it, the more you will come to realize it is true.

OO can't solve everything, however. Just like any other way of thinking, the Object Oriented paradigm ignores certain elements of problem solving in order to strengthen its analogy to natural systems. Most likely, however, you will not be faced with these issues when you write UnrealScript, unless you are authoring one helluva transcending mod.

Unreal Classes

About this document

This is a quick attempt to describe the most important classes and variables in Unreal. It is currently very sparse but will be expanded over time.

Engine.Object

Purpose:

Serves as the base class of all objects in Unreal. All objects inherit from Object.

Variables:

- **Parent:** The object's parent object (for scoping).
- **ObjectFlags:** The object's flags, described in the Packages document.
- **Name:** The object's name.
- **Class:** The object's class.

Functions:

- **Log**: Writes a message to the log file, usually \Unreal\System\Unreal.log.
- **Warn:** Writes a script warning to the log file, including the current script and function.
- **Localize**: Returns a localized (internationally translated) string from a package's .int file.
- **GotoState**: Sets the object's current state, None means no state. If no label is specified, the Begin label is gone to.
- **IsInState**: Returns whether this object is in the specified state.
- **GetStateName**: Returns the name of this object's current stae, None if none.
- **Enable**: Enables a probe event. The only functions which work with Enable and Disable are: Spawned, Destroyed, GainedChild, LostChild, Trigger, UnTrigger, Timer, HitWall, Falling, Landed, ZoneChange, Touch, UnTouch, Bump, BeginState, EndState, BaseChange, Attach, Detach, ActorEntered, ActorLeaving, KillCredit, AnimEnd, EndedRotation, InterpolateEnd, EncroachingOn, EncroachedBy, FootZoneChange, HeadZoneChange, PainTimer, SpeechTimer, MayFall, Die, Tick, PlayerTick, Expired, SeePlayer, EnemyNotVisible, HearNoise, UpdateEyeHeight, SeeMonster, SeeFriend, SpecialHandling, BotDesireability.
- **Disable**: Disables a probe event.
- **GetPropertyText**: Converts the value of an arbirary variable to text.
- **SetPropertyText**: Sets the value of an arbitrary variable from text.
- **SaveConfig**: Saves the current values of all "config" variables to the Unreal.ini file.
- **ResetConfig**: Resets the values of the "config" variables to the originals in the Default.ini file.

Static Functions:

- **ClassIsChildOf**: Returns whether one class is a subclass of another class.
- **GetEnum**: Returns the nth element of an enumeration.
- **DynamicLoadObject**: Tries to load an object from a file, and returns it.

Events:

- **BeginState**: Called when a new state is entered.
- **EndState**: Called when the current state is ended.

Engine.Actor

Functions:

- **Error**: Causes a critical error; exits the engine.
- **SetCollision**: Changes the actor's collision flags.
- **SetCollisionSize**: Changes the actor's collision size.

- **Move**: Moves the actor by the specified displacement vector, handling collision (based on the collision flags), Touch and Bump notifications.
- **MoveSmooth**: Like move, but smoothly brushes against walls.
- **SetLocation**: Teleports the actor to a new location.
- **SetRotation**: Sets the actor's new rotation.
- **SetBase**: Sets the actor's Base. A base of None means that the actor moves alone; setting the base to another actor in the world causes this actor to move and rotate along with its base. An example of using a base is standing on a moving platform.
- **SetOwner**: Sets this actor's owner.
- **IsA**: Return whether this actor belongs in a named class.
- **PlayAnim**: Plays a named animation sequence in the actor's Mesh once (use the Mesh Viewer in UnrealEd to see the animation sequence names for a given mesh). The optional Rate scales the animation's default rate. If a nonzero TweenTime is specified, the animation is first tweened from whatever is currently displayed to the start of the named animation sequence, before the animation sequence plays. When the animation playing completes, it stops and calls your optional AnimEnd() event and causes any latent FinishAnim() calls to return.
- **LoopAnim**: Like PlayAnim, but loops the animation sequence over and over without end. As with PlayAnim, the AnimEnd() is called at the end of the sequence, and FinishAnim() calls return at the end of each iteration of the loop.
- **TweenAnim**: Tweens from whatever animation is currently being displayed, to the start of the specified animation sequence, then stops and calls AnimEnd() and releases any latent FinishAnim() calls.
- **IsAnimating**: Returns whether the actor's mesh is currently animating.
- **GetAnimGroup**: Returns the group name of the specified animation sequence.
- **SetPhysics**: Sets the actor's current physics mode.
- **BroadcastMessage**: Sends a text message to all players.

Latent Functions:
- **Sleep**: Waits for a certain (fractional) number of seconds to pass, then continues.
- **FinishAnim**: Waits for the currently playing or looping animation to reach the end of the sequence.
- **FinishInterpolation**: Only relevent with the physics mode PHYS_Interpolating. Waits until the next interpolation point is reached.

Iterator Functions
- AllActors: Iterates through all actors in the level with the specified tag; if you don't specify a tag, it iterates through all actors in the level. This is pretty slow if you specify a tag, and very slow if you don't.
- ChildActors: Iterates through all actors owned by this actor.
- BasedActors: Iterates through all actors directly based on this actor.
- TouchingActors: Iterates through all actors touching this actor. This is fast.
- TraceActors: Iterates through all actors hit by a trace along a line with an optional collision extent. This is pretty fast.
- RadiusActors: Iterates through all actors within a specified radius. This is fast.
- VisibleActors: Iterates through all actors visible to this actor (using a simple line trace, rather than exact mesh-to-mesh visible). This is very, very slow.
- VisibleCollidingActors: Iterates through all actors visible to this actor which have collision enabled; this is much faster than VisibleActors.

Events:
- **AnimEnd**: Called when the currently playing or looping animation reaches the end of the sequence.
- **Spawned**: Called when an actor is spawned during gameplay. Never called if the actor is preexisting when a level is loaded.
- **Destroyed**: Called immediately when an actor is destroyed. This event is not delayed like Java's Finalize() event is.

- **Expired**: Called immediately before Destroyed() when an actor's LifeTime has counted downward past zero.
- **GainedChild**: Another actor has done a SetOwner() to this actor.
- **LostChild**: An actor owned by this actor has been destroyed or done a SetOwner() to another actor.
- **Tick**: Called each iteration of the game loop; varies depending on frame rate. DeltaTime indicates how much time has passed.
- **Trigger**: This actor has been triggered by another actor Other, optionally caused by a pawn EventInstigator.
- **UnTrigger**: This actor has been untriggered by another actor Other, optionally caused by a pawn EventInstigator.
- **BeginEvent**: A simple protocol to indicate that this actor called a Trigger() function that began a complex chain of events. This enables switches to be coordinated with doors intelligently, i.e.: You press a switch. The switch moves in. It triggers a door which calls your BeginEvent() then slowly opens. The door then calls your EndEvent(). Now you move the button back out to give the user feedback that the button is finished with what it was doing.
- **EndEvent**: The event specified by BeginEvent has ended.
- **Timer**: If you call SetTimer, this Timer() function is called each time the timer has counted down from your time value.
- **HitWall**: This actor ran into a wall while moving.
- **Falling**: This actor has transitioned into PHYS_Falling.
- **Landed**: This actor has transitioned out of PHYS_Falling.
- **ZoneChange**: This actor has entered a new zone.
- **Touch**: This actor has begun touching (interpenetrating) another actor.
- **UnTouch**: This actor has stopped touching (interpenetrating) another actor.
- **Bump**: This actor has bumped into an impenetrable actor.
- **BaseChange**: This actor's base has changed.
- **Attach**: Some other actor has set its base to this actor.
- **Detach**: Some other actor based on this one is no longer based on this actor.
- **KillCredit**: This actor has successfully killed another actor.
- **SpecialHandling**: ?
- **EncroachingOn**: This actor is now encroaching on (overlapping with) another actor as a result of moving and it may return true to abort the move, or false to continue the move.
- **EncroachedBy**: This actor has been successfully encroached by (overlapped by) another actor. This is where players may opt to telefrag themselves or ignore the encroach.
- **InterpolateEnd**: Called when Physics==PHYS_Interpolating and the next interpolation point has been reached.
- **KilledBy**: Called when killed by another actor (killed just means dead, it doesn't mean destroyed).
- **TakeDamage**: A certain amount of damage has been imparted on this actor.
- **PreTeleport**: Called before this actor is teleported by a teleporter.
- **PostTeleport**: Called after this actor is teleported by a teleporter.

Physics Modes:

- To be written.

Game Startup:

When programming new actor scripts, you need to be wary of the order in which actors that pre-exist in a level (at load time) are initialized, and the order in which new actors are initialized when they are spawned. The exact order of initialization is:

1. The actor object is created and its variables are initialized to their default values (if spawning an actor), or loaded from a file (if loading a pre-existing actor).
2. The actor's location, rotation, and owner are set.
3. If the actor is being spawned, its Spawn() event is called.
4. The actor's PreBeginPlay() event is called.
5. The actor's BeginPlay() event is called.

6. The actor's zone is set, and any ZoneChange messages are sent to the actor, and any ActorEntered messages are sent to its zone. Prior to this point, the actor is not in a valid zone, and you must not call the Move, MoveSmooth, SetLocation, or SetRotation functions, or any functions which call them.
7. The actor's PostBeginPlay() event is called.
8. The actor's SetInitialState() event is called.
9. If the actor hasn't set its base yet, its base is set.

The Actor Class

Class Definition

class Actor expands Object
abstract
intrinsic;

The base class of all actors.

Imported Data

#exec Texture Import File=Textures\S_Actor.pcx Name=S_Actor Mips=Off Flags=2

Default texture loading.

Class Enums and Related Class Data Members

EDodgeDir

```
var enum EDodgeDir
{
   DODGE_None,
   DODGE_Left,
   DODGE_Right,
   DODGE_Forward,
   DODGE_Back,
   DODGE_Active,
   DODGE_Done
} DodgeDir;
```

Enumerators for dodging move direction.

EDrawType

```
var(Display) enum EDrawType
{
   DT_None,
   DT_Sprite,
   DT_Mesh,
   DT_Brush,
   DT_RopeSprite,
   DT_VerticalSprite,
   DT_Terraform,
   DT_SpriteAnimOnce,
} DrawType;
```

Enumerator for types of drawing effects.

EInputAction

```
enum EInputAction
{
   IST_None, // Not performing special input processing.
   IST_Press, // Handling a keypress or button press.
   IST_Hold, // Handling holding a key or button.
   IST_Release, // Handling a key or button release.
   IST_Axis, // Handling analog axis movement.
};
```

Enumerator for input system states.

EInputKey

```
enum EInputKey
{
/*00*/ IK_None, IK_LeftMouse, IK_RightMouse, IK_Cancel,
/*04*/ IK_MiddleMouse, IK_Unknown05, IK_Unknown06, IK_Unknown07,
/*08*/ IK_Backspace, IK_Tab, IK_Unknown0A, IK_Unknown0B,
/*0C*/ IK_Unknown0C, IK_Enter, IK_Unknown0E, IK_Unknown0F,
/*10*/ IK_Shift, IK_Ctrl, IK_Alt, IK_Pause,
/*14*/ IK_CapsLock, IK_Unknown15, IK_Unknown16, IK_Unknown17,
/*18*/ IK_Unknown18, IK_Unknown19, IK_Unknown1A, IK_Escape,
/*1C*/ IK_Unknown1C, IK_Unknown1D, IK_Unknown1E, IK_Unknown1F,
/*20*/ IK_Space, IK_PageUp, IK_PageDown, IK_End,
/*24*/ IK_Home, IK_Left, IK_Up, IK_Right,
/*28*/ IK_Down, IK_Select, IK_Print, IK_Execute,
/*2C*/ IK_PrintScrn, IK_Insert, IK_Delete, IK_Help,
/*30*/ IK_0, IK_1, IK_2, IK_3,
/*34*/ IK_4, IK_5, IK_6, IK_7,
/*38*/ IK_8, IK_9, IK_Unknown3A, IK_Unknown3B,
/*3C*/ IK_Unknown3C, IK_Unknown3D, IK_Unknown3E, IK_Unknown3F,
/*40*/ IK_Unknown40, IK_A, IK_B, IK_C,
/*44*/ IK_D, IK_E, IK_F, IK_G,
/*48*/ IK_H, IK_I, IK_J, IK_K,
/*4C*/ IK_L, IK_M, IK_N, IK_O,
/*50*/ IK_P, IK_Q, IK_R, IK_S,
/*54*/ IK_T, IK_U, IK_V, IK_W,
/*58*/ IK_X, IK_Y, IK_Z, IK_Unknown5B,
/*5C*/ IK_Unknown5C, IK_Unknown5D, IK_Unknown5E, IK_Unknown5F,
/*60*/ IK_NumPad0, IK_NumPad1, IK_NumPad2, IK_NumPad3,
/*64*/ IK_NumPad4, IK_NumPad5, IK_NumPad6, IK_NumPad7,
/*68*/ IK_NumPad8, IK_NumPad9, IK_GreyStar, IK_GreyPlus,
/*6C*/ IK_Separator, IK_GreyMinus, IK_NumPadPeriod, IK_GreySlash,
/*70*/ IK_F1, IK_F2, IK_F3, IK_F4,
/*74*/ IK_F5, IK_F6, IK_F7, IK_F8,
/*78*/ IK_F9, IK_F10, IK_F11, IK_F12,
/*7C*/ IK_F13 , IK_F14, IK_F15, IK_F16,
/*80*/ IK_F17, IK_F18, IK_F19, IK_F20,
/*84*/ IK_F21 , IK_F22, IK_F23, IK_F24,
/*88*/ IK_Unknown88, IK_Unknown89, IK_Unknown8A, IK_Unknown8B,
/*8C*/ IK_Unknown8C, IK_Unknown8D, IK_Unknown8E, IK_Unknown8F,
/*90*/ IK_NumLock, IK_ScrollLock , IK_Unknown92, IK_Unknown93,
/*94*/ IK_Unknown94, IK_Unknown95, IK_Unknown96, IK_Unknown97,
/*98*/ IK_Unknown98, IK_Unknown99, IK_Unknown9A, IK_Unknown9B,
/*9C*/ IK_Unknown9C, IK_Unknown9D, IK_Unknown9E, IK_Unknown9F,
/*A0*/ IK_LShift, IK_RShift, IK_LControl, IK_RControl,
/*A4*/ IK_UnknownA4, IK_UnknownA5, IK_UnknownA6, IK_UnknownA7,
/*A8*/ IK_UnknownA8, IK_UnknownA9, IK_UnknownAA, IK_UnknownAB,
/*AC*/ IK_UnknownAC, IK_UnknownAD, IK_UnknownAE, IK_UnknownAF,
/*B0*/ IK_UnknownB0, IK_UnknownB1, IK_UnknownB2, IK_UnknownB3,
/*B4*/ IK_UnknownB4, IK_UnknownB5, IK_UnknownB6, IK_UnknownB7,
/*B8*/ IK_UnknownB8, IK_UnknownB9, IK_Semicolon, IK_Equals,
/*BC*/ IK_Comma, IK_Minus, IK_Period, IK_Slash,
/*C0*/ IK_Tilde, IK_UnknownC1, IK_UnknownC2, IK_UnknownC3,
/*C4*/ IK_UnknownC4, IK_UnknownC5, IK_UnknownC6, IK_UnknownC7,
/*C8*/ IK_Joy1, IK_Joy2, IK_Joy3, IK_Joy4,
/*CC*/ IK_Joy5, IK_Joy6, IK_Joy7, IK_Joy8,
/*D0*/ IK_Joy9, IK_Joy10, IK_Joy11, IK_Joy12,
/*D4*/ IK_Joy13, IK_Joy14, IK_Joy15, IK_Joy16,
/*D8*/ IK_UnknownD8, IK_UnknownD9, IK_UnknownDA, IK_LeftBracket,
/*DC*/ IK_Backslash, IK_RightBracket, IK_SingleQuote, IK_UnknownDF,
/*E0*/ IK_JoyX, IK_JoyY, IK_JoyZ, IK_JoyR,
/*E4*/ IK_MouseX, IK_MouseY, IK_MouseZ, IK_MouseW,
/*E8*/ IK_JoyU, IK_JoyV, IK_UnknownEA, IK_UnknownEB,
/*EC*/ IK_MouseWheelUp, IK_MouseWheelDown, IK_Unknown10E, UK_Unknown10F,
/*F0*/ IK_JoyPovUp, IK_JoyPovDown, IK_JoyPovLeft, IK_JoyPovRight,
/*F4*/ IK_UnknownF4, IK_UnknownF5, IK_Attn, IK_CrSel,
/*F8*/ IK_ExSel, IK_ErEof, IK_Play, IK_Zoom,
/*FC*/ IK_NoName, IK_PA1, IK_OEMClear
};
```
Enumerator for input keys.

ELightEffect
```
var(Lighting) enum ELightEffect
{
   LE_None,
   LE_TorchWaver,
   LE_FireWaver,
   LE_WateryShimmer,
   LE_Searchlight,
   LE_SlowWave,
   LE_FastWave,
   LE_CloudCast,
   LE_StaticSpot,
   LE_Shock,
   LE_Disco,
   LE_Warp,
   LE_Spotlight,
   LE_NonIncidence,
   LE_Shell,
   LE_OmniBumpMap,
   LE_Interference,
   LE_Cylinder,
   LE_Rotor,
   LE_Unused
} LightEffect;
```
Enumerator for different spatial light effects to use.

ELightType
```
var(Lighting) enum ELightType
{
   LT_None,
   LT_Steady,
   LT_Pulse,
   LT_Blink,
   LT_Flicker,
   LT_Strobe,
   LT_BackdropLight,
   LT_SubtlePulse,
   LT_TexturePaletteOnce,
   LT_TexturePaletteLoop
} LightType;
```
Enumerator for different light modulation types.

EMusicTransition
```
enum EMusicTransition
{
   MTRAN_None,
   MTRAN_Instant,
   MTRAN_Segue,
   MTRAN_Fade,
   MTRAN_FastFade,
    MTRAN_SlowFade,
};
```
Enumerators for music transitions.

ENetRole
```
enum ENetRole
{
    ROLE_None,
    ROLE_DumbProxy,
    ROLE_SimulatedProxy,
    ROLE_AutonomousProxy,
    ROLE_Authority,
};
```
Enumerator for the diferent roles this actor may play in a net based game.

EPhysics
```
var(Movement) const enum EPhysics
{
    PHYS_None,
    PHYS_Walking,
    PHYS_Falling,
    PHYS_Swimming,
    PHYS_Flying,
    PHYS_Rotating,
    PHYS_Projectile,
    PHYS_Rolling,
    PHYS_Interpolating,
    PHYS_MovingBrush,
    PHYS_Spider,
    PHYS_Trailer
} Physics;
```
Enumerators for supported physics modes.

ERenderStyle
```
var(Display) enum ERenderStyle
{
    STY_None,
    STY_Normal,
    STY_Masked,
    STY_Translucent,
    STY_Modulated,
} Style;
```
Enumerators for different styles of rendering sprites and meshes.

ESoundSlot
```
enum ESoundSlot
{
    SLOT_None,
    SLOT_Misc,
    SLOT_Pain,
    SLOT_Interact,
    SLOT_Ambient,
    SLOT_Talk,
    SLOT_Interface,
};
```
Enumerators for different sound slots for actors.

ETravelType
```
enum ETravelType
{
    TRAVEL_Absolute, // Absolute URL.
```

TRAVEL_Partial, // Partial (carry name, reset server).
TRAVEL_Relative, // Relative URL.
};
Enumerator for travelling from server to server.

Class Data Members

Acceleration
var vector Acceleration;
Actor’s current acceleration.

bAlwaysRelevant
var(Advanced) bool bAlwaysRelevant;
Never destroy based on game.

bAlwaysTick
var Const bool bAlwaysTick;
Update even when players-only.

AmbientGlow
var(Display) byte AmbientGlow;
Ambient brightness, or 255=pulsing.

AmbientSound
var(Sound) sound AmbientSound;
Ambient sound effect.

AnimFrame
var(Display) float AnimFrame;
Current animation frame, 0.0 to 1.0.

AnimLast
var float AnimLast;
Last frame of current animation.

AnimMinRate
var float AnimMinRate;
Minimum rate for velocity-scaled animation.

AnimRate
var(Display) float AnimRate;
Animation rate in frames per second, 0=none, negative=velocity scaled.

AnimSequence
var(Display) name AnimSequence;
Current animation sequence being played.

bActorShadows
var(Lighting) bool bActorShadows;
This light casts actor shadows.

bAnimFinished
var bool bAnimFinished;
Unlooped animation sequence has finished.

bAnimLoop
var bool bAnimLoop;
Whether animation is looping.

bAnimNotify
var bool bAnimNotify;
Whether a notify is applied to the current sequence of animation.

bAssimilated
var transient const bool bAssimilated;
Actor dynamics are assimilated in world geometry.

bBlockActors
var(Collision) bool bBlockActors;
Blocks other nonplayer actors.

bBlockPlayers
var(Collision) bool bBlockPlayers;
Blocks other player actors.
bBounce
var(Movement) bool bBounce;
Actor bounces when it hits ground fast.
bCanTeleport
var(Advanced) bool bCanTeleport;
This actor can be teleported.
bCollideActors
var(Collision) const bool bCollideActors;
Collides with other actors.
bCollideWhenPlacing
var(Advanced) bool bCollideWhenPlacing;
This actor collides with the world when placing.
bCollideWorld
var(Collision) bool bCollideWorld;
Collides with the world.
bCorona
var(Lighting) bool bCorona;
This light uses Skin as a corona.
bDeleteMe
var const bool bDeleteMe;
Actor is about to be deleted.
bDirectional
var(Advanced) bool bDirectional;
Actor shows direction arrow during editing.
bDynamicLight
var bool bDynamicLight;
Temporarily treat this as a dynamic light.
bEdLocked
var bool bEdLocked;
Actor is locked in editor, therefore, no movement or rotation.
bEdShouldSnap
var(Advanced) bool bEdShouldSnap;
Actor should be snapped to grid in UnrealEd.
bEdSnap
var transient bool bEdSnap;
Actor is snapped to grid in UnrealEd.
bFixedRotationDir
var(Movement) bool bFixedRotationDir;
Actor has a fixed direction of rotation.
bForceStasis
var(Advanced) bool bForceStasis;
Actor is forced into stasis when not recently rendered, even if physics not PHYS_None or PHYS_Rotating.
bHidden
var(Advanced) bool bHidden;
Actor is hidden during gameplay.
bHiddenEd
var(Advanced) bool bHiddenEd;
Actor is hidden during editing.
bHighDetail
var(Advanced) bool bHighDetail;
Actor only shows up on high detail mode.

bHighlighted
var const bool bHighlighted;
Actor is highlighted in UnrealEd.
bHurtEntry
var bool bHurtEntry;
keep HurtRadius from being reentrant
bInterpolating
var bool bInterpolating;
Actor is performing an interpolating operation.
bIsItemGoal
var(Advanced) bool bIsItemGoal;
This actor counts in the "item" count.
bIsKillGoal
var(Advanced) bool bIsKillGoal;
This actor counts in the "death" toll.
bIsMover
var Const bool bIsMover;
Is a mover.
bIsPawn
var const bool bIsPawn;
Actor is a pawn.
bIsSecretGoal
var(Advanced) bool bIsSecretGoal;
This actor counts in the "secret" total.
bJustTeleported
var const bool bJustTeleported;
Used by engine physics - not valid for scripts.
bLensFlare
var(Lighting) bool bLensFlare;
Whether or not to use zone lens flare.
bLightChanged
var transient bool bLightChanged;
Recalculate this light's lighting now.
bMemorized
var const bool bMemorized;
Actor is remembered in UnrealEd.
bMeshCurvy
var(Display) bool bMeshCurvy;
Curvy mesh.
bMeshEnviroMap
var(Display) bool bMeshEnviroMap;
Environment-map the mesh.
bMovable
var(Advanced) bool bMovable;
Actor is capable of travelling among servers.
bNetFeel
var const bool bNetFeel;
Player collides with/feels this actor in network play. . Symmetric network flag, valid during replication only.
bNetHear
var const bool bNetHear;
Player hears this actor in network play. Symmetric network flag, valid during replication only.
bNetInitial
var const bool bNetInitial;
Initial network update flag. Symmetric network flag, valid during replication only.

bNetOptional
var const bool bNetOptional;
Actor should only be replicated if bandwidth available. Symmetric network flag, valid during replication only.
bNetOwner
var const bool bNetOwner;
Player owns this actor. Symmetric network flag, valid during replication only.
bNetSee
var const bool bNetSee;
Player sees it in network play. Symmetric network flag, valid during replication only.
bNoDelete
var(Advanced) const bool bNoDelete;
Actor cannot be deleted during play.
bNoSmooth
var(Display) bool bNoSmooth;
Don't smooth actor's texture.
bOnlyOwnerSee
var(Advanced) bool bOnlyOwnerSee;
Only owner can see this actor.
bParticles
var(Display) bool bParticles;
Mesh is a particle system.
bProjTarget
var(Collision) bool bProjTarget;
Projectiles should potentially target this actor.
bRotateToDesired
var(Movement) bool bRotateToDesired;
Actor rotates to DesiredRotation.
Brush
var const export model Brush;
Brush if DrawType=DT_Brush.
bSelected
var const bool bSelected;
Actor is selected in UnrealEd.
bShadowCast
var(Display) bool bShadowCast;
Casts shadows. *Not yet implemented.*
bSimulatedPawn
var const bool bSimulatedPawn;
True if this actor is a Pawn and a simulated proxy. Symmetric network flag, valid during replication only.
bSpecialLit
var(Lighting) bool bSpecialLit;
Only affects special-lit surfaces.
bStasis
var(Advanced) bool bStasis;
In standalone games, actor is turned off if not in a recently rendered zone, or the actor is turned off if bCanStasis == True and physics mode is PHYS_None or PHYS_Rotating.
bStatic
var(Advanced) const bool bStatic;
Actor does not move or change over time.
bTempEditor
var transient const bool bTempEditor;
Internal UnrealEd use.

bTicked
var transient const bool bTicked;
Actor has been updated.
bTimerLoop
var bool bTimerLoop;
If True then the actor Timer loops, else it is one-shot.
bTravel
var(Advanced) bool bTravel;
Actor is capable of travelling among servers.
bUnlit
var(Display) bool bUnlit;
Lights don't affect actor.
Buoyancy
var(Movement) float Buoyancy;
Actor’s water buoyancy value.
CollisionHeight
var(Collision) const float CollisionHeight;
Half-height cylinder.
CollisionRadius
var(Collision) const float CollisionRadius;
Radius of collision cyllinder.
ColLocation
var const vector ColLocation;
Actor's old location one move ago.
DesiredRotation
var(Movement) rotator DesiredRotation;
Physics subsystem will rotate pawn to this if bRotateToDesired.
DrawScale
var(Display) float DrawScale;
Scaling factor, 1.0=normal size.
Fatness
var(Display) byte Fatness;
Fatness (mesh distortion).
Group
var(Object) name Group;
???
InitialState
var(Object) name InitialState;
???
LifeSpan
var(Advanced) float LifeSpan;
How long the actor lives before dying, 0=forever. Used for executing timer-related events for the actor.
LightBrightness
var(LightColor) byte LightBrightness;
Light brightness value.
LightCone
var(LightColor) byte LightCone;
Light cone value.
LightHue
var(LightColor) byte LightHue;
Light hue value.
LightPeriod
var(Lighting) byte LightPeriod;
Light period value.

LightPhase
var(Lighting) byte LightPhase;
Light phase value.
LightRadius
var(Lighting) byte LightRadius;
Light radius value.
LightSaturation
var(LightColor) byte LightSaturation;
Light saturation value.
Location
var(Movement) const vector Location;
Actor's location; use Move to set.
Mass
var(Movement) float Mass;
Mass of this actor.
Mesh
var(Display) mesh Mesh;
Mesh if DrawType=DT_Mesh.
NetPriority
var(Networking) float NetPriority;
Higher priorities means update it more frequently.
OldAnimRate
var float OldAnimRate;
Animation rate of previous animation (= AnimRate until animation completes).
OldLocation
var const vector OldLocation;
Actor's old location one tick ago.
Owner
var const Actor Owner;
Owner of this actor.
PhysAlpha
var float PhysAlpha;
Interpolating position, 0.0-1.0.
PhysRate
var float PhysRate;
Interpolation rate per second.
PrePivot
var vector PrePivot;
Offset from box center for drawing.
RemoteRole
var(Networking) ENetRole RemoteRole;
Actor’s role in the current networked game on all other machines other than it’s original machine.
Role
var ENetRole Role;
Actor/s role in the current networked game on it’s original machine.
Rotation
var(Movement) const rotator Rotation;
Actor’s current rotation.
RotationRate
var(Movement) rotator RotationRate;
Change in rotation per second.
ScaleGlow
var(Display) float ScaleGlow;
Multiplies lighting.

SimAnim
var plane SimAnim;
Replicated to simulated proxies.
Skin
var(Display) texture Skin;
Special skin or environment map texture.
SoundPitch
var(Sound) byte SoundPitch;
Ambient sound pitch shift, 64.0=none.
SoundRadius
var(Sound) byte SoundRadius;
Radius of ambient sound.
SoundVolume
var(Sound) byte SoundVolume;
Volume of ambient sound.
Sprite
var(Display) texture Sprite;
Sprite texture if DrawType=DT_Sprite.
Texture
var(Display) texture Texture;
Misc. texture.
TimerCounter
var const float TimerCounter;
Counts up until it reaches TimerRate. Used for executing timer-related events for the actor.
TimerRate
var float TimerRate;
Timer event, 0=no timer. Used for executing timer-related events for the actor.
TransientSoundVolume
var(Sound) float TransientSoundVolume;
Volume of regular (non-ambient) sounds.
TweenRate
var float TweenRate;
Animation tween-into rate.
Velocity
var(Movement) vector Velocity;
Actor’s player’s pulse rate . . . just wanted to see if you were awake. ;-) Actor’s current velocity.
VolumeBrightness
var(Lighting) byte VolumeBrightness;
Light volume brightness value.
VolumeFog
var(Lighting) byte VolumeFog;
Light volume fog value.
VolumeRadius
var(Lighting) byte VolumeRadius;
Light volume radius value.

Scriptable Class Data Members

Base
var const Actor Base;
Moving brush actor we're standing on.
Event
var(Events) name Event;
The event this actor causes.

Instigator
var Pawn Instigator;
Pawn responsible for damage.
Inventory
var Inventory Inventory;
Inventory chain.
LevelInfo
var const LevelInfo Level;
Level this actor is on.
Region
var const PointRegion Region;
Region this actor is in.
Tag
var(Events) name Tag;
Actor's tag name.
Target
var Actor Target;
Actor we're aiming at (other uses as well).
XLevel
var const Level XLevel;
Level object.

Gameplay Scenarios Class Data Members

bDifficulty0
var(Filter) bool bDifficulty0;
Actor appears in difficulty level 0 gameplay scenarios.
bDifficulty1
var(Filter) bool bDifficulty1;
Actor appears in difficulty level 1 gameplay scenarios.
bDifficulty2
var(Filter) bool bDifficulty2;
Actor appears in difficulty level 2 gameplay scenarios.
bDifficulty3
var(Filter) bool bDifficulty3;
Actor appears in difficulty level 3 gameplay scenarios.
bSinglePlayer
var(Filter) bool bSinglePlayer;
Actor appears in single player gameplay scenarios.
bNet
var(Filter) bool bNet;
Actor appears in regular network play gameplay scenarios.
bNetSpecial
var(Filter) bool bNetSpecial;
Actor appears in special network gameplay mode.
OddsOfAppearing
var(Filter) float OddsOfAppearing;
Chance the actor will appear in relevant gameplay modes, values of 0.0 to 1.0, 1.0=always appears.

Internal Use Class Data Members

CollisionTag, ExtraTag, LightingTag, NetTag, OtherTag, SpecialTag
var const transient int CollisionTag;
var const transient int ExtraTag;
var const transient int LightingTag;
var const transient int NetTag;
var const transient int OtherTag;

var const transient int SpecialTag;
All tag variables for internal use. Purposes unknown.
Deleted
var const actor Deleted;
Next actor in just-deleted chain.
LatentActor
var const actor LatentActor;
Internal latent function use.
LatentByte
var const byte LatentByte;
Internal latent function use.
LatentFloat
var const float LatentFloat;
Internal latent function use.
LatentInt
var const int LatentInt;
Internal latent function use.
MiscNumber
var const byte MiscNumber;
Internal use.
StandingCount
var const byte StandingCount;
Count of actors standing on this actor.
Touching[4]
var const actor Touching[4];
List of touching actors.

Class Structure Definitions

PointRegion

```
struct PointRegion
{
   var zoneinfo Zone; // Zone.
   var int iLeaf; // Bsp leaf.
   var byte ZoneNumber; // Zone number, to be eliminated!!
};
```

Identifies a unique convex volume in the world.

Class Information for Network Replication

???. I have my guesses about this special piece of code for dealing with networking. I still think it best we wait for Tim Sweeney to enlighten us. If you have some thoughts on this piece of code please email me at Valiant.

```
replication
{
   // Relationships.
   reliable if( Role==ROLE_Authority )
   Owner, Role, RemoteRole;
   reliable if( Role==ROLE_Authority && bNetOwner )
   bNetOwner, Inventory;
   // Ambient sound.
   reliable if( Role==ROLE_Authority )
   AmbientSound;
   reliable if( Role==ROLE_Authority && AmbientSound!=None )
   SoundRadius, SoundVolume, SoundPitch;
   // Collision.
   reliable if( Role==ROLE_Authority )
   bCollideActors;
   reliable if( Role==ROLE_Authority )
   bCollideWorld;
   reliable if( Role==ROLE_Authority && bCollideActors )
   bBlockActors, bBlockPlayers;
   reliable if( Role==ROLE_Authority && (bCollideActors || bCollideWorld) )
   CollisionRadius, CollisionHeight;
   // Location.
   unreliable if( Role==ROLE_Authority && (bNetInitial || bSimulatedPawn ||
   RemoteRole<ROLE_SimulatedProxy) )
   Location, Rotation;
   unreliable if( Role==ROLE_Authority && (bNetInitial || bSimulatedPawn) &&
   RemoteRole==ROLE_SimulatedProxy )
   Base;
   // Velocity.
   unreliable if( (RemoteRole==ROLE_SimulatedProxy && (bNetInitial ||
bSimulatedPawn)) || bIsMover     )
   Velocity;
   // Physics.
   unreliable if( RemoteRole==ROLE_SimulatedProxy && bNetInitial )
   Physics, Acceleration, bBounce;
   unreliable if( RemoteRole==ROLE_SimulatedProxy && Physics==PHYS_Rotating &&
bNetInitial )
   bFixedRotationDir, bRotateToDesired, RotationRate, DesiredRotation;
   // Animation.
   unreliable if( DrawType==DT_Mesh && (RemoteRole<=ROLE_SimulatedProxy) )
   AnimSequence;
   unreliable if( DrawType==DT_Mesh && (RemoteRole==ROLE_SimulatedProxy) )
   bAnimNotify;
   unreliable if( DrawType==DT_Mesh && (RemoteRole<ROLE_AutonomousProxy) )
   SimAnim, AnimMinRate;
   // Rendering.
   reliable if( Role==ROLE_Authority )
   bHidden, bOnlyOwnerSee;
   unreliable if( Role==ROLE_Authority ) // and see...
   Texture, DrawScale, PrePivot, DrawType, AmbientGlow, Fatness, ScaleGlow, bUnlit,
bNoSmooth,
   bShadowCast, bActorShadows;
   unreliable if( Role==ROLE_Authority && DrawType==DT_Sprite && !bHidden &&
      (!bOnlyOwnerSee     || bNetOwner) )
   Sprite;
   unreliable if( Role==ROLE_Authority && DrawType==DT_Mesh )
   Mesh, bMeshEnviroMap, bMeshCurvy, Skin;
   unreliable if( Role==ROLE_Authority && DrawType==DT_Brush )
   Brush;
   // Lighting.
   unreliable if( Role==ROLE_Authority )
   LightType;
   unreliable if( Role==ROLE_Authority && LightType!=LT_None )
   LightEffect, LightBrightness, LightHue, LightSaturation,
   LightRadius, LightPeriod, LightPhase, LightCone,
   VolumeBrightness, VolumeRadius,
   bSpecialLit;
   // Messages
   unreliable if( Role < ROLE_Authority )
   BroadcastMessage;
}
```

Class Intrinsic Operators and Functions
Vector Operators

-
intrinsic(211) static final preoperator vector - (vector A);
Negation operator

*****
*intrinsic(212) static final operator(16) vector * (vector A, float B);*
*intrinsic(213) static final operator(16) vector * (float A, vector B);*
*intrinsic(296) static final operator(16) vector * (vector A, vector B);*
Multiplication operator

/
intrinsic(214) static final operator(16) vector / (vector A, float B);
Division operator

+
intrinsic(215) static final operator(20) vector + (vector A, vector B);
Addition operator

-
intrinsic(216) static final operator(20) vector - (vector A, vector B);
Subtraction operator

<<
intrinsic(275) static final operator(22) vector << (vector A, rotator B);
Left rotation operator

>>
intrinsic(276) static final operator(22) vector >> (vector A, rotator B);
Right rotation operator

==
intrinsic(217) static final operator(24) bool == (vector A, vector B);
Equality operator

!=
intrinsic(218) static final operator(26) bool != (vector A, vector B);
Inequality operator

Dot
intrinsic(219) static final operator(16) float Dot (vector A, vector B);
Dot product operator

Cross
intrinsic(220) static final operator(16) vector Cross (vector A, vector B);
Cross product operator

***=**
*intrinsic(221) static final operator(34) vector *= (out vector A, float B);*
*intrinsic(297) static final operator(34) vector *= (out vector A, vector B);*
Multiplication assignment operator

/=
intrinsic(222) static final operator(34) vector /= (out vector A, float B);
Division assignment operator

+=
intrinsic(223) static final operator(34) vector += (out vector A, vector B);
Addition assignment operator

-=
intrinsic(224) static final operator(34) vector -= (out vector A, vector B);
Subtraction assignment operator

Vector Functions

VSize
intrinsic(225) static final function float VSize (vector A);
Get the vertical size of a vector.

Normal
intrinsic(226) static final function vector Normal (vector A);
Get the normal of a vector.
Invert
intrinsic(227) static final function Invert (out vector X, out vector Y, out vector Z);
Invert a vector.
Vrand
intrinsic(252) static final function vector VRand ( );
???. Please email Valiant if you know what this does.
MirrorVectorByNormal
intrinsic(300) static final function vector MirrorVectorByNormal(vector Vect, vector Normal);
Return the mirror about the given normal of a vector.

Rotator Operators

==
intrinsic(142) static final operator(24) bool == (rotator A, rotator B);
Equality operator
!=
intrinsic(203) static final operator(26) bool != (rotator A, rotator B);
Inequality operator
*****
*intrinsic(287) static final operator(16) rotator * (rotator A, float B);*
*intrinsic(288) static final operator(16) rotator * (float A, rotator B);*
Multiplication operator
/
intrinsic(289) static final operator(16) rotator / (rotator A, float B);
Division operator
***=**
*intrinsic(290) static final operator(34) rotator *= (out rotator A, float B);*
Multiplication assignment operator
/=
intrinsic(291) static final operator(34) rotator /= (out rotator A, float B);
Division assignment operator
+
intrinsic(316) static final operator(20) rotator + (rotator A, rotator B);
Addition operator
-
intrinsic(317) static final operator(20) rotator - (rotator A, rotator B);
Subtraction operator
+=
intrinsic(318) static final operator(34) rotator += (out rotator A, rotator B);
Addition assignment operator
-=
intrinsic(319) static final operator(34) rotator -= (out rotator A, rotator B);
Subtraction assignment operator

Rotator Functions

GetAxes
intrinsic(229) static final function GetAxes (rotator A, out vector X, out vector Y, out vector Z);
Get X,Y and Z axes values for the given rotator.
GetUnAxes
intrinsic(230) static final function GetUnAxes (rotator A, out vector X, out vector Y, out vector Z);
???. Email me at Valiant if you know what this does.

RotRand
intrinsic(320) static final function rotator RotRand (optional bool bRoll);
Return randomized rotator.

<u>Iterator Functions</u>

AllActors
intrinsic(304) final iterator function AllActors(class<actor> BaseClass, out actor Actor, optional name MatchTag);
Iterates through all actors in the level with the specified tag. If you don't specify a tag, it iterates through all actors in the level. This is pretty slow if you specify a tag, and very slow if you don't.

BasedActors
intrinsic(306) final iterator function BasedActors(class<actor> BaseClass, out actor Actor);
Iterates through all actors directly based on this actor.

ChildActors
intrinsic(305) final iterator function ChildActors(class<actor> BaseClass, out actor Actor);
Iterates through all actors owned by this actor.

RadiusActors
intrinsic(310) final iterator function RadiusActors(class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc);
Iterates through all actors within a specified radius. This is fast.

TouchingActors
intrinsic(307) final iterator function TouchingActors(class<actor> BaseClass, out actor Actor);
Iterates through all actors touching this actor. This is fast.

TraceActors
intrinsic(309) final iterator function TraceActors(class<actor> BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent);
Iterates through all actors hit by a trace along a line with an optional collision extent. This is pretty fast.

VisibleActors
intrinsic(311) final iterator function VisibleActors(class<actor> BaseClass, out actor Actor, optional float Radius, optional vector Loc);
Iterates through all actors visible to this actor (using a simple line trace, rather than exact mesh-to-mesh visible). This is very, very slow.

VisibleCollidingActors
intrinsic(312) final iterator function VisibleCollidingActors(class<actor> BaseClass, out actor Actor, optional float Radius, optional vector Loc);
Iterates through all actors visible to this actor which have collision enabled; this is much faster than VisibleActors.

<u>Class Member Functions</u>

AnimEnd
event AnimEnd();
Called when the currently playing or looping animation reaches the end of the sequence.

Attach
event Attach(Actor Other);
Some other actor has set its base to this actor.

BaseChange
event BaseChange();
This actor's base has changed.

BeginEvent
event BeginEvent();
A simple protocal to indicate that this actor called a Trigger() function that began a complex

chain of events. This enables switches to be coordinated with doors intelligently. For example, you press a switch, the door switch moves in, it triggers a door which calls your BeginEvent() then slowly opens, then the door calls your EndEvent(). Nowyou move the button back out to give the user feedback that the button is finished with what it was doing.

BeginPlay

event BeginPlay();

Called when a level is entered for playing.

BroadcastMessage

```
function BroadcastMessage( coerce string[240] Msg, bool bBeep )
{
   local Pawn P;
   if ( Level.Game.AllowsBroadcast(self, Len(Msg)) )
      for( P=Level.PawnList; P!=None; P=P.nextPawn )
         if( P.bIsPlayer )
            P.ClientMessage( Msg );
}
```

Broadcast a message to all players.

Bump

event Bump(Actor Other);

This actor has bumped into an impenetrable actor.

Destroy

intrinsic(279) final function bool Destroy();

Destroy this actor. Returns true if destroyed, false if indestructable. Destruction is latent. It occurs at the end of the tick.

Destroyed

event Destroyed();

Called immediately when an actor is destroyed.

Detach

event Detach(Actor Other);

Some other actor based on this one is no longer based on this actor.

EncroachedBy

event EncroachedBy(actor Other);

This actor has been successfully encroached by (overlapped by) another actor. This is where players may opt to telefrag themselves or ignore the encroach.

EncroachingOn

event bool EncroachingOn(actor Other);

This actor is now encroaching on (overlapping with) another actor as a result of moving and it may return true to abort the move, or false to continue the move.

EndEvent

event EndEvent();

The event called when BeginEvent() has ended.

EndedRotation

event EndedRotation();

Called when the actor’s current rotation has ended.

Error

intrinsic(233) final function Error(coerce string[255] S);

Handle an error and kill this one actor. Causes a critical error; exits the engine.

Expired

event Expired();

Called immediately before Destroyed() when an actor’s LifeTime has counted downward past zero.

Falling

event Falling();

This actor has transitioned into PHYS_Falling.

FinishAnim
intrinsic(261) final latent function FinishAnim();
Waits for the currently playing or looping animation to reach the end of the sequence.
FinishInterpolation
intrinsic(301) final latent function FinishInterpolation();
Only relevent with the physics mode PHYS_Interpolating. Waits until the next interpolation point is reached.
GainedChild
event GainedChild(Actor Other);
Another actor has done a SetOwner() to this actor.
GetAnimGroup
intrinsic(293) final function name GetAnimGroup(name Sequence);
Returns the group name of the specified animation sequence.
GetMapName
intrinsic(539) final function string[32] GetMapName(string[32] NameEnding, string[32] MapName, int Dir);
Returns the name of the map specified.
GetNextSkin
intrinsic(545) final function string[64] GetNextSkin(string[64] Prefix, string[64] CurrentSkin, int Dir);
Returns the name of the next skin in the current file of skins being accessed.
HitWall
event HitWall(vector HitNormal, actor HitWall);
This actor ran into a wall while moving.
HurtRadius

```
final function HurtRadius( float DamageAmount, float DamageRadius, name
DamageName, float Momentum, vector HitLocation )
{
local actor Victims;
local float damageScale, dist;
local vector dir;
   if( bHurtEntry )
      return;
   bHurtEntry = true;
   foreach VisibleCollidingActors( class 'Actor', Victims, DamageRadius, HitLocation )
   {
      if( Victims != self )
      {
         dir = Victims.Location - HitLocation;
         dist = FMax(1,VSize(dir));
         dir = dir/dist;
         damageScale = 1 - FMax(0,(dist - Victims.CollisionRadius)/DamageRadius);
         Victims.TakeDamage(damageScale * DamageAmount,Instigator, Victims.Location -
0.5 *                    (Victims.CollisionHeight + Victims.CollisionRadius) * dir,
(damageScale * Momentum *                    dir), DamageName);
      }
   }
   bHurtEntry = false;
}
```

Hurt actors within the radius.
InterpoloateEnd
event InterpolateEnd(actor Other);
Called when Physics==PHYS_Interpolating and the next interpolation point has been reached.

IsA
intrinsic(303) final function bool IsA(name ClassName);
Return whether this actor belongs in a named class.
IsAnimating
intrinsic(282) final function bool IsAnimating();
Returns whether the actor's mesh is currently animating.
KilledBy
event KilledBy(pawn EventInstigator);
Called when killed by another actor (killed just means dead, it doesn't mean destroyed).
KillCredit
event KillCredit(Actor Other);
This actor has successfully killed another actor.
Landed
event Landed(vector HitNormal);
Ouch! That's gotta hurt! This actor has transitioned out of PHYS_Falling.
LoopAnim
intrinsic(260) final function LoopAnim(name Sequence, optional float Rate, optional float TweenTime, optional float MinRate);
Loops the animation dequence forever. The AnimEnd() is called at the end of the sequence, and FinishAnim() calls return at the end of each iteration of the loop.
LostChild
event LostChild(Actor Other);
An actor owned by this actor has been destroyed or done a SetOwner() to another actor.
MakeNoise
intrinsic(512) final function MakeNoise(float Loudness);
Inform other creatures that you've made a noise they might hear (they are sent a HearNoise message). Senders of MakeNoise should have an instigator if they are not pawns.
Move
intrinsic(266) final function bool Move(vector Delta);
Moves the actor by the specified displacement vector, handling collision (based on the collision flags), Touch and Bump notifications.
MoveSmooth
intrinsic(3969) final function bool MoveSmooth(vector Delta);
Like move, but smoothly brushes against walls.
PlayAnim
intrinsic(259) final function PlayAnim(name Sequence, optional float Rate, optional float TweenTime);
Plays a named animation sequence in the actor's Mesh once. The optional Rate scales the animation's default rate. If a nonzero TweenTime is specified, the animation is first tweened from whatever is currently displayed to the start of the named animation sequence, before the animation sequence plays. When the animation playing completes, it stops and calls your optional AnimEnd() event and causes any latent FinishAnim() calls to return.
PlayerCanSeeMe
intrinsic(532) final function bool PlayerCanSeeMe();
PlayerCanSeeMe returns true if some player has a line of sight to actor's location.
PlaySound
intrinsic(264) final function PlaySound(sound Sound, optional ESoundSlot Slot, optional float Volume, optional bool bNoOverride, optional float Radius, optional float Pitch);
Play a sound effect.
PostBeginPlay
event PostBeginPlay();
Called immediately after gameplay begins.
PostTeleport
event PostTeleport(Teleporter OutTeleporter);
Called after this actor is teleported by a teleporter.

PreBeginPlay

```
event PreBeginPlay()
{
// Handle autodestruction if desired.
   if( !bAlwaysRelevant && (Level.NetMode != NM_Client) &&
!Level.Game.IsRelevant(Self))
      Destroy();
}
```

Called immediately before gameplay begins.

PreTeleport

event bool PreTeleport(Teleporter InTeleporter);

Called before this actor is teleported by a teleporter.

SetBase

intrinsic(298) final function SetBase(actor NewBase);

Sets the actor's Base. A base of None means that the actor moves alone; setting the base to another actor in the world causes this actor to move and rotate along with its base. An example of using a base is standing on a moving platform.

SetCollision

intrinsic(262) final function SetCollision(optional bool NewColActors, optional bool NewBlockActors, optional bool NewBlockPlayers);

Changes the actor's collision flags.

SetCollisionSize

intrinsic(283) final function bool SetCollisionSize(float NewRadius, float NewHeight);

Changes the actor's collision size.

SetInitialState

```
simulated event SetInitialState()
{
   if( InitialState!='' )
      GotoState( InitialState );
   else
      GotoState( 'Auto' );
}
```

Called after PostBeginPlay.

SetLocation

intrinsic(267) final function bool SetLocation(vector NewLocation);

Teleports the actor to a new location.

SetOwner

intrinsic(272) final function SetOwner(actor NewOwner);

Sets this actor's owner.

SetPhysics

intrinsic(3970) final function SetPhysics(EPhysics newPhysics);

Sets the actor's current physics mode.

SetRotation

intrinsic(299) final function bool SetRotation(rotator NewRotation);

Sets the actor's new rotation.

SetTimer

intrinsic(280) final function SetTimer(float NewTimerRate, bool bLoop);

Causes Timer() events every NewTimerRate seconds.

Sleep

intrinsic(256) final latent function Sleep(float Seconds);

Waits for a certain number of seconds to pass, then continues.

Spawn

intrinsic(278) final function actor Spawn(class<actor> SpawnClass, optional actor SpawnOwner, optional name SpawnTag, optional vector SpawnLocation, optional rotator SpawnRotation);

Spawn an actor. Returns an actor of the specified class, not of class Actor (this is hardcoded

in the compiler). Returns None if the actor could not be spawned (either the actor wouldn't fit in the specified location, or the actor list is full). Defaults to spawning at the spawner's location.
Spawned
event Spawned();
Called when an actor is spawned during gameplay. Never called if the actor is preexisting when a level is loaded.
SpecialHandling
event Actor SpecialHandling(Pawn Other);
???.
TakeDamage
event TakeDamage(int Damage, Pawn EventInstigator, vector HitLocation, vector Momentum, name DamageType);
A certain amount of damage has been imparted on this actor.
Tick
event Tick(float DeltaTime);
Called each iteration of the game loop; varies depending on frame rate. DeltaTime indicates how much time has passed.
Trigger
event Trigger(Actor Other, Pawn EventInstigator);
Roy Roger’s house. ;-) This actor has been triggered by another actor Other, optionally caused by a pawn EventInstigator.
Timer
event Timer();
If you call SetTimer, this Timer function is called each time the timer has counted down from your time value.
Touch
event Touch(Actor Other);
This actor has begun touching (interpenetrating) another actor. Now, now, I know what you’re thinking, let’s keep it clean.
Trace
intrinsic(277) final function Actor Trace(out vector HitLocation, out vector HitNormal, vector TraceEnd, optional vector TraceStart, optional bool bTraceActors, optional vector Extent);
Traced a line and see what it collides with first. Takes this actor's collision properties into account. Returns first hit actor, Level if hit level, or None if hit nothing.
TravelPostAccept
event TravelPostAccept();
Called when carried into a new level, after AcceptInventory.
TravelPreAccept
event TravelPreAccept();
Called when carried onto a new level, before AcceptInventory.
TweenAnim
intrinsic(294) final function TweenAnim(name Sequence, float Time);
Tweens from whatever animation is currently being displayed, to the start of the specified animation sequence, then stops and calls AnimEnd() and releases any latent FinishAnim() calls.
UnTouch
event UnTouch(Actor Other);
This actor has stopped touching (interpenetrating) another actor.
UnTrigger
event UnTrigger(Actor Other, Pawn EventInstigator);
This actor has been untriggered by another actor Other, optionally caused by a pawn EventInstigator.

ZoneChange
event ZoneChange(ZoneInfo NewZone);
This actor has entered a new zone.

The Object Class

Class Definition

class Object
intrinsic;

Serves as the base class of all objects in Unreal.

Data Members

ObjectInternal
var intrinsic private const int ObjectInternal[6];
???

Parent
var intrinsic const parent;
The object's parent object, used for scoping purposes.

ObjectFlags
var intrinsic const int ObjectFlags;
The object's flags, described in the package documentation.

Name
var(object) intrinsic const editconst name Name;
The object's name.

Class
var(object) intrinsic const editconst class Class;
The object's class name.

Class Structure Definitions

BoundingBox
struct BoundingBox
{
var vector Min, Max;
var byte IsValid;
};
A bounding box, used for collision purposes.

BoundingVolume
struct BoundingVolume expands boundingbox
{
var plane Sphere;
};
A bounding box sphere together, used for collision purposes.

Color
struct Color
{
var() config byte R, G, B, A;
};
A color value structure.

Coords
struct Coords
{
var() config vector Origin, XAxis, YAxis, ZAxis;
};
An arbitrary coordinate system in 3d space.

DynamicArray
struct DynamicArray
{

var const int Num, Max, Ptr;
};
A dynamic array.
DynamicString
struct DynamicString
{
};
A dynamic string.
Guid
struct Guid
{
var int A, B, C, D;
};
A globally unique identifier.
Plane
struct Plane expands Vector
{
var() config float W;
};
A plane definition in 3d space.
Rotator
struct Rotator
{
var() config int Pitch, Yaw, Roll;
};
An orthogonal rotation in 3d space.
Scale
struct Scale
{
var() config vector Scale;
var() config float SheerRate;
var() config enum ESheerAxis
  {
  SHEER_None,
  SHEER_XY,
  SHEER_XZ,
  SHEER_YX,
  SHEER_YZ,
  SHEER_ZX,
  SHEER_ZY,
  } SheerAxis;
};
Structure used for scaling and sheering.
Vector
struct Vector
{
var() config float X, Y, Z;
};
A point or direction vector in 3d space.

<u>**Class Constants**</u>
MaxInt
const MaxInt = 0x7fffffff;
Value of the maximum integer.

Pi
const Pi = 3.1415926535897932;
Value used for PI.

Class Intrinsic Operators and Functions

Bool Operators

!
intrinsic(129) static final preoperator bool ! (bool A);
Logical negation operator
==
intrinsic(242) static final operator(24) bool == (bool A, bool B);
Equality operator
!=
intrinsic(243) static final operator(26) bool != (bool A, bool B);
Inequality operator
&&
intrinsic(130) static final operator(30) bool && (bool A, skip bool B);
Logical AND operator
^^
intrinsic(131) static final operator(30) bool ^^ (bool A, bool B);
??? if you know please email me the answer, TIA, Valiant
||
intrinsic(132) static final operator(32) bool || (bool A, skip bool B);
Logical OR operator

Byte Operators

***=**
*intrinsic(133) static final operator(34) byte *= (out byte A, byte B);*
Multiplication operator
/=
intrinsic(134) static final operator(34) byte /= (out byte A, byte B);
Division operator
+=
intrinsic(135) static final operator(34) byte += (out byte A, byte B);
Addition operator
-=
intrinsic(136) static final operator(34) byte -= (out byte A, byte B);
Subtraction operator
++
intrinsic(137) static final preoperator byte ++ (out byte A);
Increment preoperator
--
intrinsic(138) static final preoperator byte -- (out byte A);
Decrement preoperator
++
intrinsic(139) static final postoperator byte ++ (out byte A);
Increment postoperator
--
intrinsic(140) static final postoperator byte -- (out byte A);
Increment postoperator

Integer operators

~
intrinsic(141) static final preoperator int ~ (int A);
One’s complement operator

-
intrinsic(143) static final preoperator int - (int A);
Unary minus operator
*****
*intrinsic(144) static final operator(16) int * (int A, int B);*
Multiplication operator
/
intrinsic(145) static final operator(16) int / (int A, int B);
Division operator
+
intrinsic(146) static final operator(20) int + (int A, int B);
Addition operator
-
intrinsic(147) static final operator(20) int - (int A, int B);
Subtraction operator
<<
intrinsic(148) static final operator(22) int << (int A, int B);
Left shift operator, or, for vectors, forward vector transformation
>>
intrinsic(149) static final operator(22) int >> (int A, int B);
Right shift operator, or, for vectors, reverse vector transformation
<
intrinsic(150) static final operator(24) bool < (int A, int B);
Less than operator
>
intrinsic(151) static final operator(24) bool > (int A, int B);
Greater than operator
<=
intrinsic(152) static final operator(24) bool <= (int A, int B);
Less or equal operator
>=
intrinsic(153) static final operator(24) bool >= (int A, int B);
Greater or equal operator
==
intrinsic(154) static final operator(24) bool == (int A, int B);
Equality operator
!=
intrinsic(155) static final operator(26) bool != (int A, int B);
Inequality operator
&
intrinsic(156) static final operator(28) int & (int A, int B);
Bitwise AND operator
^
intrinsic(157) static final operator(28) int ^ (int A, int B);
Bitwise exclusive OR operator
|
intrinsic(158) static final operator(28) int | (int A, int B);
Logical negation operator
***=**
*intrinsic(159) static final operator(34) int *= (out int A, float B);*
Multiplication assignment operator
/=
intrinsic(160) static final operator(34) int /= (out int A, float B);
Division assignment operator

+=
intrinsic(161) static final operator(34) int += (out int A, int B);
Addition assignment operator
-=
intrinsic(162) static final operator(34) int -= (out int A, int B);
Subtraction assignment operator
++
intrinsic(163) static final preoperator int ++ (out int A);
Increment preoperator
--
intrinsic(164) static final preoperator int -- (out int A);
Decrement preoperator
++
intrinsic(165) static final postoperator int ++ (out int A);
Increment postoperator
--
intrinsic(166) static final postoperator int -- (out int A);
Decrement postoperator

Integer functions

Clamp
intrinsic(251) static final function int Clamp (int V, int A, int B);
Returns the first number clamped to the interval from A to B
Max
intrinsic(250) static final function int Max (int A, int B);
Returns the maximum of two integers
Min
intrinsic(249) static final function int Min (int A, int B);
Returns the minimum of two integers
Rand
intrinsic(167) static final Function int Rand (int Max);
Returns pseudo-random integer from 0 to MaxInt

Float operators

-
intrinsic(169) static final preoperator float - (float A);
Unary minus operator
******
*intrinsic(170) static final operator(12) float ** (float A, float B);*
Exponentiation
*****
*intrinsic(171) static final operator(16) float * (float A, float B);*
Multiplication operator
/
intrinsic(172) static final operator(16) float / (float A, float B);
Division operator
%
intrinsic(173) static final operator(18) float % (float A, float B);
Modulus operator
+
intrinsic(174) static final operator(20) float + (float A, float B);
Addition operator
-
intrinsic(175) static final operator(20) float - (float A, float B);
Subtraction operator

<
intrinsic(176) static final operator(24) bool < (float A, float B);
Less than operator
>
intrinsic(177) static final operator(24) bool > (float A, float B);
Greater than operator
<=
intrinsic(178) static final operator(24) bool <= (float A, float B);
Less than equal operator
>=
intrinsic(179) static final operator(24) bool >= (float A, float B);
Greater than equal operator
==
intrinsic(180) static final operator(24) bool == (float A, float B);
Equality operator
~=
intrinsic(210) static final operator(24) bool ~= (float A, float B);
Approximate equality (within 0.0001), case insensitive equality
!=
intrinsic(181) static final operator(26) bool != (float A, float B);
Inequality operator
***=**
*intrinsic(182) static final operator(34) float *= (out float A, float B);*
Multiplication assignment operator
/=
intrinsic(183) static final operator(34) float /= (out float A, float B);
Division assignment operator
+=
intrinsic(184) static final operator(34) float += (out float A, float B);
Addition assignment operator
-=
intrinsic(185) static final operator(34) float -= (out float A, float B);
Subtraction assignment operator

Float functions

Abs
intrinsic(186) static final function float Abs (float A);
Returns the absolute value of its argument.
Sin
intrinsic(187) static final function float Sin (float A);
Returns the sine of its argument.
Cos
intrinsic(188) static final function float Cos (float A);
Returns the cosine of its argument.
Tan
intrinsic(189) static final function float Tan (float A);
Returns the tangent of its argument.
Atan
intrinsic(190) static final function float Atan (float A);
Returns the arctangent of its argument.
Exp
intrinsic(191) static final function float Exp (float A);
Returns the exponential of its argument.
Loge
intrinsic(192) static final function float Loge (float A);
Returns the natural logarithm of its argument.

Sqrt
intrinsic(193) static final function float Sqrt (float A);
Returns the square root of its argument.
Square
intrinsic(194) static final function float Square(float A);
Returns the square of its argument.
FRand
intrinsic(195) static final function float FRand ();
Returns a pseudo-random number equal to or between 0 and 1.0.
FMin
intrinsic(244) static final function float FMin (float A, float B);
Returns the minimum argument.
FMax
intrinsic(245) static final function float FMax (float A, float B);
Returns the maximum argument.
FClamp
intrinsic(246) static final function float FClamp(float V, float A, float B);
Returns the first number clamped to the interval from A to B
Lerp
intrinsic(247) static final function float Lerp (float Alpha, float A, float B);
Returns the linear interpolation between A and B
Smerp
intrinsic(248) static final function float Smerp (float Alpha, float A, float B);
Returns an Alpha-smooth nonlinear interpolation between A and B

String operators

$
intrinsic(228) static final operator(40) string[255] $ (coerce string[255] A, coerce String[255] B);
String Concatenation
<
intrinsic(197) static final operator(24) bool < (string[255] A, string[255] B);
Less than operator
>
intrinsic(198) static final operator(24) bool > (string[255] A, string[255] B);
Greater than operator
<=
intrinsic(199) static final operator(24) bool <= (string[255] A, string[255] B);
Less than or equal operator
>=
intrinsic(200) static final operator(24) bool >= (string[255] A, string[255] B);
Greater than or equal operator
==
intrinsic(201) static final operator(24) bool == (string[255] A, string[255] B);
Equality operator
!=
intrinsic(202) static final operator(26) bool != (string[255] A, string[255] B);
Inequality operator
~=
intrinsic(168) static final operator(24) bool ~= (string[255] A, string[255] B);
Aproximate equality (within 0.0001), or, for strings, case insensitive equality

String functions

Asc
intrinsic static final function int Asc (string[255] S);
Returns the character code corresponding to the first letter in a string.

Caps
intrinsic(209) static final function string[255] Caps (coerce string[255] S);
Capitalizes the string.
Chr
intrinsic static final function string[16] Chr (int i);
Returns the character associated with the specified character code.
InStr
intrinsic(205) static final function int InStr (coerce string[255] S, coerce string[255] t);
Returns the position of the first occurrence of one string within another.
Left
intrinsic(207) static final function string[255] Left (coerce string[255] S, int i);
Returns a specified number of characters from the left side of a string.
Len
intrinsic(204) static final function int Len (coerce string[255] S);
Returns the length of the argument.
Mid
intrinsic(206) static final function string[255] Mid (coerce string[255] S, int i, optional int j);
Returns a string containing a specified number of characters from a string.
Right
intrinsic(208) static final function string[255] Right (coerce string[255] S, int i);
Returns a specified number of characters from the right side of a string.

<u>Object operators</u>
==
intrinsic(114) static final operator(24) bool == (Object A, Object B);
Equality operator
!=
intrinsic(119) static final operator(26) bool != (Object A, Object B);
Inequality operator

<u>Name operators</u>
==
intrinsic(254) static final operator(24) bool == (name A, name B);
Equality operator
!=
intrinsic(255) static final operator(26) bool != (name A, name B);
Inequality operator
<u>Class Member Functions</u>
BeginState
event BeginState();
Remarks
Called immediately when entering a state, while within the GotoState call that caused the state change.
ClassIsChildOf
intrinsic(258) static final function bool ClassIsChildOf(class TestClass, class ParentClass);
Parameters
class TestClass
class ParentClass
Return Value
True if class is a child, else False.
Remarks
Returns whether one class is a subclass of another class.
Disable
intrinsic(118) final function Disable(name ProbeFunc);

Parameters
name ProbeFunc
Remarks
Disables a probe event.
DynamicLoadObject
intrinsic static final function object DynamicLoadObject(string[32] ObjectName, class ObjectClass);
Parameters
String[32] ObjectName
Class ObjectClass
Return Value
If the object is successfully loaded it is returned, else None.
Remarks
Tries to load an object from a file, and returns it.
Enable
intrinsic(117) final function Enable(name ProbeFunc);
Parameters
Name ProbeFunc
Remarks
Enables a probe event. The only functions which work with Enable and Disable are: Spawned, Destroyed, GainedChild, LostChild, Trigger, UnTrigger, Timer, HitWall, Falling, Landed, ZoneChange, Touch, UnTouch, Bump, BeginState, EndState, BaseChange, Attach, Detach, ActorEntered, ActorLeaving, KillCredit, AnimEnd, EndedRotation, InterpolateEnd, EncroachingOn, EncroachedBy, FootZoneChange, HeadZoneChange, PainTimer, SpeechTimer, MayFall, Die, Tick, PlayerTick, Expired, SeePlayer, EnemyNotVisible, HearNoise, UpdateEyeHeight, SeeMonster, SeeFriend, SpecialHandling, BotDesireability.
EndState
event EndState();
Remarks
Called immediately before going out of the current state, while within the GotoState call that caused the state change.
GetEnum
intrinsic static final function name GetEnum(object E, int i);
Parameters
Object E
Int i
Return Value
Name of the nth element of an enumeration.
Remarks
Returns the nth element of an enumeration.
GetPropertyText
intrinsic final function string[192] GetPropertyText(string[32] PropName);
Parameters
String[32] PropName
Return Value
String[192] value of the argument.
Remarks
Converts the value of an arbitrary variable to text.
GetStateName
intrinsic(284) final function name GetStateName();
Return Value
name
Remarks
Returns the name of this object's current state, None if none.
GotoState
intrinsic(113) final function GotoState(optional name NewState, optional name Label);

Parameters
Optional name NewState
Optional name Label
Remarks
Sets the object’s current state, None means no state. If no label is specified, the Begin label is gone to.
IsInState
intrinsic(281) final function bool IsInState(name TestState);
Parameters
Name TestState
Return Value
bool
Remarks
Returns whether this object is in the specified state.
Localize
intrinsic static function string[192] Localize(name SectionName, name KeyName, name PackageName);
Parameters
Name SectionName
Name KeyName
Name PackageName
Return Value
String[192]
Remarks
Returns a localized (internationally translated) string from a package’s .int file.
Log
intrinsic(231) final static function Log(coerce string[240] S, optional name Tag);
Parameters
Coerce string[240] S
Optional name Tag
Remarks
Writes a message to the log file, usually \Unreal\System\Unreal.log.
ResetConfig
intrinsic(543) final function ResetConfig();
Remarks
Resets the values of the "config" variables to the originals in the Default.ini file.
SaveConfig
intrinsic(536) final function SaveConfig();
Remarks
Saves the current values of all "config" variables to the Unreal.ini file.
SetPropertyText
intrinsic final function SetPropertyText(string[32] PropName, string[192] PropValue);
Parameters
String[32] PropName
String[192] PropValue
Remarks
Sets the value of an arbitrary variable from text.
Warn
intrinsic(232) final static function Warn(coerce string[240] S);
Parameters
Coerce string[240] S
Remarks
Writes a script warning to the log file, including the current script and function.

Tutorials

The Dispersion Pistol

OK ppl, for the first mod, we're going to give you an easy time. It's pretty simple really, mostly to show off the power of OO programming and the amazing UnrealEd development environment.
There are two classes, one for the new weapon, the DP2k, and one for it's ammo. The functions declared override the parent class and so, in order to keep the game running, you need to copy the code across. Of course, you could use the 'super' prefix, but that would execute all of the code before you had time to alter it.
In the first class, the only alteration is the type of ammo to use. Changed from DAmmo to DP2kAmmo, which is the second class we create. The second class is the dispersion ammo modified slightly, check the original code to see what alterations have been made. Play around with the code to see if you can alter the damage and create more explosions!
Note : The default properties are altered in the UnrealEd environment and only show when dumped to ASCII.

```
// FILE : DP2k.uc =================================\
class DP2k expands DispersionPistol;
// tacosalad - The following code is borrowed from the parent class.
// Changes to the orginal code are commented below.
function Projectile ProjectileFire( class < projectile > ProjClass, float = ProjSpeed, bool bWarn)
{
        local Vector Start, X, Y, Z;
        local DispersionAmmo da;
        local float Mult;
        Owner.MakeNoise( Pawn( Owner ).SoundDampening);
        if ( Amp != None ) Mult = Amp.UseCharge(80);
                else Mult = 1.0;
        GetAxes( Pawn( owner ).ViewRotation, X, Y, Z);
        Start = Owner.Location + CalcDrawOffset() + FireOffset.X * X + = FireOffset.Y * Y + FireOffset.Z
        * Z;
        AdjustedAim = pawn( owner ).AdjustAim( ProjSpeed, Start, AimError, True, = (3.5 * FRand()-1 <
        PowerLevel));
        if ( (PowerLevel == 0) || (AmmoType.AmmoAmount < 10) )
                da = Spawn( class 'DP2kAmmo', , , Start, AdjustedAim);      // tacosalad - Use the
                appropriate ammo
        else
        {
                if ( (PowerLevel == 1) && AmmoType.UseAmmo(2) )
                        da = Spawn( class 'DAmmo2' , , , Start, AdjustedAim);
                if ( (PowerLevel == 2) && AmmoType.UseAmmo(4) )
                        da = Spawn( class 'DAmmo3' , , , Start, AdjustedAim);
                if ( (PowerLevel == 3) && AmmoType.UseAmmo(5) )
                        da =3D Spawn( class 'DAmmo4' , , , Start ,AdjustedAim);
                if ( (PowerLevel > 4) && AmmoType.UseAmmo(6) )
                        da = Spawn( class 'DAmmo5' , , , Start,AdjustedAim);
        }
        if ( da != None )
        {
                if ( Mult > 1.0 ) da.InitSplash( FMin( da.damage * Mult, 100 ) );
        }
}
defaultproperties
{
        PickupMessage = "You got the DP-2K"
        PlayerViewOffset = ( Y = 2.000000 )
}

// FILE : DP2kAmmo.uc ================================\
class DP2kAmmo expands DispersionAmmo;
// tacosalad - The following code is a modification of the Explode function
// found in the code for the FlakShell.
function Explode(vector HitLocation, vector HitNormal)
{
	local vector start;
	HurtRadius( damage, 150, 'exploded', MomentumTransfer, HitLocation);
	start = Location + 100 * HitNormal;
	Spawn( class 'FlameExplosion' , , , Start);
	Spawn( class 'FlameExplosion' , , , Start);
	Spawn( class 'FlameExplosion' , , , Start);
	Destroy();
}
defaultproperties
{
	ParticleType = Class 'UnrealI.ParticleBurst2'
}
```

Ok, now you've done that you need to compile it. Now to test it. Firstly try booting the game, and in the console type 'SUMMON TACO-DP2K.DP2k'. This should spawn the DP2k right in front of you. Go pick it up. Now, because the weapon is based on the dispersion pistol, Unreal should allow you to activate it by changing to the pistol (#1), and pressing the key again will bring it up. The weapon gets added to a list of weapons on the key.
Alternatively, download the mod and try out the new map. It has the DP2k in it, and an edited baddy!
Best of luck!

Proximity Mines

Straight out of Goldeneye, come the Proximity mines. I just started with UnrealScript and this is my first mod. When I first started, I orignally named the class "Timedmines" but quickly changed a lot around and forgot about it, so thats why the class is named "TimedMines" instead of "ProxityMines". I'll probably be working on some more versions of the Proximity Mines that will include different things. I also changed the ASMD to launch the Proximity Mines so when using TML (ASMD replacement), don't forget to hit 'S' for secondary fire. Well thats enough of my rambling... have fun!
The Proximity Mines will remain a weapon because it is much easier to keep them as a Projectile instead of making them an inventory item. Well I didn't make the blind explosion because I have other mods I want to do so this will probably be the last revision of the Proximity Mines unless I think of something spectacular :)
If you want to summon the Proximity Mines, hit during a game and type "summon proxmines.tml"

```
//===========================================================================

class TimedMine expands Grenade;


//Starting here.  This usually controls the smoke release of the grenade.  If the level detail mode is high,
//then allow smoke.  However, our proximity mine doesn't smoke at this point, so remove it!

simulated function BeginPlay()

		{
				SmokeRate = 0;
```

```
        }


//This is called just after the mine enters the level.  The purpose of it is to calculate the direction
//and spped of the projectile when fired from the gun.

function PostBeginPlay()
{
        local vector X,Y,Z;
        local rotator RandRot;

        PlayAnim('WingIn');
        GetAxes(Instigator.ViewRotation,X,Y,Z);


        Calculate velocity

        Velocity = X * (Instigator.Velocity Dot X)*0.4 + Vector(Rotation) * Speed +
                FRand() * 100 * Vector(Rotation);
        Velocity.z += 210;
        SetTimer(0+FRand()*0,false);            //Grenade begins unarmed


        Calculate spin

        RandRot.Pitch = FRand() * 1400 - 700;
        RandRot.Yaw = FRand() * 1400 - 700;
        RandRot.Roll = FRand() * 1400 - 700;
        MaxSpeed = 1000;
        Velocity = Velocity >> RandRot;
        RandSpin(50000);
        bCanHitOwner = False;


        If it hits water - don't explode

        if (Instigator.HeadRegion.Zone.bWaterZone)
        {
                bHitWater = True;
                Disable('Tick');
                Velocity=0.6*Velocity;
        }

}



//Occurs when the mine hits a solid object (which is not a player)

simulated function HitWall( vector HitNormal, actor Wall )
{
        bCanHitOwner = True;
        Velocity = 0.8*(( Velocity dot HitNormal ) * HitNormal * (-2.0) + Velocity);   // Reflect off Wall
w/damping
        RandSpin(100000);
        speed = VSize(Velocity);
        RotationRate = RotRand();
        SetCollisionSize(100,100);

        if ( Level.NetMode != NM_DedicatedServer )
                PlaySound(ImpactSound, SLOT_Misc, FMax(0.5, speed/800) );
        if ( Velocity.Z > 400 )
                Velocity.Z = 0.5 * (400 + Velocity.Z);
        else if ( speed > 0 )
        {
                bBounce = False;
                SetPhysics(PHYS_None);

        }

}



//The tick function happens once every game 'tick'.

simulated function Tick(float DeltaTime)
{
        local BlackSmoke b;


        If in water - then stop

        if (bHitWater)
        {
                Disable('Tick');
                Return;
        }

        Count += DeltaTime;

        if ( (Count>Frand()*SmokeRate+SmokeRate+NumExtraGrenades*0.03) &&
(Level.NetMode!=NM_DedicatedServer) )
        {
                b.RemoteRole = ROLE_None;
                b.RemoteRole = ROLE_None;
                Count=0;
        }


        if ( (Physics == PHYS_None) && (WarnTarget != None) && WarnTarget.bCanDuck
                && (WarnTarget.Physics == PHYS_Walking) && (WarnTarget.Acceleration !=
vect(0,0,0)) )
                WarnTarget.Velocity = WarnTarget.Velocity + 2 * DeltaTime *
WarnTarget.GroundSpeed
                                                                        *
Normal(WarnTarget.Location - Location);
}



When explodes *doh*
```

```
function Explode(vector HitLocation, vector HitNormal)
{
		local vector start;

		HurtRadius(damage, 150, 'exploded', MomentumTransfer, HitLocation);
		start = Location;
		Spawn( class'SpriteBallExplosion',,,Start);
		Destroy();
}


//Passes damage to others in level

function TakeDamage( int NDamage, Pawn instigatedBy, Vector hitlocation,
								Vector momentum, name damageType)
{
		Spawn(Class 'SpriteBallExplosion',,,Location+Vect(0,0,9));
		HurtRadius(50, 50, 'exploded', 0, Location);
		Destroy();
}


//=============================================================================
// TazerProj2.
//=============================================================================

class TazerProj2 expands TazerProj;


//An altered secondary fire mechanism

function Explode(vector HitLocation,vector HitNormal)
{
		PlaySound(ImpactSound, SLOT_Misc, 0.5,,, 0.5+FRand());
		HurtRadius(Damage, 1, 'jolted', MomentumTransfer, Location );
		if (Damage < 60) Spawn(class'RingExplosion3',,, HitLocation+HitNormal*8,rotator(HitNormal));
		else Spawn(class'RingExplosion3',,, HitLocation+HitNormal*8,rotator(HitNormal));
		Destroy();
}


// Tml.
//=============================================================================
//This is just an ASMD weapon, altered to fire the timed mine created above
//=============================================================================

class Tml expands ASMD;

function PreBeginPlay()
{
		Super.PreBeginPlay();
		AltProjectileClass = class'TimedMine';
		AltProjectileSpeed = class'TimedMine'.Default.speed;
}
```

Dispersion Pistol Overwiew

The Dispersion Pistol is the default weapon used in Unreal DM, and probably the first weapon you will find in Unreal SP. It has two modes of fire, both projectile based. Primary fires a energy projectile based upon the current mode of the weapon (the Dispersion Pistol can be upgraded by another item). Secondary fires a more powerful projectile based upon how long the Pawn charges.

```
//=============================================================================
// DispersionPistol.
//=============================================================================
class DispersionPistol expands Weapon;

//Whenever you come across something that begins with "#exec", that tells the compiler
//to import some file. Generally the things that are imported are models/meshes, sounds,
//and textures...anything that you find in Unreal that Unrealed doesn't create.  You do
//not need to use any "#exec" lines if you are not importing any new models/meshes, or
//if you import textures and/or sounds inside of Unrealed.

//For all weapons there are 3 different models/meshes that the Unreal package uses.
//Player view is the model that is shown to whoever is holding the weapon.  Pickup view
//is the model that is shown when a weapon is sitting in a level waiting to be picked
//up.  3rd person is the model shown to other Players when they look at a Pawn using that
//particular weapon.

//The player view model is held up close to the Player's body, so generally it is a
//fraction of the size that pickup and 3rd person versions, but all 3 of the models come
//from the same source.  All of the following code imports first the mesh, sets the origin
//of the model (for rotation etc.), scales the model to the appropriate size, identifies
//all of the animations of the model (SEQUENCE), and then imports a texture and applies it
//to the mesh.

// player view version
#exec MESH IMPORT MESH=DPistol ANIVFILE=MODELS\dgun_a.3D
DATAFILE=MODELS\dgun_d.3D X=0 Y=0 Z=0
#exec MESH ORIGIN MESH=DPistol X=0 Y=0 Z=0 YAW=-64 PITCH=0
#exec MESHMAP SCALE MESHMAP=DPistol X=0.005 Y=0.005 Z=0.01
#exec MESH SEQUENCE MESH=DPistol SEQ=All     STARTFRAME=0  NUMFRAMES=141
#exec MESH SEQUENCE MESH=DPistol SEQ=Select1 STARTFRAME=0  NUMFRAMES=11
GROUP=Select
#exec MESH SEQUENCE MESH=DPistol SEQ=Shoot1  STARTFRAME=11 NUMFRAMES=3
#exec MESH SEQUENCE MESH=DPistol SEQ=Idle1   STARTFRAME=14  NUMFRAMES=2
#exec MESH SEQUENCE MESH=DPistol SEQ=Down1   STARTFRAME=16  NUMFRAMES=5
#exec MESH SEQUENCE MESH=DPistol SEQ=PowerUp1 STARTFRAME=21  NUMFRAMES=4
#exec MESH SEQUENCE MESH=DPistol SEQ=Still    STARTFRAME=25  NUMFRAMES=5
#exec MESH SEQUENCE MESH=DPistol SEQ=Select2 STARTFRAME=30  NUMFRAMES=11
GROUP=Select
#exec MESH SEQUENCE MESH=DPistol SEQ=Shoot2  STARTFRAME=41 NUMFRAMES=3
#exec MESH SEQUENCE MESH=DPistol SEQ=Idle2   STARTFRAME=44  NUMFRAMES=2
#exec MESH SEQUENCE MESH=DPistol SEQ=Down2   STARTFRAME=46  NUMFRAMES=5
#exec MESH SEQUENCE MESH=DPistol SEQ=PowerUp2 STARTFRAME=51  NUMFRAMES=9
#exec MESH SEQUENCE MESH=DPistol SEQ=Select3 STARTFRAME=60  NUMFRAMES=11
GROUP=Select
#exec MESH SEQUENCE MESH=DPistol SEQ=Shoot3  STARTFRAME=71 NUMFRAMES=3
#exec MESH SEQUENCE MESH=DPistol SEQ=Idle3   STARTFRAME=74  NUMFRAMES=2
#exec MESH SEQUENCE MESH=DPistol SEQ=Down3   STARTFRAME=76  NUMFRAMES=5
#exec MESH SEQUENCE MESH=DPistol SEQ=PowerUp3 STARTFRAME=81  NUMFRAMES=9
```

```
#exec MESH SEQUENCE MESH=DPistol SEQ=Select4 STARTFRAME=90  NUMFRAMES=11
GROUP=Select
#exec MESH SEQUENCE MESH=DPistol SEQ=Shoot4  STARTFRAME=101 NUMFRAMES=3
#exec MESH SEQUENCE MESH=DPistol SEQ=Idle4   STARTFRAME=104  NUMFRAMES=2
#exec MESH SEQUENCE MESH=DPistol SEQ=Down4   STARTFRAME=106  NUMFRAMES=5
#exec MESH SEQUENCE MESH=DPistol SEQ=PowerUp4 STARTFRAME=111  NUMFRAMES=9
#exec MESH SEQUENCE MESH=DPistol SEQ=Select5 STARTFRAME=120 NUMFRAMES=11
GROUP=Select
#exec MESH SEQUENCE MESH=DPistol SEQ=Shoot5  STARTFRAME=131 NUMFRAMES=3
#exec MESH SEQUENCE MESH=DPistol SEQ=Idle5   STARTFRAME=134 NUMFRAMES=2
#exec MESH SEQUENCE MESH=DPistol SEQ=Down5   STARTFRAME=136  NUMFRAMES=5

#exec TEXTURE IMPORT NAME=DPistol1 FILE=MODELS\dgun.PCX GROUP="Skins"
#exec OBJ LOAD FILE=Textures\SmokeEffect2.utx PACKAGE=UNREALI.SEffect2
#exec MESHMAP SETTEXTURE MESHMAP=DPistol NUM=1  TEXTURE=DPistol1
#exec MESHMAP SETTEXTURE MESHMAP=DPistol NUM=0
TEXTURE=Unreali.SEffect2.SmokeEffect2


// pickup version
#exec MESH IMPORT MESH=DPistolPick ANIVFILE=MODELS\dgunlo_a.3D
DATAFILE=MODELS\dgunlo_d.3D X=0 Y=0 Z=0
#exec MESH ORIGIN MESH=DPistolPick X=0 Y=0 Z=0 YAW=64
#exec MESH SEQUENCE MESH=DPistolPick SEQ=All  STARTFRAME=0  NUMFRAMES=1
#exec TEXTURE IMPORT NAME=DPistol1 FILE=MODELS\dgun.PCX GROUP="Skins"
#exec MESHMAP SCALE MESHMAP=DPistolPick X=0.04 Y=0.04 Z=0.08
#exec MESHMAP SETTEXTURE MESHMAP=DPistolPick NUM=1 TEXTURE=DPistol1

// 3rd person perspective version
#exec MESH IMPORT MESH=DPistol3rd ANIVFILE=MODELS\dgunlo_a.3D
DATAFILE=MODELS\dgunlo_d.3D X=0 Y=0 Z=0
#exec MESH ORIGIN MESH=DPistol3rd X=0 Y=-200 Z=-110 YAW=-64 ROLL=9
#exec MESH SEQUENCE MESH=DPistol3rd SEQ=All  STARTFRAME=0  NUMFRAMES=6
#exec MESH SEQUENCE MESH=DPistol3rd SEQ=Still  STARTFRAME=0  NUMFRAMES=1
#exec MESH SEQUENCE MESH=DPistol3rd SEQ=Shoot1 STARTFRAME=0  NUMFRAMES=6
#exec TEXTURE IMPORT NAME=DPistol1 FILE=MODELS\dgun.PCX GROUP="Skins"
#exec MESHMAP SCALE MESHMAP=DPistol3rd X=0.025 Y=0.025 Z=0.05
#exec MESHMAP SETTEXTURE MESHMAP=DPistol3rd NUM=1 TEXTURE=DPistol1

//Here "#exec" is used to import in sounds for the weapon, one for charging, another for
//firing, and another for when a Pawn picks it up.

#exec AUDIO IMPORT FILE="Sounds\dispersion\Powerup3.WAV" NAME="PowerUp3"
GROUP="Dispersion"
#exec AUDIO IMPORT FILE="Sounds\dispersion\DShot1.WAV" NAME="DispShot"
GROUP="Dispersion"
#exec AUDIO IMPORT FILE="Sounds\dispersion\Dpickup2.WAV" NAME="DispPickup"
GROUP="Dispersion"

//Here after all the "#exec" lines the global variables are defined.  These variables
//stay alive as long as their is a Spawned copy of this particular class in the game,
//whereas the variables defined inside of functions later on are local, and die as
//soon as the function is finished.
//Powerlevel stores what level of upgrade this particular Dispersion Pistol is, WeaponPos
//is an extra vector that is used later on to keep track of the current position of the
//weapon.  Count and ChargeSize are both used to calculate the amount of charge when the
```

```
//Pistol is fired in Secondary mode.  cl1 and cl2 are instances of ChargeLight, that as far
//as I can tell aren't used for anything useful in this version of the code.  Amp is a copy
//used to determine/use the inventory Amplifier, if the Pawn has an active version.
//PowerUpSound is the power up sound (tough one eh?).

var travel int PowerLevel;
var vector WeaponPos;
var float Count,ChargeSize;
var ChargeLight cl1,cl2;
var Pickup Amp;
var Sound PowerUpSound;

//The Engine calls three functions whenever an Actor is created, PreBeginPlay(),
//BeginPlay(), and PostBeginPlay().  They are very useful for executing code no matter
//what, every time an instance of this Actor is created.  Here PostBeginPlay() is used
//to store what Projectile to fire for both Primary and Secondary Fire.

function PostBeginPlay()
{
		Super.PostBeginPlay();
		ProjectileClass = class'DispersionAmmo';
		AltProjectileClass = class'DispersionAmmo';
		ProjectileSpeed = class'DispersionAmmo'.Default.speed;
		AltProjectileSpeed = class'DispersionAmmo'.Default.speed;
}

//RateSelf() is an AI function used solely by Bots (at the moment) to determine what
//priority this weapon should take when compared to others in the Bot's inventory.

function float RateSelf( out int bUseAltMode )
{
		local float rating;

		if ( Pawn(Owner).bShootSpecial )
				return 1000;

		if ( Amp != None )
				rating = 3.1 * AIRating;
		else
				rating = AIRating;

		if ( AmmoType.AmmoAmount <=0 )
				return 0.05;
		if ( Pawn(Owner).Enemy == None )
		{
				bUseAltMode = 0;
				return rating * (PowerLevel + 1);
		}

		bUseAltMode = int( FRand() < 0.3 );
		return rating * (PowerLevel + 1);
}

//SuggestAttackStyle() determines whether the bot should go with primary or secondary fire.  (I
//am not completely sure on this....)
```

```
// return delta to combat style
function float SuggestAttackStyle()
{
        local float EnemyDist;
        local Inventory Inv;

        if ( !Pawn(Owner).bIsPlayer || (PowerLevel > 0) )
                return 0;

        return -0.3;
}

//HandlePickupQuery() is a great function that allows an inventory item to perform some action
//whenever its Owner picks up another item.  The Dispersion Pistol uses it here to check if
//its Owner just picked up a Powerup, and if so it will 'upgrade' itself.

function bool HandlePickupQuery( inventory Item )
{
        if ( Item.IsA('WeaponPowerup') )
        {
                AmmoType.AddAmmo(AmmoType.MaxAmmo);
                Pawn(Owner).ClientMessage(Item.PickupMessage);
                Item.PlaySound (PickupSound);
                if ( PowerLevel<4 )
                {
                        ShakeVert = Default.ShakeVert + PowerLevel;
                        PowerUpSound = Item.ActivateSound;
                        if ( Pawn(Owner).Weapon == self )
                        {
                                PowerLevel++;
                                GotoState('PowerUp');
                        }
                        else if ( (Pawn(Owner).Weapon != Self) &&
!Pawn(Owner).bNeverSwitchOnPickup )
                        {
                                Pawn(Owner).Weapon.PutDown();
                                Pawn(Owner).PendingWeapon = self;
                                GotoState('PowerUp', 'Waiting');
                        }
                        else PowerLevel++;
                }
                Item.SetRespawn();
                return true;
        }
        else
                return Super.HandlePickupQuery(Item);
}

//SetSwitchPriority() is a function used to automatically determine what weapon should come
//up when you either pick up a new weapon, or run out of ammo for the current weapon.

function SetSwitchPriority(pawn Other)
{
        local int i;
        local name MyType;
        if (PowerLevel == 0)
                MyType = 'DispersionPistol';
        else if (PowerLevel == 1)
                MyType = 'DispersionPower1';
        else if (PowerLevel == 2)
                MyType = 'DispersionPower2';
        else if (PowerLevel == 3)
                MyType = 'DispersionPower3';
        else if (PowerLevel == 4)
                MyType = 'DispersionPower4';
        else
                MyType = 'DispersionPower5';

        if ( PlayerPawn(Other) != None )
                for ( i=0; i<20; i++)
                        if ( PlayerPawn(Other).WeaponPriority[i] == MyType )
                        {
                                AutoSwitchPriority = i;
                                return;
                        }
}

function BecomePickup()
{
        Amp = None;
        Super.BecomePickup();
}

//PlayFiring() (and PlayAltFiring()) are basically the animation and sound control functions
//that are called whenever the weapon is fired.  The Dispersion Pistol plays a firing sound,
//shakes it's Owner's view via ShakeView(), and then based upon the PowerLevel it plays a
//firing animation.

function PlayFiring()
{
        AmmoType.GoToState('Idle2');
        Owner.PlaySound(AltFireSound, SLOT_None, 1.8*Pawn(Owner).SoundDampening,,,1.2);
        if ( PlayerPawn(Owner) != None )
                PlayerPawn(Owner).ShakeView(ShakeTime, ShakeMag, ShakeVert);
        if (PowerLevel==0)
                PlayAnim('Shoot1',0.4,0.2);
        else if (PowerLevel==1)
                PlayAnim('Shoot2',0.3,0.2);
        else if (PowerLevel==2)
                PlayAnim('Shoot3',0.2, 0.2);
        else if (PowerLevel==3)
                PlayAnim('Shoot4',0.1,0.2);
        else if (PowerLevel==4)
                PlayAnim('Shoot5',0.1,0.2);
}

//ProjectileFire() is the function used by all Projectile-based weapons to fire the
//appropriate projectile, in the appropriate direction, from the correct starting point.

function Projectile ProjectileFire(class ProjClass, float ProjSpeed, bool bWarn)
{
	local Vector Start, X,Y,Z;
	local DispersionAmmo da;
	local float Mult;

	Owner.MakeNoise(Pawn(Owner).SoundDampening);

//If the Pistol's Owner has an Amplifier then set the damage multiplier equal to whatever
//Amp.UseCharge(80) returns.

	if (Amp!=None) Mult = Amp.UseCharge(80);
	else Mult=1.0;

//GetAxes is a function that returns normals along the axis of a given rotator.  Here is used
//to store the axes from the Pistol's Owner's ViewRotation (the direction the Owner is looking).

	GetAxes(Pawn(owner).ViewRotation,X,Y,Z);

//Start is calculated by first using the Owner's Location, then adding ClacDrawOffset(), and then
//moving along the proper axis the amount determined by FireOffset specified in the Default
//Properties.

	Start = Owner.Location + CalcDrawOffset() + FireOffset.X * X + FireOffset.Y * Y + FireOffset.Z
* Z;

//AdjustedAim is essentially the direction that the projectile is to be fired.  It is calculated
//here by a function in Pawn called AdjustAim, that essentially just uses the Owner's ViewRotation
//and adjusts it towards any targets if AutoAim is enabled.

	AdjustedAim = pawn(owner).AdjustAim(ProjSpeed, Start, AimError, True, (3.5*FRand()-
1<PowerLevel));


//Here is where the actual sparkling projectile you see is created.  Depending on the Powerlevel,
//different projectiles (with different properties/meshes) are created.

	if ( (PowerLevel == 0) || (AmmoType.AmmoAmount < 10) )
		da = DispersionAmmo(Spawn(ProjClass,,, Start,AdjustedAim));
	else
	{
		if ( (PowerLevel==1) && AmmoType.UseAmmo(2) )
			da = Spawn(class'DAmmo2',,, Start,AdjustedAim);
		if ( (PowerLevel==2) && AmmoType.UseAmmo(4) )
			da = Spawn(class'DAmmo3',,, Start,AdjustedAim);
		if ( (PowerLevel==3) && AmmoType.UseAmmo(5) )
			da = Spawn(class'DAmmo4',,, Start ,AdjustedAim);
		if ( (PowerLevel>=4) && AmmoType.UseAmmo(6) )
			da = Spawn(class'DAmmo5',,, Start,AdjustedAim);
	}
	if ( da != None )
	{
		if (Mult>1.0) da.InitSplash(FMin(da.damage * Mult, 100));
	}
}

//Fire() and AltFire() are one of the many functions called when a Pawn decides it wants
//to fire the weapon its using.  This version of AltFire() just sets a boolean to true
```

```
//for AI purposes, calls CheckVisibility() which I'm not exactly sure does, and then goes
//to the AltFiring state.

function AltFire( float Value )
{
        bPointing=True;
        CheckVisibility();
        GoToState('AltFiring');
}

/////////////////////////////////////////////////////

//The AltFiring state in the Dispersion Pistol doesn't actually fire the weapon, but instead
//increments a value while the AltFire is pushed and once the button is released it changes
//states to ShootLoad which takes care of firing the projectile.

state AltFiring
{
ignores AltFire;

//The Tick() function is a way to make rapid updates/changes to a class.  It is called
//whenever possible (when enabled), and can be used for pretty much anything you can think
//of.  Be wary of putting too much code in the tick function, because it is called alot in
//a short amount of time, and can bring the game to a halt if it has to process too much.
//The Dispersion Pistol uses it to keep track of how long it has been charging, and limits
//how much you can charge.

        function Tick( float DeltaTime )
        {
                if ( Level.NetMode == NM_StandAlone )
                {
                        PlayerViewOffset.X = WeaponPos.X + FRand()*ChargeSize*7;
                        PlayerViewOffset.Y = WeaponPos.Y + FRand()*ChargeSize*7;
                        PlayerViewOffset.Z = WeaponPos.Z + FRand()*ChargeSize*7;
                }
                ChargeSize += DeltaTime;
                if( (pawn(Owner).bAltFire==0)) GoToState('ShootLoad');
                Count += DeltaTime;
                if (Count > 0.3)
                {
                        Count = 0.0;
                        If (!AmmoType.UseAmmo(1)) GoToState('ShootLoad');
                        AmmoType.GoToState('Idle2');
                }
        }

//EndState() and BeginState() are two function that can be used to do something at the end
//and beginning of a state respectively.  Dispersion Pistol uses BeginState() to initialize
//two variables, and uses EndState() to clean up.

        Function EndState()
        {
                PlayerviewOffset = WeaponPos;
                if (cl1!=None) cl1.Destroy();
                if (cl2!=None) cl2.Destroy();
        }
```

```
		function BeginState()
		{
				WeaponPos = PlayerviewOffset;
				ChargeSize=0.0;
		}

//The Begin: and End: labels are nearly identical to BeginState() and EndState(), the only
//main difference being that BeginState() and EndState() are functions, and if you need
//to use a local variable (to find some object for example) you can only create variables
//inside functions.  Otherwise Begin: and End: can be used for the same purposes.

Begin:
		if (AmmoType.UseAmmo(1))
		{
				Owner.Playsound(Misc1Sound,SLOT_Misc, Pawn(Owner).SoundDampening*4.0);
				Count = 0.0;
				Sleep(2.0 + 0.6 * PowerLevel);
				GoToState('ShootLoad');
		}
		else GotoState('Idle');

}

///////////////////////////////////////////////////////////

//ShootLoad is used to fire off a projectile when the Dispersion has either reached its max
//charge, or the Owner decided to let go.  Nothing really special in here, the BeginState()
//function is essentially ProjectileFire() except that it takes into account the ChargeSize
//variable, to increase damage and size appropriately.

state ShootLoad
{
		function Fire(float F) {}
		function AltFire(float F) {}

		function BeginState()
		{
				local DispersionAmmo d;
				local Vector Start, X,Y,Z;
				local float Mult;

				if (Amp!=None) Mult = Amp.UseCharge(ChargeSize*50+50);
				else Mult=1.0;

				Owner.MakeNoise(Pawn(Owner).SoundDampening);
				GetAxes(Pawn(owner).ViewRotation,X,Y,Z);
				InvCalcView();
				Start = Location + FireOffset.X * X + FireOffset.Y * Y + FireOffset.Z * Z;
				AdjustedAim = pawn(owner).AdjustAim(AltProjectileSpeed, Start, AimError, True,
True);
				d = DispersionAmmo(Spawn(AltProjectileClass,,, Start,AdjustedAim));
				d.bAltFire = True;
				d.DrawScale = 0.5 + ChargeSize*0.6;
				d.InitSplash(d.DrawScale * Mult * 1.1);
				Owner.PlaySound(AltFireSound, SLOT_Misc, 1.8*Pawn(Owner).SoundDampening);
```

```
                        if ( PlayerPawn(Owner) != None )
                                PlayerPawn(Owner).ShakeView(ShakeTime, ShakeMag*ChargeSize,
ShakeVert);
                        if (PowerLevel==0) PlayAnim('Shoot1',0.2, 0.05);
                        else if (PowerLevel==1) PlayAnim('Shoot2',0.2, 0.05);
                        else if (PowerLevel==2) PlayAnim('Shoot3',0.2, 0.05);
                        else if (PowerLevel==3) PlayAnim('Shoot4',0.2, 0.05);
                        else if (PowerLevel==4) PlayAnim('Shoot5',0.2, 0.05);
                }
Begin:
                FinishAnim();
                Finish();
}



///////////////////////////////////////////////////////////

//PlayIdleAnim() is an animation function that is called to play an idle animation, and
//because the Dispersion Pistol happens to have more than one idle animation, PlayIdleAnim()
//plays the proper animation depending on the PowerLevel.

function PlayIdleAnim()
{
        if (PowerLevel==0) LoopAnim('Idle1',0.04,0.2);
        else if (PowerLevel==1) LoopAnim('Idle2',0.04,0.2);
        else if (PowerLevel==2) LoopAnim('Idle3',0.04,0.2);
        else if (PowerLevel==3) LoopAnim('Idle4',0.04,0.2);
        else if (PowerLevel==4) LoopAnim('Idle5',0.04,0.2);
}

////////////////////////////////////////////////////////

//The PowerUp state is active when the weapon is powering up after picking up a weaponpowerup
// (tough one eh?), and can be used to play animations, sounds, modify variables or anything
//else.  The Dispersion Pistol needs this to play the proper animation (remember those cool
//things popping out of your gun?), and modify the ammo.

state PowerUp
{
ignores fire, altfire;

        function BringUp()
        {
                bWeaponUp = false;
                PlaySelect();
                GotoState('Powerup', 'Raising');
        }

        function bool PutDown()
        {
                bChangeWeapon = true;
                return True;
        }

        function BeginState()
```

```
		{
			bChangeWeapon = false;
		}

Raising:
		FinishAnim();
		PowerLevel++;
Begin:
		if (PowerLevel<5)
		{
			AmmoType.MaxAmmo += 10;
			AmmoType.AddAmmo(10);
			if ( PowerLevel < 5 )
				Owner.PlaySound(PowerUpSound, SLOT_None,
Pawn(Owner).SoundDampening);
			if (PowerLevel==1)
				PlayAnim('PowerUp1',0.1, 0.05);
			else if (PowerLevel==2)
				PlayAnim('PowerUp2',0.1, 0.05);
			else if (PowerLevel==3)
				PlayAnim('PowerUp3',0.1, 0.05);
			else if (PowerLevel==4)
				PlayAnim('PowerUp4',0.1, 0.05);
			FinishAnim();
			if ( bChangeWeapon )
				GotoState('DownWeapon');
			Finish();
		}
Waiting:
}

//All Tween* functions are animation related functions that are used to make the transition
//between different animations.  It is important to note that TweenAnim is not the same as
//PlayAnim.  You should use PlayAnim when there isn't currently an animation already playing,
//and use TweenAnim when one is playing, otherwise you could end up with noticeable "jumps"
//from one animation to the other.

function TweenDown()
{
		if ( GetAnimGroup(AnimSequence) == 'Select' )
			TweenAnim( AnimSequence, AnimFrame * 0.4 );
		else
		{
			if (PowerLevel==0) PlayAnim('Down1', 1.0, 0.05);
			else if (PowerLevel==1) PlayAnim('Down2', 1.0, 0.05);
			else if (PowerLevel==2) PlayAnim('Down3', 1.0, 0.05);
			else if (PowerLevel==3) PlayAnim('Down4', 1.0, 0.05);
			else if (PowerLevel==4) PlayAnim('Down5', 1.0, 0.05);
		}
}

function TweenSelect()
{
		TweenAnim('Select1',0.001);
}
```

```
function PlaySelect()
{
        Owner.PlaySound(SelectSound, SLOT_None, Pawn(Owner).SoundDampening);
        if (PowerLevel==0) PlayAnim('Select1',0.5,0.0);
        else if (PowerLevel==1) PlayAnim('Select2',0.5,0.0);
        else if (PowerLevel==2) PlayAnim('Select3',0.5,0.0);
        else if (PowerLevel==3) PlayAnim('Select4',0.5,0.0);
        else if (PowerLevel==4) PlayAnim('Select5',0.5,0.0);
}
```

Melee Weapons

Melee weapons are really very simple, once you understand the math behind finding a target that is in front of the player, the rest is just like a normal weapon.
This tutorial assumes that you have a model for a melee weapon and are familiar with the animation functions necessary. This will not cut and paste very well, and it will look awkward if you hit somebody with your rifle... :)

In order to create a weapon that will attack in closerange we need to be able to get a target that is nearby, make sure that the target is in front of the player, and then finally damage the target. I have gone ahead and stored all this into a neat MeleeFire function to keep the new code as clean as possible.

```
//=============================================================

// We need to override our existing Fire function to call
// our new MeleeFire function.

function Fire(float Value)
{
   MeleeFire(100);
}

function MeleeFire(int range)
{
   local vector x, y, z, hitloc, momentum;
   local actor Other;


   // First use the GetAxes function to get three vectors which
   // are equal to the X, Y, and Z axis, relative to the player.
   // We need to forward vector for the check to see if the
   // target is in front.

   GetAxes(PlayerPawn(Owner).ViewRotation, x, y, z);


   // Use the foreach iterator function to grab a list of actors
   // within the specified range.

   foreach RadiusActors(class'Actor', Other, range, Owner.Location)
   {

         // Other.Location - Owner.Location gives us a vector that
         // points to the target (Other) from the Owner.Location.
```

```
        // When we get the dot product of this vector and the forward
        // vector we get a value that if positive means that the
        // target is in front of us, and negative would indicate that it
        // is behind us.

        if ((Other.Location - Owner.Location) Dot X > 0 && Other.IsA('Pawn'))
        {

                // Now that we know that the target is in front go ahead
                // calculate the hitlocation using the targets CollisionRadius
                // NOTE: This isn't really necessary, but it makes more sense
                // when applying the damage.

                hitloc = Other.Location + (Other.Location - Owner.Location) * Other.CollisionRadius;

                // Go ahead and apply the damage.

                Other.TakeDamage(Damage, Pawn(Owner), hitloc, 3000.0 * x, 'hit');
        }


        // Also add a little kick so that the target will get knocked back
        // a bit realistically

        momentum = (Other.Location - Owner.Location) * 1000/Other.Mass;
        Other.Velocity += momentum;
    }
}
```

Bullet Holes

```
/*
Notes: Here is some cheese code i did as temp for a project i'm currently apart of. Use it as you like, however i'd be pleased to know what you use it for. its fairly generalized and you can  use it for bullet holes, scorch marks, blood splatters ect..ect.. its a modified processtrace hit, so its called by TraceHit(), now it doesn't have to be used by trace hit, you could copy it into a projectiles hitwall, but i'd suggest if your using a projectile to still call traceHit() and leave the code here as is, for better reliability.
*/

function ProcessTraceHit(Actor Other, Vector HitLocation, Vector HitNormal, Vector X, Vector Y, Vector Z)
{
  local actor BH;   //This is a container for the Bullet Hole effect
  local Float FloorNormal;   //This var will be used later to determine the floor

  FloorNormal = HitNormal Dot vect(0,0,1);   //This dot product returns a float number
  //1.0 is the ceiling, 1.0 is the floor, 0.0 is a wall, this of course assumes the surfaces are flat
  if ((!Other.Isa('Pawn')) && (!Other.IsA('Carcass')))  /* you don't want it to appear on people, or their dead bodies =) */
  {
    if (FloorNormal < 1.00) /* and it looks like crap on the floor if the texture is like grass or somthing so don't draw it */
    {
     BH = Spawn(HitEffect,,,HitLocation); // - Do the Deed =P - //
     // -  HitEffect is declared elsewhere as var() Class HitEffect; - //
```

```
      BH.SetBase(Other); // - Make it stick to the obj - //
      BH.DrawScale /= 10; // - the gfx used was to big, so i  shrank it - //
    }
  }
}
```

TeleFRAG

TeleFRAG - This mod is an ASMD that teleports you wherever you want to go whenever you press alt-fire. The concept behind the gun is the TeleFrag principle, you kill whatever you teleport on... ;)

```
//=============================================================================
// The gun itself... Not much changed, just the pick-up message,
// and what the gun shoots out.
//=============================================================================

class TeleFRAG expands ASMD;

// PreBeginPlay is called right when the level starts, before the player does
// anything.  Here, this basically modifies what Projectile is fired out

function PreBeginPlay()
{
		Super.PreBeginPlay();

// TeleProj is the Projectile

		AltProjectileClass = class'TeleProj';

// This sets the Projectile speed to the default
		AltProjectileSpeed = class'TeleProj'.Default.speed;
}


defaultproperties
{
    PickupMessage="You got the TeleFragger!"
}


//=============================================================================
// TeleProj - The projectile fired.
//=============================================================================

class TeleProj expands TazerProj;

// SuperExplosion is what happens when you Alt-Fire and then hit
// that ball with your primary-fire.  It just makes a bigger
// explosion

function SuperExplosion()
{

		// This next line will hurt everyone within 240 units by 3.9x the regular damage
```

```
      HurtRadius(Damage*3.9, 240, 'jolted', MomentumTransfer*2, Location );

              // Makes an explosion

              Spawn(Class'RingExplosion2',,'',Location, Instigator.ViewRotation);

              // Make teleport sparkles where the guy who fired is (The Instigator)

              Spawn ( class 'PawnTeleportEffect',,, Instigator.Location );

              // Make teleport sparkles where you will end up

              Spawn ( class 'PawnTeleportEffect',,, Location );


              // Here, you are transported to wherever you hit

              Instigator.SetLocation(Location);

              // This destroys the projectile, 'cause it exploded!

              Destroy();
}


// Explosion is what happened when the ball hit something or other

function Explode(vector HitLocation,vector HitNormal)
{

              // Play the Explosion sound

              PlaySound(ImpactSound, SLOT_Misc, 0.5,,, 0.5+FRand());

// Hurt everyone 70 units away

              HurtRadius(Damage, 70, 'jolted', MomentumTransfer, Location );

// If the damage is high, make a big explosion

              if (Damage > 60) Spawn(class'RingExplosion3',,, HitLocation+HitNormal*8,rotator(HitNormal));

// If not, make a smaller explosion

              else Spawn(class'RingExplosion',,, HitLocation+HitNormal*8,rotator(HitNormal));

// Make teleport sparkles where the guy who fired is (The Instigator)

              Spawn ( class 'PawnTeleportEffect',,, Instigator.Location );

// Make teleport sparkles where you will end up

              Spawn ( class 'PawnTeleportEffect',,, Location );

// Here, you are transported to wherever you hit
```

```
		Instigator.SetLocation(Location);

// This destroys the projectile, 'cause it exploded!

		Destroy();
}

defaultproperties
{
}
```

Modifying the HUD Pt. 1

This is Part 1 (of 2) that goes into detail and explains various parts of the HUD. Very useful for adding your own look/feel to your modification, and it allows you to possibly integrates all kinds of cool new features like motion detectors, compasses, etc...

ModifyingtheHud Part 1.
**This is not a mod or a modification, instead this tries to provide information and give examples of where the code needs to be modified, I don't know about you, but I find that I learn little to nothing from just reading someone else's code.

Preperation: (if you're pretty adept on creating mods proceed to **)
* Open up UnrealED, on the right hand side drop down menu called browse click on classes, then on the new list click on HUD, then on UnrealHUD, the best thing to start out doing is creating a new class under UnrealHUD so right click on it and click on...
** Create a new class under UnrealHUD. Then choose the project name you want.

Note:
For the purpose of this tutorial I suggest copying at least the function PostRender from the UnrealHUD class, this is basically where all of the work is done. I will try to comment all the functions from the hud and tell you where to turn to modify things.

Things to know before hand -- The canvas:
The biggest thing you need to be aware of is the Canvas class, the canvas is an Unreal graphics window, and for the hud you need to draw directly on to it. As far as I know there can only be one Canvas in use at a time, and the engine is what controls the canvas and the calling of these functions. As far as variables go, the biggest thing you need to know are ClipX and Y and CurX and Y. The clip is the bottom right of the screen and the CurX and CurY are the current position for drawing. You can take a look at the Canvas code, the biggest things you need to be aware of are the functions DrawIcon and DrawText. This is what you will utilize to get things drawn on the screen. DrawIcon takes two parameters, the texture name and the draw scale. DrawText, takes in the text you want drawn.

```
//=============================================================================
class ModifyingtheHud expands UnrealHUD;

// Function HUDSetup, sets up the defaults for the hud, including
// the font, you can also set any of the other Canvas defaults from
// here.
 simulated function HUDSetup(canvas canvas)
 {
 // Setup the way we want to draw all HUD elements
  Canvas.Reset();
  Canvas.SpaceX=0;
  Canvas.bNoSmooth = True;
  Canvas.DrawColor.r = 255;
  Canvas.DrawColor.g = 255;
  Canvas.DrawColor.b = 255;
 // set the font
  Canvas.Font = Canvas.LargeFont;
 }


// Function PostRender, does the actual rendering of the hud on the
// screen. For every new icon or text you want drawn on the screen, you
// need to tell post render to either draw it or call another function
// that will do the work. PostRender is called every tick. */

simulated function PostRender( canvas Canvas )
{
		//sets the defaults up for the canvas
		HUDSetup(canvas);

		//if the owner of this hud is a player, then...
		if ( PlayerPawn(Owner) != None )
		{
		//if you have pressed escape then show the menu, I'll make a menu
		//tutorial soon.
			if ( PlayerPawn(Owner).bShowMenu )
			{
				if ( MainMenu == None )
					CreateMenu();
				if ( MainMenu != None )
					MainMenu.DrawMenu(Canvas);
				return;
			}
			//self explanatory show the score
			if ( PlayerPawn(Owner).bShowScores )
			{
				if ((PlayerPawn(Owner).Scoring == None) &&
(PlayerPawn(Owner).ScoringType != None) )
					PlayerPawn(Owner).Scoring =
Spawn(PlayerPawn(Owner).ScoringType, PlayerPawn(Owner));
				if ( PlayerPawn(Owner).Scoring != None )
				{
					PlayerPawn(Owner).Scoring.ShowScores(Canvas);
					return;
				}
			}
			//draw the crosshair if the player has a weapon
			else if ( (PlayerPawn(Owner).Weapon != None) && (Level.LevelAction ==
LEVACT_None) )
				DrawCrossHair(Canvas, 0.5 * Canvas.ClipX - 8, 0.5 * Canvas.ClipY - 8);
			if ( PlayerPawn(Owner).ProgressTimeOut > Level.TimeSeconds )
				DisplayProgressMessage(Canvas);

		}

// here's where the fun really begins :).  The rest of the code basically says depending on the hud mode then
//draw this (armor, ammo, health) on the screen at different positions, base it off of the Clip to make sure it
```

```
//looks right at no matter what the resolution of the screen. To totally remake the HUD then erase all this and
//use the code here as an example, I'll show you how to do the drawing to the screen in a couple of
//paragraphs.
		if (HudMode==5)
		{
			DrawInventory(Canvas, Canvas.ClipX-96, 0,False);
			Return;
		}
		if (Canvas.ClipX<320) HudMode = 4;

		// Draw Armor
		if (HudMode<2) DrawArmor(Canvas, 0, 0,False);
		else if (HudMode==3 || HudMode==2) DrawArmor(Canvas, 0, Canvas.ClipY-32,False);
		else if (HudMode==4) DrawArmor(Canvas, Canvas.ClipX-64, Canvas.ClipY-64,True);
		if (HudMode!=4) DrawAmmo(Canvas, Canvas.ClipX-48-64, Canvas.ClipY-32);
		else DrawAmmo(Canvas, Canvas.ClipX-48, Canvas.ClipY-32);
		//		Draw Health
		if (HudMode<2) DrawHealth(Canvas, 0, Canvas.ClipY-32);
		else if (HudMode==3||HudMode==2) DrawHealth(Canvas, Canvas.ClipX-128, Canvas.ClipY-32);
		else if (HudMode==4) DrawHealth(Canvas, Canvas.ClipX-64, Canvas.ClipY-32);

		//Display Inventory
		if (HudMode<2) DrawInventory(Canvas, Canvas.ClipX-96, 0,False);
		else if (HudMode==3) DrawInventory(Canvas, Canvas.ClipX-96, Canvas.ClipY-64,False);
		else if (HudMode==4) DrawInventory(Canvas, Canvas.ClipX-64, Canvas.ClipY-64,True);
		else if (HudMode==2) DrawInventory(Canvas, Canvas.ClipX/2-64, Canvas.ClipY-32,False);

		//Display Frag count
		if ( (Level.Game == None) || Level.Game.IsA('DeathMatchGame') )
		{
			if (HudMode<3) DrawFragCount(Canvas, Canvas.ClipX-32,Canvas.ClipY-64);
			else if (HudMode==3) DrawFragCount(Canvas, 0,Canvas.ClipY-64);
			else if (HudMode==4) DrawFragCount(Canvas, 0,Canvas.ClipY-32);
		}
}


//as an example lets look at DrawHealth, you can use this as a base for one of your functions.
simulated function DrawHealth(Canvas Canvas, int X, int Y)
{
		 //set the current canvas position to what you had wanted in
	//PostRender
		Canvas.CurY = Y;
		Canvas.CurX = X;
		 //set the Font to large
		Canvas.Font = Canvas.LargeFont;
		 //if your health is less then 25 then use the red font
		if (Pawn(Owner).Health<25) Canvas.Font = Font'LargeRedFont';
		 //call drawicon to draw the health icon and use the drawscale of
		 //one
		Canvas.DrawIcon(Texture'IconHealth', 1.0);
		 //then set the cursor to draw the health ammount
		Canvas.CurY += 29;
		 //draw icon value draws the number inside the icon
		DrawIconValue(Canvas, Max(0,Pawn(Owner).Health));
		Canvas.CurY -= 29;
```

```
                if (HudMode==0) Canvas.DrawText(Max(0,Pawn(Owner).Health),False);
                 //ever wonder how they did that health bar effect?  pretty easy,
        //draw the bar as a tile, and tile in respect to the health
        //that's left
                Canvas.CurY = Y+29;
                Canvas.CurX = X+2;
                if (HudMode!=1 && HudMode!=2 && HudMode!=4)

                Canvas.DrawTile(Texture'HudLine',FMin(27.0*(float(Pawn(Owner).Health)/float(Pawn(Owner).
Default.Health)),27),2.0,0,0,32.0,2.0);
   }

//Now, you've designed your wonderful HUD, how do you get it to show
//up in your level?  Well, I'll show you that in Part 2
```

Modifyingthe Hud Part 2.

This part is relatively simple, create a new class under UnrealGameInfo. Then right-click on your new class and choose edit default properties, then click on GameInfo, then where it says HUDType type in the name of your project .u file followed by a dot and the name of your hudclass. Then when you create a map, click on options up on top and select level properties then click on LevelInfo, and where it says DefaultGameInfo choose this game info class. Save the level, and you are all set :P

```
class ModifyingtheHud2 expands UnrealGameInfo;

defaultproperties
{
}
```

Capture the Flag

This tutorial covers 3 basic classes used to make a Capture the Flag game, CTFGame, FlagSpawnPoint, and Flag. CTFGame is a GameInfo that handles the startup of games and is a subclass of TeamGame. FlagSpawnPoint is a NavigationPoint that handles the creation of the Flags in a level. Flag is a Pickup that is the flag(s), and it takes care of checking for captures ...etc.
This tutorial covers how to make a *basic* Capture the Flag game with very basic scoring and other rules. Feel free to use this as a base to add CTF support to your modification, and if you make money off of it, at least think kindly of me if you're too stingy to send me the money. :)

```
//=============================================================================
// CTFGame - Capture the Flag GameInfo
//
// Since Unreal already has a Team game already setup, and CTF requires teams,
// we only need to subclass the TeamGame class in order to get team support in.
// In this class we need to make sure the flags get spawned, and handle scoring
// for teams/individuals.
//=============================================================================

class CTFGame expands TeamGame;


// The number of points to award to the player and team upon a successful capture

var() int CaptureScore;
```

```
// In PostBeginPlay() we search for all FlagSpawnPoints and tell them to spawn a
// flag.  NOTE: This requires that maps need to have FlagSpawnPoints already
// placed in them in order to properly spawn flags (although you could easily
// write some code to play FlagSpawnPoints if none are present).

function PostBeginPlay()
{
        local FlagSpawnPoint fs;

        foreach AllActors(class'FlagSpawnPoint', fs)
        {
                fs.SpawnFlag();
        }
}


// ScoreTeamCapture does exactly what the name implies, it scores points for a
// capture.

function ScoreTeamCapture(byte Team, PlayerPawn Scorer)
{

        // TeamInfo is a class (under the Info class) that is used to store
        // pertinent information about different teams in a Team game.  We
        // need to use it to add points to the entire team, as well as the
        // individual capturer.

        local TeamInfo ti;


        // Find a matching TeamInfo based on Team and then give them some
        // points.

        foreach AllActors(class'TeamInfo', ti)
                if (ti.TeamIndex == Team)
                {
                        ti.Score += CaptureScore;
                        break;
                }


        // And finally award the person who did the actual capture.

        Scorer.PlayerReplicationInfo.Score += CaptureScore;
}


// DiscardInventory is called whenever a Pawn dies, and since we don't
// want a flag destroyed if a player dies we must override this to
// put it back to its original position.  NOTE: It would be a good idea
// to expand this to drop the flag wherever they died and only send
// it back after a certain time or if it is touched by a player of the
// same team. (as in Quake2 CTF and others)
```

```
function DiscardInventory(Pawn Other)
{
		local Inventory Inv;


		// Use FindInvetoryType to get a reference to the flag if the pawn
		// has it, and if so drop it.

		Inv = Other.FindInventoryType(class'ctf_ca.Flag');
		if (Flag(Inv) != None)
		{

				// DropFrom drops the Inventory item at the specified
				// location and handles removing the item from the player's
				// inventory.

				Inv.DropFrom(Flag(Inv).InitLocation);
		}


		// Let the normal DiscardInventory do its work now that we've done
		// what we needed.

		Super.DiscardInventory(Other);
}

defaultproperties
{
    CaptureScore=10
    bSpawnInTeamArea=True
}


//=============================================================================
// FlagSpawnPoint - Flag Spawning Point
//
// This class should be placed where you want the flags to be in
// the level.  The SpawnFlag() function is called at the beginning
// of the game and creates the actual flag.  NOTE: Since FlagSpawnPoint
// is a subclass of NavigationPoint it will be part of the pathnode
// table of a level, and as such will be usuable by bots, so it should
// be relatively simple to alter Unreal's bots to play CTF.
//=============================================================================

class FlagSpawnPoint expands NavigationPoint;


// TeamNumber should be set to the owner Team

var() byte TeamNumber;

// MyFlag is a local reference of the flag spawned

var Flag MyFlag;
```

```
// SpawnFlag() just creates the flag, then sets the Team and InitLocation
// variables.

function SpawnFlag()
{
        MyFlag = spawn(class'ctf_ca.Flag');
        MyFlag.Team = TeamNumber;
        MyFlag.InitLocation = Location;
}

defaultproperties
{
}


//=============================================================================
// Flag - CTF Flag
//
// This is a normal Pickup item, and we alter the Pickup state (the Touch()
// function in particular) to check for flag captures...etc.  It is created
// by a FlagSpawnPoint, and it handles the majority of the CTF game code itself.
//=============================================================================

class Flag expands Pickup;


// Team and Initial location

var byte Team;
var vector InitLocation;

auto state Pickup
{

        // Touch() is called whenever an Actor collides with this Actor, and
        // normally for a Pickup item it would handle adding this item to the
        // inventory of the Pawn if possible.  We alter it to see if the
        // PlayerPawn touching us has a flag (for a capture), or if the PlayerPawn
        // is on an opposite team we give the flag to them like a normal
        // item.

        function Touch( actor Other )
        {
                local Inventory Copy, EnemyFlag;


                // If it isn't a playerpawn then can't grab the flag.
                // NOTE: Might be a good idea to change this to Pawn, or add
                // a check for Bots if you want to add Bot support.

                if (!Other.IsA('PlayerPawn'))
                        return;


                // If it is on the same team as us then go ahead and check for a
                // capture.
```

```
				if (PlayerPawn(Other).PlayerReplicationInfo.Team == Team)
				{

						// Check for capture by seeing if they have a flag.

						EnemyFlag = Pawn(Other).FindInventoryType(class'ctf_ca.Flag');
						if (EnemyFlag != None)
						{

										// Score points, reset flag, and let the world know
about it.


		BroadcastMessage(PlayerPawn(Other).PlayerReplicationInfo.PlayerName $ " captured the enemy
flag!");
										CTFGame(Level.Game).ScoreTeamCapture(Team,
PlayerPawn(Other));

		Flag(EnemyFlag).DropFrom(Flag(EnemyFlag).InitLocation);
						}
						return;
				}

				// If they aren't on the same team then give the flag to them.

				else
				{

						// Give flag to enemy and let the world know about it.

						BroadcastMessage(PlayerPawn(Other).PlayerReplicationInfo.PlayerName
$ " stole the enemy flag!");
						bHeldItem = true;
						GiveTo(Pawn(Other));
						Pawn(Other).ClientMessage(PickupMessage, 'Pickup');   // add to inventory
and run pickupfunction
						PlaySound (PickupSound,,2.0);
				}
		}
}

defaultproperties
{
   PickupMessage="Got the flag!"
   ItemName="Flag"
   PickupViewMesh=Mesh'UnrealI.Flag1M'
}
```

Unreal Rain

The third tutorial from c0mpi1e which have turned out to be an excellent, original series of total conversion snippets that will appear in the final game. This tutorial is no different, and is probably the most atmospheric-based mod I have seen so far.

```
// RainDrops.
//=============================================================================

class RainDrops expands Effects;
var effects d;

function PreBeginPlay()
{
		Super.PostBeginPlay();

		//make the rain actually fall :)
		SetPhysics(PHYS_Falling);
}


//=============================================================================
//when a drop hits a wall or lands on the ground then create
//a small rain puddle effect

function HitWall( vector HitNormal, actor HitWall )
{
		Super.HitWall(HitNormal, HitWall);

		//60% chance that there will be a puddle left on the ground
		//do this to keep performance up
		if (Frand() < 0.8)
		{
				spawn(class'RainPuddle',,,Location);
		}
		Destroy();
}

//same for Landed
function Landed(vector HitNormal)
{
		Super.Landed(HitNormal);
		if (Frand() < 0.8)
		{
				spawn(class'RainPuddle',,,Location);
		}
		Destroy();
}


//=============================================================================
// RainPuddle.
// most of the code here is copied from the RingExplosion code, basically
// I just changed the DrawScale to a smaller value, I do not take
// any credit for this outside of that, its makes a very nice effect.
//=============================================================================

class RainPuddle expands Effects;

var bool bSpawnOnce;

simulated function Timer()
```

```
{
        local WaterRing r;

        if ( Level.NetMode != NM_DedicatedServer )
        {
                r = Spawn(class'WaterRing',,,,rot(16384,0,0));
                r.DrawScale = 0.02;
                r.RemoteRole = ROLE_None;
        }
        else
                Destroy();
        if (bSpawnOnce) Destroy();
        bSpawnOnce=True;
}


simulated function PostBeginPlay()
{
        SetTimer(0.3,True);
}

//=============================================================================
// RainGen.
// author: c0mpi1e(Carlos Cuello)
// Total Time So Far: 20 hrs.
//
// legal: Everything in this class is 100% original and is the property of me.  It was originally
// produced for the USCM:Infestation TC, and any revisions to this mod will be used in that.  This is the final
// version that I will release to the public.  You may use this in any map and release it as part of any map you
//  create as long as you give me full credit.  If you need anything added in particular you can email me and I'll
// see what I can do. This is basically what controls the Rain and is the only thing you
// will have to work with.  I wanted to keep this something that anyone can use, tried to keep it very
// configurable and tried to make everything be handled from UnrealEd.  So for all you map makers,
// simply add as many raingen's to the ceiling of the sky where you want it to fall from.  The RainGen delivers
// a limited range of raindrops, so do not expect one raingen to create a monsoon.  To change the various
// settings for the rain, select all the RainGens in your map and click on the properties dialog box, then click
// on the several Rain propety menu's that appear(Rain_Behaviour, Rain_Looks, Rain_Sounds).  The
// intensity variable should be kept between 0 and 5 or so to keep  rendering speed good.  DropSpeed should
//  be negative for faster rain, and positive for slower rain, however beware that a positive integer
// greater than 100 or so will cause the raindrops to go up and hit the ceiling.
// note: for this release, lightning is not working, if you absolutley need it, then email me and I'll see what I
// can do.
//=============================================================================

class RainGen expands Effects;

//Variables you can manipulate from UnrealED===================================
//this enum lets a mapper or whatnot choose between 3 different sprites
//for the rain drop.  The parenthesis after var tells unreal that this
//variable can be changed from Unrealed, if you put something in the
//parenthesis then that will be the parent menu of the variable.

var(Rain_Looks) enum ERainType
{
        Rain_Drop,
        Rain_Line,
          Rain_LongLine
} RainDrawType;
var(Rain_Looks) float DropSize;               //the size of each drop
var(Rain_Behaviour) int intensity;            //the intensity of the rain
var(Rain_Behaviour) int DropSpeed;            //determines the drop speed. Neg numbers
                                                        //make it drop faster, positive slows it
                                                        //down, should be modified for the
                                                        //default properties dialog

var(Rain_Behaviour) int RainRadius;           //The radius of the rain, which is
                                                        //random from 0 to RainRadius
var(Rain_Behaviour) bool bThunder;
var(Rain_Behaviour) bool bLightning;          //not implemented yet
var(Rain_Sounds)       Sound ThunderSound1;
var(Rain_Sounds)       Sound ThunderSound2;
var(Rain_Sounds)    int SoundRadius;
var(Rain_Sounds)       int RainVolume;

//===============================================================================
//Private Variables we want to keep to ourselves===============================

var            int ct;                        //misc counter variable, used for
                                                        //for loops, rather have it as a global
                                                        //variable because we might use it more
                                                        //than once.
var   Effects D;                              //the actual raindrop
var            Effects   L;                             //used for lightining
var   vector NewLoc;                          //the location of the random rain drop

//===============================================================================

//set all our variables.

simulated function PreBeginPlay()
{
          Super.PreBeginPlay();

          //we want to enable tick and timer
          //Enable('Tick');

          Enable('Timer');

          //set the timer for every ten clicks we want the thunder to be
          //handled here, randomly.  If you put this into the tick function
          //it slows things down considerably.

          SetTimer(10, true);

          //play the rain sound, using the Ambient slot

          PlaySound(EffectSound1, Slot_Ambient, RainVolume,, SoundRadius);
}


//find a random number from -Max to Max
```

```
function int RandomNeg(int Max)
{
   local int k;
          k = Rand(Max + 1);
          if (Rand(2) == 1)
                    k = -k;
          return k;

}


//this is where thunder and lightning are handled

function Timer()
{
          Super.Timer();
          if (bThunder == true && Rand(5) < 3)
          {
//randomly choose which thunder sound to use
                    if (Frand() < 0.5)
                              PlaySound(ThunderSound1, Slot_Misc, 25, true);
                    else
                              PlaySound(ThunderSound2, Slot_Misc, 25, true);
          }

          if (bThunder == false)

                    //if we go into the timer, but have bthunder set to false
                    //then we no longer want to call the timer, we can put this before
                    //in PreBeginPlay, but because we use default properties
                    //to handle everything, we will not know then.
                    Disable('Timer');
}


//=====================================================================
//this is where everything takes place

function Tick(float deltatime)
{


          //for the number of intensity that is set from UnrealEd, then
          //spawn a new raindrop

          for (ct = 0; ct < intensity; ct ++)
          {

                    //find a new location for the spawned RainDrop, using my
                    //function RandomNeg() and base it off of the current Location
                    //we only want to produce one on the horizontal plane of
                    //sky or ceiling, so we leave the Z axes alone.

                    NewLoc.X = Location.X + RandomNeg(RainRadius);
                    NewLoc.Y = Location.Y + RandomNeg(RainRadius);
                    NewLoc.Z = Location.Z;
			//spawn a raindrop at NewLoc
			d = Spawn(class'RainDrops',,,NewLoc);

			//set the size to what the user specified
			d.DrawScale = DropSize;

			//same with the speeed
			d.Velocity.Z = DropSpeed;

			//Depending on what you choose for RainDrawType then set
			//the appropriate Texture.

			switch RainDrawType
			{
				case Rain_Drop:
					d.Texture = Texture'RainDrop4';
					break;
				case Rain_Line:
					d.Texture = Texture'RainLong';
					break;
				case Rain_LongLine:
					d.Texture = Texture'RainLonger';
					break;
				default:
					d.Texture = Texture'RainDrop4';

			}
		}
}
```

Snow

```
//=============================================================================

class SnowFlake expands Effects;

//=============================================================================
// Ok, Prebeginplay does all the bracketed functions before the item is created,
// or "Enters play",  Super calls the parent function of the function after the
// . so in this case ir calls Effects PostBeginPlay and that is the opisite of
// PreBeginPlay
//=============================================================================

function PrebeginPlay()
{
 Super.PostBeginPlay();
 setPhysics(Phys_Falling);
}

//=============================================================================
// hitwall tests to see if said object hits a wall and in this case it destroys
// which can be thought of as either melting it or it combining with the snow on
// the ground itself. It would bog things down to much to allow the sprites to
// build up.
//==========================================================================

function hitwall(vector hitnormal, actor hitwall)
{
 Super.hitwall(hitnormal, hitwall);
 destroy();
}

//==========================================================================
// Landed Is not Unlike Hitwall
//==========================================================================

function landed(vector HitNormal)
{
 Super.landed(HitNormal);
 destroy();
}

defaultproperties
{
    DrawType=DT_Sprite
    Sprite=Texture'y2k.snow_flake'
    Texture=Texture'y2k.snow_flake'
}

//==========================================================================
// SnowGen.
//==========================================================================

class SnowGen expands Effects;


//==========================================================================
// This are vars setup to be used later, puting them up here basicly makes them
// global.
//==========================================================================

var(Flake_appearance) float SFSize;
// [ Snow flake size ] //

var(Flake_Behaviour) int intensity;
// [ how rapid they fall ] //

var(Flake_Behaviour) int FlakeSpeed;
// [ Thate rate of speed they fall ] //

var(Flake_Behaviour) int SnowRadius;
// [ The field they fall in ] //

var effects SF;
// [ The Snow Flake ] //

var vector NewLocal;
// [ A dummy var to be used for location ] //
```

```
//Borrowed from c0mpi1es code, roughly//

function int RandNeg(int max)
{
 local int num;
  num = rand(max +1);
  if (rand(2) == 1)
   {
    num -= num;
   }
  return num;
}

//===========================================================================
// Tick is the smallest unit of time measurement in the game, and i think its
// derived from cpu ticks so its realitive to your machine, and so if thats
// the case it updates every cpu tick. The lines With NewLocal are used to
// generate a random patern in a given radius, so the flakes don't all fall
// In the same spot. Spawn creates the object, DrawScale is how big to make
// it, and velocity is kinda self explanitory.
//===========================================================================

function Tick(float deltatime)
{
 local int i;

  for (i = 0; i < intensity; i++)
   {
    NewLocal.x = location.x + RandNeg(SnowRadius);
    Newlocal.y = location.y + RandNeg(SnowRadius);
    Newlocal.z = location.z;

    SF = Spawn(class'SnowFlake',,,Newlocal);
    SF.DrawScale = SFSize;
    SF.Velocity.z = FlakeSpeed;

   }
}

defaultproperties
{
}
```