
UnrealScript Language Reference

Tim Sweeney
Epic MegaGames?, Inc.
tim@epicgames.com
http://www.epicgames.com/

Last Updated: 12/21/98

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)

Variables

Simple Variables

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 );
}
```

Functions

Declaring Functions

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.

Function parameter specifiers

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 );
}
```

Do-While Loops

"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
```

While Loops

"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" );
}
```

Case Statements

"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
{
        //...
}
```

State Labels and Latent Functions

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.

UnrealEd? Command Interface

About #exec commands

...

Supported #exec commands

...

End