COMPUTER SCIENCE

DEPARTMENT

FACIN/PUCRS – Brazil

http://www.inf.pucrs.br

UnrealScript Language Syntax

Leonardo Sewald Cunha and Lucia Maria Martins Giraffa

TECHNICAL REPORT SERIES

Number 027

July, 2002

Contact:
sewald@inf.pucrs.br
http://www.inf.pucrs.br/~sewald

giraffa@inf.pucrs.br
http://www.inf.pucrs.br/~giraffa

Leonardo Sewald Cunha is a graduate student of PUCRS – PUC/RS – Brazil. He is a member of the GAMES AND AI research group since 2001. He receives a graduate research grant from Dell Computers to support his research.

Lucia Maria Martins Giraffa works at PUCRS/Brazil since 1986. She is a titular professor and leader of the GIE group. She develops research in Multi-agent systems, Artificial Intelligence applied in Education and design of educational software. She got her Ph.D. in 1999 at UFRGS (Brazil).

Published by the Campus Global – FACIN – PUCRS
Av. Ipiranga, 6681
90619-900 Porto Alegre – RS – Brazil

UNREALSCRIPT LANGUAGE SYNTAX

Technical Report Nº 027/2002
Leonardo Sewald Cunha (masters course student)[1]
Lucia Maria Martins Giraffa[2]

1. Introduction

Real-time interactive computer games have been showing many innovations concerning about performance, interfaces and project techniques. According to Laird [LAI01], real-time interactive computer games offer robust environments for researchers to test and develop AI techniques, aside of the fact that games are relatively cheap and accessible, comparing to industrial or commercial applications. These games have environments populated with human and/or computer-controlled characters. Since a direct addressing between interactive game characters and agents can be done, as stated by Laird and others in Cunha [CUN01], these games become a rich laboratory for Artificial Intelligence (AI) research, especially in what concerns agent development.

Moreover, interactive computer games avoid many of the criticisms leveled against research based on simulations: because researchers do not develop them, the games avoid embodying preconceived notions about which aspects of the world designers can simulate with ease and which aspects are more difficult to simulate. These games constitute real products that create real environments with which millions of humans can interact vigorously. For these attributes, the use of real-time interactive computer game environments in intelligent agent research is an interesting field to experiment and explore.

Having this thought in mind, the GAMES AND AI research group at PPGCC/PUCRS is developing SCORE (Simulator for COgnitive agent's behavioR), a research project which aims to integrate an agent specification formalism, programming language and tool called X-BDI with the game *Unreal Tournament* (UT). Our project intends to make possible to model and program high-level behavior for BDI agents and visualize the results in an interactive game environment.

[1] e-mail: sewald@inf.pucrs.br

[2] e-mail: giraffa@inf.pucrs.br

UT was created by the company *Epic Megagames* and released in 1999. The game belongs to the *first-person shooter* (FPS) game category, where the player visualizes the environment from the game character's perspective. UT was the second game title released by *Epic Megagames* based on *Unreal's* architecture, the first one being *Unreal*. UT has state-of-the-art three-dimensional (3D) graphics, many multiplayer features, allowing players to compete against each other or play cooperatively over a network and a fairly complex Artificial Intelligence (AI) implementation. But more importantly, it was developed with the use of an architecture which allows third-party developers, users and researchers to easily expand, add or modify game content, either through an editing tool (provided with the game) or through the game's built-in script language, called *UnrealScript* (US).

The goal of this Technical Report (TR) is to gather and organize information about the US language, used as a testbed for our reserarch project related to the Master's Dissertation of the first author. This TR is intended to be a quick reference for people familiar with Java/C++ object-oriented programming approach. In particular, it is directed at those using US for academic purposes.

This TR was written with a heavy basis on two documents: *The UnrealScript Reference* [SWE01], created by the *Unreal* development team, and on *The UnrealScript Syntax* [MAR02], an online US reference created by Robert J. Martin. For updates and more information about US, it is recommended to read those documents.

This text is divided into six sections. Section 2 presents a brief description on UT and it's associated script language, US. Section 3 describes the US' basic syntax. Section 4 covers advanced aspects of US' syntax. Section 5 covers aspects related to debugging US code. References appear at the end of the volume (Section 6).

2. UnrealScript Language

UT (as some other modern computer games) has it's *engine*[3] implemented over the form of *Dynamic Link Libraries* (DLL's). They are files which store the system's object/function libraries. These DLL's are used to manipulate game data (textures,

[3] In what concerns computer game development, an *engine* consists of a set of methods/functions and data structures created to ease the manipulation of game data, thus assisting programmers in creating computer games. The engine is tipically considered as a game's core.

graphical elements) and interact with certain parts of the system which typically are external to the engine itself, such as music/sound effects, AI and other subsystems.

UT's architecture is quite complex, considering a computer game project. The approach consists of the use of a Virtual Machine, a Compiler and script and byte code (similar to the Java language architecture), as showed in Figure 1.

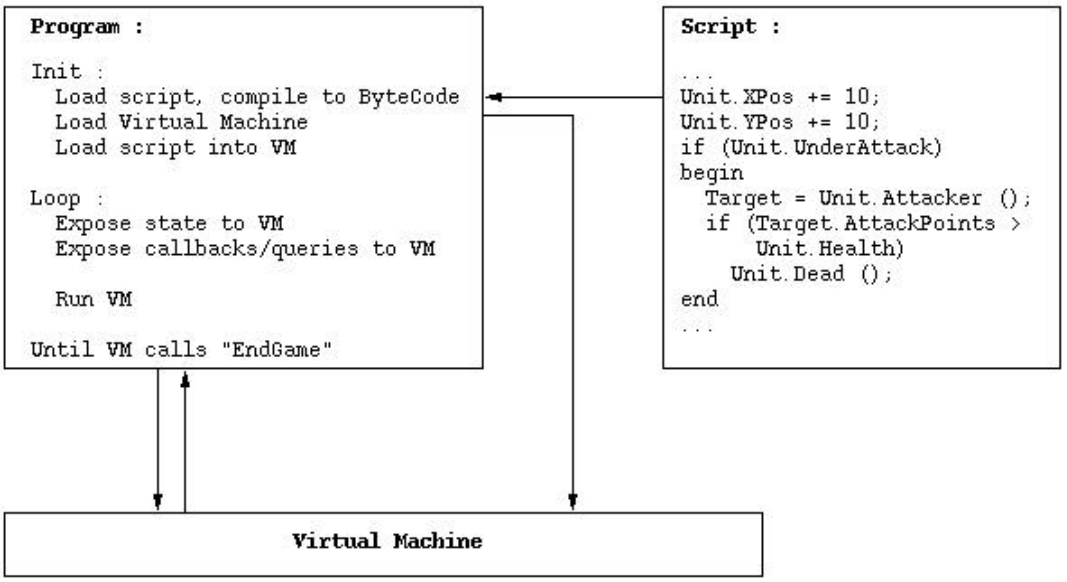

Figure 1: *Unreal* architecture scheme [PEE02].

The *Init* sequence compiles the script code into byte code and loads it on the Virtual Machine to be executed. From this point, the game's execution loop begins. During the game's execution, the system's state changes. The Virtual Machine executes functions and service calls until an "EndGame" request is called.

The *Script* part of the scheme is represented by *Unreal's* built-in, integrated script language, US. The script language was created to provide a personalized programming tool, which was first used by *Unreal's* development team during the game's implementation phase. Now many unreal players who have programming knowledge use the script language.

The language is object-oriented. The implemented code is compiled by using the UCC tool, a compiler created by *Unreal's* development team to be used with the language, and then be executed by the Unreal Virtual Machine, which is basically represented by the game's engine.

For the most part, US is very close to Java in syntax and behavior (a mix of C++ and Java features). Objects and inheritence are treated in a very similar way, and source code is compiled into byte code. There is an environment in which objects are

instantiated and data is bound dynamically. Function calls operate by making references to the underlying native code where appropriate. The parser uses a two-pass strategy.

1. Like Java, there is no need to use pointers directly or destroy objects. There is a service comparable to garbage collection that handles this task.
2. There are NOT separate header/source files. Declarations and definitions are done in xxx.uc source file.
3. Source files are compiled into a byte code object file, xxx.u.
4. Class references can only be made to classes defined in the same package or in one that was compiled before the class that is referencing it.

Package compilation order is determined by its order in the EditPackages section of the xxx.ini file that UCC.exe uses ("UnrealTournament.ini" by default). The default package compilation order (followed by a short descripition of the package) found in "UnrealTournament.ini" is described in Table 1:

Table 1: default UT packages, in their compilation order

Package name	**Description**
Core	contains fundamental unrealscript classes. Core, like many .u files, has a related DLL. The DLL contains the C++ part of the package
Engine	contains the core definitions of classes which are useful for content development (e.g. GameInfo describes basic game rules; PlayerPawn describes basic player behavior)
Editor	contains classes relevant to the Unreal Editor (UnrealEd)
Uwindow	contains the basic classes relevant to the game's windowing system
Fire	contains the US interface to the "Fire Engine.", which makes fire and water effects in the game
IpDrv	contains classes for handling UDP or TCP interfaces. Also used by the game's IRC interface in the game
Uweb	contains classes for remote web administration
Ubrowser	contains the core classes for the in-game server browser
UnrealShare	contains the code from the shareware version of Unreal
UnrealI	contains 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

Umenu	contains any menus for UT that don't depend on Botpack
IpServer	contains the querying interface for GameSpy, an online gaming service
Botpack	contains the game logic for *Unreal Tournament*
UTServerAdmin	contains *Unreal Tournament* specific web admin code
UTMenu	contains UT menus that require content from Botpack
UTBrowser	contains UT's browser code that requires content from Botpack

When developing content for UT, usually a programmer would insert the newly created package right after the last package in the EditPackages list (in this case, UTBrowser), so that the new package can inherit the properties from all the previous packages, thus allowing for the programmer to use or modify existing code.

3. Basic Syntax

In general, the language is similar to C/C++ in its syntax. 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. Operators and expression elements (+, -, /, =, ==) are used in the same way as in C++. There are some exceptions, though: all keywords, variable names, functions, and object names in US are case-insensitive. To US, "Unreal", "UnreAL", and "unreal" are the same thing. The following sections describe the laguage's elements, pointing out similarities and differences when comparing US with C/C++ and/or Java.

3.1. Operators

US provides a wide variety of C++/Java-style operators for such operations as adding numbers together, comparing values, and incrementing variables. Table 2 below shows the standard operators, in order of precedence (the last in the table has the highest priority).

Table 2: list of standard operators

Operator	Type(s) it applies to	Description
$	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 exclusive 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	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

In addition, UnrealScript supports the following unary operators:

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

3.2. 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.

There are two commands which can be used interchangably for inheritance: *extends* and *expands*, as shown below:

```
- class MyGame expands TournamentGameInfo;
```

or...

```
- class MyGame extends TournamentGameInfo;
```

The class declaration must be the first line of code. The *#exec* commands, useful for loading and importing resources (sounds, textures, models, etc.) into the code, must follow immediately after the class declaration.

3.3. Types and Variables

Variables can appear in two different places in US: 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.

To declare regular global variables, the word *var* must be used, followed by the variable's type and name. Variables cannot be initialized in their declarations. The examples below illustrate this:

- `var int MyInt;`
- `var int MyInt=100; (wrong!)`

In addition, variables can be made into editable properties which designers can access through menus in UnrealEd without any programming:

- `var() int HitDamage; //variable is accessible main class properties menu`
- `var(AI) float SightRadius; //variable is accessible through the "AI" properties sub-menu`

To declare a variable inside of a function, change the "var" word for "local". This will inform the garbage collector that the variable must be recycled when a function goes out of scope.

```
- function myFunction( ) {
                local int myInt;
      }
```

Variables can also contain additional specifiers that further describe the variable. The available variable specifiers are:

- *private*: the variable can only be accessed by the class' script, meaning that no other classes (including subclasses) can access it. It works the same way as in Java, but not used to the same extent;
- *const*: the variable's content cannot be changed. This specifier is usually for variables which the engine is responsible for updating;
- *input*: makes the variable accessible to UT's input system, so that input (e.g. button presses and joystick movements) can be directly mapped onto it;
- *transient*: informs that the variable is for temporary use, and will not be saved to disk;
- *config*: variable is configurable through the use of .ini and .int files (such as UnrealTournament.ini and UnrealTournament.int). The variable is stored in a configuration file;

- *native*: informs that the variable is manipulated by native C++ code rather than US code;

US supports a very diverse set of variable types including most base C++/Java types, object references, structs, and arrays. Here are the basic variable types supported in UnrealScript:

- **byte**: a single byte value ranged from 0 to 255;
- **int**: a 32-bit integer value;
- **float**: a 32-bit floating point number. Floats ranging from -1 to 1 must start with a 0 (e.g. 0.314, not .314);
- **bool**: a boolean value. Either "true" or "false";
- **enum**: a variable that can take on one of several predefined name values. Enums are stored internally as a byte. Here is an example:

```
enum EColor{
   CO_Red,
   CO_Green,
   CO_Blue
   };
   var EColor ShirtColor, HatColor;
```

- **string**: a string of characters. String is a primitive type, not an object. Strings can be concacenated with the use of the "$" operator. For example:

```
string s, t;
s = "Rocks";
t = "UT" $s $"MyWorld"; // t = "UTRocksMyWorld"
```

  - relevant string-related functions:
    - int Len(string) - returns the length of the string;
    - int inStr(string s, string t) - returns the offset of the first appearance of t in s, or -1 if it is not found;
    - string Mid(string s, int i, optional int j) - returns the first j characters of the string starting at offset i;
    - string Left(string, int i) - returns the i leftmost characters of the string;
    - string Right(string, int i) - returns the i rightmost characters of the string;
    - string Caps(string) - returns the string converted to uppercase;

- **name**: the name of an item in UT (such as the name of a function, state, class, etc.). Names are stored as a 16-bit index in the global name table, thus an object's Name is guaranteed to be unique within a level. Names correspond to simple strings of 1-31 characters. Names are not like strings, though: strings can be modified dinamically, while names can only receive predefined values.

  e.g. `SniperRifle12`, `TMale2`, etc...

- **struct**: similar to C structures, US structs let you create new variable types that contain sub-variables.There are some predefined structs, listed below:
  - *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);
  - *Rotator*: a rotation defining a unique orthogonal coordinate system. A rotation contains Pitch, Yaw, and Roll components;
  - *Color*: an RGB color value;
  - *Region*: defines a unique convex region within a level;
- **array**: US has some restrictions concerning arrays. Only one-dimensional arrays are valid. Also, only static arrays are valid, meaning that the array size cannot be defined with a variable, only with a literal.
  - relevant array functions:
    - int arrayCount(array) - returns the number of elements in the array;
- **Object**: a variable that refers to another object or actor in the environment. For instance: the Pawn class (which represents an Actor with AI capabilities) has an "Enemy" actor reference that specifies which actor the pawn should be trying to attack. Callingou "Enemy.Damage(123)" is calling the enemy's Damage function – resulting in the enemy taking damage. Object and actor references are powerful tools, because they enable you to access the variables and functions of another actor. Instantiation is done like the in following example:

```
Console C;
```

```
C = new (none) class 'myNewConsole'; /*
instantiate a new Console of type myNewConsole */
```

Some important considerations:

- "none" is the equivalent of NULL in Java/C++;
- A "." is used for object resolution (e.g. `myActor.moveTo(myAmmo.location);`);
- Instantiation can be done in two ways, depending if the instantiated object derives from the Actor class or not (in the above example, a non-Actor object was instantiated);

  Objects vs. Actors: Instantiation for actors is done like by calling the Spawn function:

  ```
  Actor A; // called from an actor
  A = spawn(class 'BoomStick', self, , location, rotation);
  ```

  Where:

  - *class*: describes the class of the object that is to be spawned (must be of type Actor);
  - *Actor*: owner of the newly spawned Actor (optional);
  - *Name*: name of the new Actor (optional);
  - *Vector*: location of the new Actor (optional);
  - *Rotator*: rotation of the new Actor (optional).

- **class**: class is a special kind of object which describes a class of object. In UT, classes are objects just like actors, textures, sounds and other objects. Class objects belong to the class named "class". This is used in situations where there is the need to store a refecence to a class object, so that it is possible to instantiate an object/actor belonging to a certain class without knowing what the class is at compile time. For example:

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

  When declaring variables that reference class objects, there is the option to use the class<classlimiter> syntax to limit the variable to only containing references to classes which expand a given superclass. For example:

```
var class<actor> ActorClass; //variable can only
reference a class which expands the "Actor" class.
```

3.3.1. Type Conversions

Conversions among all numerical data types (byte, int and float) happen automatically. There are also functions that can convert among some data types, for example:

```
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.
        s = string(v); // Convert vector v to a string.
        v = q + vector(r); // Convert rotation r to a vector.
}
```

Table 3 below describes the complete set of non-automatic conversions that can be done in US:

Table 3: US available type conversions.

Conversion	Description
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

3.4. Selection Commands

US's selection commands are very similar to the ones used in C++ and Java, as we will see below.

3.4.1. If 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" );
}
```

It is important to mention that "None" is not treated in the same way as "NULL" in C++. Boolean statements are needed. For example:

```
if( Actor != None ) //correct
{...}

if( !Actor )        //wrong !
{...}
```

3.4.2. Switch Statements

"Switch", "Case", "Default", and "Break" let you handle lists of conditions easily. It is used the same way as in Java/C++, like in the example below:

```
// 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;
    }
}
```

3.5. Loops

US supports all the standard flow control statements of C++ and Java, as we will see below.

3.5.1. For

"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
```

It is not possible to declare a variable in the condition section of the "for" loop:

```
for( i=0; ... )     //correct

    for( int i=0; ... )     //wrong !
```

3.5.2. While

"While" loops let you cycle through a loop while some starting expression is true. Used the same way as in Java/C++.

```
// 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
```

3.5.3. Do-Until

"Do"-"Until" loops let you cycle through a loop while some ending expression is true. Note that US'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
```

3.5.4. Foreach

US’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. This can be used 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. Table 4 below shows a list of all iterator functions that work with "foreach".

Table 4: list of iterator functions which work with the "foreach" loop.

Iterator function	Description
AllActors (class BaseClass, out actor Actor, optional name MatchTag)	Iterates through all actors in the level. If an optional MatchTag is specified, it will only include actors which have a "Tag" variable matching the specified tag
ChildActors(class BaseClass, out actor Actor)	Iterates through all actors owned by this actor
BasedActors(class BaseClass, out actor Actor)	Iterates throgh all actors which are standing on this actor
BasedActors(class BaseClass, out actor Actor)	Iterates through all actors which are touching (interpenetrating) this actor
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
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)
VisibleActors(class BaseClass, out actor Actor, optional float Radius, optional	Iterates through a list of all actors who are visible to the specified location (or if no

vector Loc)	location is specified, this actor's location)

3.6. Functions

In US, functions[4] can be declared (for new functions) or overriden with new versions of existing functions. Functions can take one or more parameters (of any variable type US supports), and can optionally return a value.

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.

Inside the function, it is possible to declare local variables (using the "local" keyword"), and execute any US code. The optional "return" keyword causes the function to immediately return a value. It is possible to pass any US types to a function (including arrays), and a function can return any type. By default, any local variables declared in a function are initialized to zero. Function calls can be recursive.

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 the actor. 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!" );
}
```

[4] Though most functions are written directly in US, there is also the possibility to declare functions that can be called from US, 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 functions.

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.

In regular function calls, US makes a local copy of the parameters passed to the function. If the function modifies some of the parameters, those don't have any effect on the variables passed in. The "out" word tells a function that it should actually modify the variable that is passed to it, rather than making a local copy. 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;
}
```

Other special parameter keywords:

optional: with the "optional" keyword, it is possible to make certain function parameters optional, as a convenience to the caller. For US 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:

```
function bool bNoArgs(optional int j, optional string s,
optional Actor A)
{
      if( (j == 0) && (s == "") && (A == none) )
            return true;
}
```

- *coerce*: the "coerce" keyword forces the caller's parameters to be converted to the specified type (even if US 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 example:

```
function returnString(coerce string i, coerce string c,
coerce string a)
```

```
{
	return i $c $a;
}
```

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;
- *native*: US functions can be declared as "native", which means that the function is callable from US, 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;
- *exec*: can be used to have this function callable from the game console or using consoleCommand(string);
- *latent*: declares that a native function is latent, meaning that it can only be called from state code, and it may return after some game-time has passed (we will see states in more detail in the next section).

3.6.1. Function Overriding

"Function overriding" refers to writing a new version of a function in a subclass. For example, a script for a new kind of monster, called a Demon, is being written. The Demon class, 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. With function overriding, the "SeePlayer" function could be modified in the newly created Demon class, so that the Demon acts in a different way.

To override a function, just cut and paste the function definition from the parent class into the new class. For example, for SeePlayer, the following code could be added to the 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. New classes can be created through expanding the contents of an existing class and then overriding the functions which need to be changed. This enables the programmer 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 will most likely not be overriden, because it results in faster script code. For instance, a "VectorSize" function, that computes the size of a vector. There's no reason for overriding this kind of function, thus the "final" keyword is useful. On the other hand, a function like "Touch" is very context-dependent and should not be declared as "final".

4. Advanced Syntax

This section covers advanced aspects of US programming.

4.1. States

Historically, game programmers have been using the concept of states ever since games evolved past "Pong". States (and what is known as "state machine programming") are a natural way of making relatively complex object behaviour manageable. However, before US, 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. US supports states at the language level.

In US, 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".

It is possible to 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.

Some important considerations about using states:

- Benefit: states provide a simple way to write state-specific functions, so that it is possible to handle the same function in different ways, depending on what the actor is doing;
- Benefit: with a state, it is possible to write special "state code", using all of the regular US 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, it is possible to write code in the same way the code was conceptualized;
- Complication: since it is possible to have functions (like "Touch") overridden in multiple states as well as in child classes, there is the problem to know which function is going to be called in a specific situation. US

provides rules which clearly delineate this process, but the developer must be aware when creating complex hierarchies of classes and states.

Here is a simple example of the use of states taken 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' );
    }
}
```

It is possible to 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
{
    //...
}
```

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

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

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. Any label name can be given to state code, but the "Begin" label is special: it is the default starting point for code in that state.

Two native US functions are particularly useful when writing state code:

- *Goto* (similar to the C/C++/Basic goto): within a state, it 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;
- *GotoState*: this function causes the actor to go to a new state, and optionally continue at a specified label (if a label is not specified, the default is the "Begin" label). 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. *GotoState* can be called from within state code, and it goes to the destination immediately. It is also possible to 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.

Whever you use the *GotoState* command to set an actor’s state, the engine can call two special notification functions: (if defined): *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.

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...
}
```

4.2. Default Properties of a Class

The *DefaultProperties* section of a class is the last section of code in a class file. It is used to set and/or initialize variable values. Most properties (variables declared in the *DefaultProperties* section) are accessible through *UnrealEd*'s menus.

Property declarations cannot end with a semicolon (";") and there can be no spaces between the "=", the variable name and value. For example:

```
Defaultproperties{
      ThisValue=100     //correct
      OtherValue = 10   //wrong !
      OneMoreValue=5;   //wrong !
}
```

The developer has the possibility 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 US, it is possible to 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;
}
```

5. Debugging US code

Debugging in US is done by calling functions that write to log files, standard output (screen) or by executing a part of the code:

- *Log(string)*: this function writes the string to any xxx.log file is defined in the games xxx.ini file;
- *BroadcastMessage(string)*: this function will write a text message to the player's interface in the game (usually called the Heads Up Display, or HUD), so that it makes possible to do runtime checking. It can only be called by classes inheriting from the Actor class;
- *Console Commands*: It is possible to declare a function that can be called through the in-game console (a part of the player's interface) by using the *exec* function specifier.

6. References

[MAR02] MARTIN, R. J. **UnrealScript Syntax.** Capturado em Abril de 2002. Disponível na Internet em: http://mimesis.csc.ncsu.edu/Unreal/Syntax.htm

[SWE01] SWEENEY, T. **UnrealScript Language Reference.** Capturado em Novembro de 2001. Disponível na Internet em: http://unreal.epicgames.com

[PEE02] PEERS, B. **Writing Modifiable Games.** Capturado em Março de 2002. Disponível na Internet em: http://lgdc.sunsite.dk/articles/3.html

[CUN01] CUNHA, L. S.; GIRAFFA, L. M. M. **Um estudo sobre o uso de Agentes em Jogos Computadorizados Interativos.** PPGCC/PUCRS, Porto Alegre, 2001. Technical Report (available at http://www.inf.pucrs.br/ppgcc).

[LAI01] LAIRD, J. E. **Using a Computer Game to Develop Advanced AI.** Computer, 34, (7), 2001.