LEARNING GAME

TEACHER TRAINING MANUAL

MULTIMEDIA APPLICATIONS FOR EDUCATION

Index

Part 5B: PROGRAMMING

PART FIVE/B

PROGRAMMING

Chapter Three
How to Programme Videogames with SDK Editors

University of Patras (GR)

1. GTK Radiant

1.1 Introduction

GtkRadiant is a level design program developed by id Software and Loki Software. It is used to create maps for a number of computer games. It is maintained by id Software together with a number of volunteers.
GtkRadiant's roots lie in id Software's in-house tools. Some of the early UI design decisions influencing it could be seen in QuakeEd, the original *Quake* mapping tool for NextStep. Code-wise, GtkRadiant is a descendant of Q3Radiant, the *Quake III Arena* level design tool, which in turn is a descendant of QERadiant. QERadiant was developed by Robert Duffy using the source code for QE4, the in house *Quake II* level editor id Software used to build *Quake II* levels and is available with the Quake 2 SDK. All three are Windows-only applications. Two major things are different in GtkRadiant: it is based on the GTK+ toolkit, so it also works in Linux and Mac OS X, and it's also game engine-independent, with functionality for new games added as game packs.

1.2 Basics

Scripting

Scripting brings your map to life. It's the part of mapping that allows players to interact with the map environment, build things, destroy objects, move vehicles, cap objectives, and win the game. To use a computer analogy, your map is the nice shiny box sitting on your desktop. It's great to look at, but without the applications and programs you load into it (the script), it's incapable of really doing anything.

Map Scripting (general)

What Do you Need To Get yor Map Script Working?
Script_Multiplayer entity. Think of this almost like the autoexec.cfg in your game script. The script_multiplayer automatically launches the game_manager script every time you launch your map
Text Editor (Notepad works fine)
Examples (can be found in the /maps folder of any .PK3 file and will be named [mapname].SCRIPT
Game_manager routine, victory script (typically found inside the game_manager routine), and your objective scripts, which you can find in lot of tutorials

How Do Scripts Get Initiated?

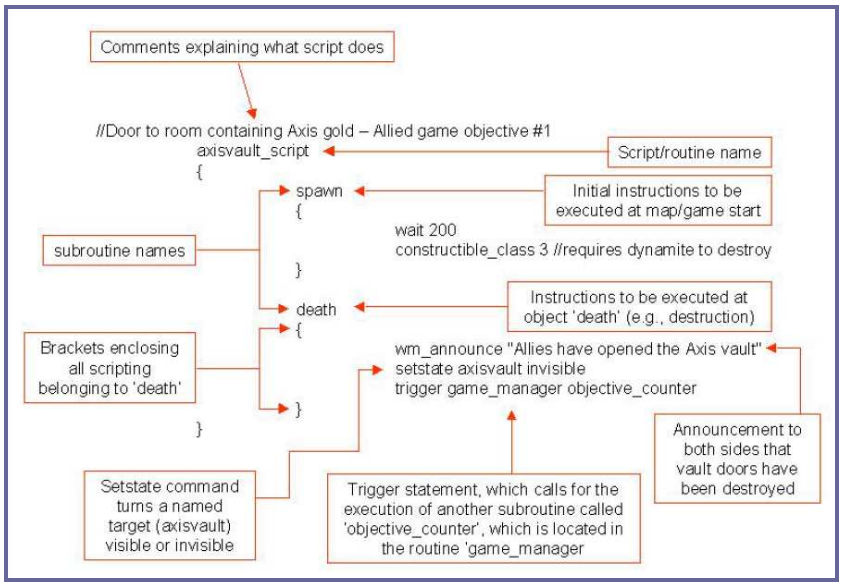
Scripts can be initiated in one of three ways:
Due to the occurrence of a game event (e.g., spawn, time expiration)

Due to a change in the status of an entity (e.g., construction completed, death of a func_destructible). This is typically caused by a player action via a trigger, but not necessarily so. An example of this would be an unfinished construction, which automatically reverts to the unbuilt state after 30 seconds of inactivity
Called by another script. You can chain scripts together, using the 'trigger' function, so that one script calls another and so forth. An example of this would be the objective_counter and gamecheck scripts, which are executed every time an objective is completed to see if the game is over.

The Basic Script Elements

You can think of each script having 3 distinct elements
Script Name
Script Contents
Spawn
Additional entity states
Additional instructions added by mapper
Comments
If you're familiar with writing scripts for your player config, then understanding how map scripts are organized should be a snap. In your player autoexec, you might have the following line:
Set gamename2 "name ^4RU5" //changes to pub name
That line says, name the script 'gamename2' and when it's executed, change my name to RU5, in blue, then has a comment (everything after //) to remind me, or tell others what that line does. Everything except for the script name and the comments are encased in quotations ("), to show the game engine where the script starts and stops.
A key difference with map scripting is the concept of script hierarchy. I like to think of scripts in terms of routines and subroutines. In fact, after this point, I'll use the terms 'routine' and 'subroutine' interchangeably with the term 'script'. Just like a book is organized into chapters and paragraphs, a script has higher-level routines (chapters), which reflect a single overarching object (e.g., a destructible objective) and lower-level sub-routines (paragraphs), which reflect distinct states or stages of the routine's subject. For example, one routine might be the script for a destroyable wall, called '*old_city_wall*' with subroutines called '*spawn*' and '*death*'. That's simple enough. Another, more complex, routine might be the script for a constructible fence, called '*fuel_dump_gate*' with subroutines (states) of *spawn*, *buildstart*, *built*, *decayed*, and *death*.
Each standalone portion of script starts with a '{' and ends with a '}', much like quotes are used to start and end a script in your player config.
Here is the script for a simple func_destructible, which calls a victory script upon destruction, to see if the game should end:

Anatomy of a Simple Section of Script



```
//Door to room containing Axis gold – Allied game objective #1
axisvault_script
{
        spawn
        {
                wait 200
                constructible_class 3 //requires dynamite to destroy
        }
        death
        {
                wm_announce "Allies have opened the Axis vault"
                setstate axisvault invisible
                trigger game_manager objective_counter
        }
}
```

Comments explaining what script does
Script/routine name
Initial instructions to be executed at map/game start
subroutine names
Instructions to be executed at object 'death' (e.g., destruction)
Brackets enclosing all scripting belonging to 'death'
Announcement to both sides that vault doors have been destroyed
Setstate command turns a named target (axisvault) visible or invisible
Trigger statement, which calls for the execution of another subroutine called 'objective_counter', which is located in the routine 'game_manager

Again, we use comments to remind people what the script does. In this case, reminding us that this script controls the door protecting the axis gold, which the allied have to blow up, and that the door requires destruction by dynamite.
The routine has a name, '*axisvault_script*', that makes it easy to figure out what it's all about. You can really call your routine anything you want, but it's much easier when the name is short, simple, and related to its function. Each of the subroutines has a name too. The specific required names of your subroutines is usually determined by the supported entity. In the case of a *func_destructible*, there are two: spawn and death. In all cases, each standalone portion of the script, after the script name, starts with a {and ends with} to signify what is associated under the scripts named '*axisvault_script*', '*spawn'*, and '*death'*. Also note, that each hierarchical level is indented to more easily show script relationships. You don't have to do this, but it sure makes editing, proofreading, and debugging scripts much easier.
Every script needs to have a subroutine called spawn. It doesn't have to contain any coding or instructions, but needs to be there. Anything in the 'spawn' subroutine automatically gets executed at the start of the game. If you do have instructions in the spawn subroutine, then added a short 'wait xxxx' or you may end up with annoying little errors here and there.

Analyze the morph of a Script Example

Let's open up an example .SCRIPT file (*Oasis.scrip*t, a pretty straightforward map with 2 destructible objectives and no moving vehicles) and take a look at the first part you'll see, the *game_manager* routine. The rest of the routines are going to pretty specific to particular entities (*func_constructibles*, *func_destructibles*, *forward spawn flags*, *command posts*, *movers*, *etc*). Open it with Notepad. I've copied the *game_manager* portion here and deleted the portions related to VOs, pumps, and the forward spawn.

```
game_manager
{
spawn
```

Every map script needs to have a *game_manager* section and every routine needs to have a spawn subroutine, even if it contains no instructions.

```
{
accum 1 set 0 // State of objective number one
accum 5 set 0 // Current number of Pak 75mm guns destroyed
accum 6 set 0 // Current number of water pumps built
```

```
   accum 7 set 0 // Value used in checking whether or not to announce
"Axis have damaged both water pumps!"
   globalaccum 5 set 0
   globalaccum 6 set 0
```

Setting all accum and global accum values to 0 at the beginning of the game is a good way of ensuring that all objectives are reset to their intended starting positions. It's just like starting your custom player config with 'unbindall'

```
   // Game rules
   wm_axis_respawntime 30
   wm_allied_respawntime 20
   wm_number_of_objectives 8
   wm_set_round_timelimit 30
```

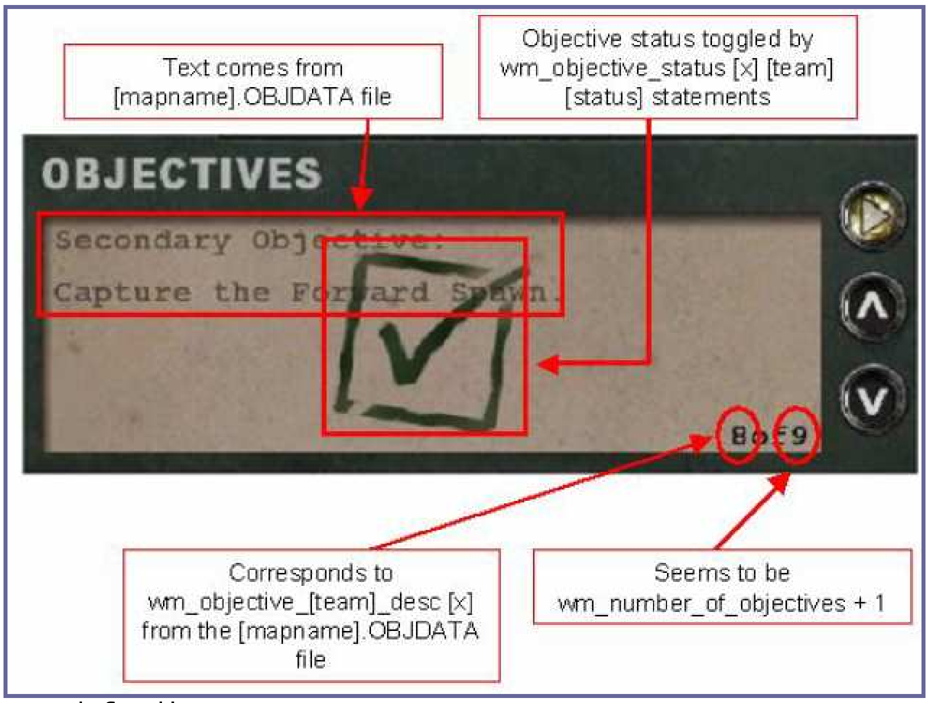
This section sets the basic game rules for match length and respawn times. Wm_number_of_objectives should match the number of objectives in your .OBJDATA file

```
   // Objectives
   // 1: Primary1 : Destroy the North gun
   // 2: Primary2 : Destroy the South gun
   // 3: Primary3 : Breach Old City wall
   // 4: Secondary1 : Capture forward spawn point
   // 5: Secondary2 : Drain/flood cave system by repairing/damaging the
Oasis pump
   // 6: Secondary3 : Drain/flood cave system by repairing/damaging the
Old City pump
   // 7: Allied command post
   // 8: Axis command post
```

Many people add a comment section here to remind them what are the map's objectives

```
   // Current main objectives for each team (0=Axis, 1=Allies)
   wm_set_main_objective 3 0
   wm_set_main_objective 3 1
   // Objective overview status indicators
   //wm_objective_status <objective> <team (0=Axis, 1=Allies)> <status
(0=neutral 1=complete
   2=failed)>
   wm_objective_status 1 1 0
   wm_objective_status 1 0 0
   wm_objective_status 2 1 0
   wm_objective_status 2 0 0
   wm_objective_status 3 1 0
   wm_objective_status 3 0 0
   wm_objective_status 4 1 0
   wm_objective_status 4 0 0
   wm_objective_status 5 1 0
   wm_objective_status 5 0 0
   wm_objective_status 6 1 0
   wm_objective_status 6 0 0
   wm_objective_status 7 1 0
   wm_objective_status 7 0 0
   wm_objective_status 8 1 0
   wm_objective_status 8 0 0
```

You have to manually tell the game which team controls each objective. This is done here, to show all objectives as uncompleted, as well as later in the script as various scripts get triggered and objectives are completed. As the comment above notes, the format is wm_objective_status [obj number (1 thru n)] [team (0 or 1)] [status (0 thru 2]. In-game, as the game progresses, you will see checkmarks in a box or x's in a box for each objective listed in the Limbo menu. The wm_objective_status commands make that happen. Here's a quick explanation that links what you see in the Limbo menu with commands in your .SCRIPT and .OBJDATA files:


Text comes from [mapname].OBJDATA file
Objective status toggled by wm_objective_status [x] [team] [status] statements
OBJECTIVES
Secondary Objective:
Capture the Forward Spawn.
8 of 9
Corresponds to wm_objective_[team]_desc [x] from the [mapname].OBJDATA file
Seems to be wm_number_of_objectives + 1

```
    wm_set_defending_team 0
    // If the round timer expires, the Axis have won, so set the current
winning team
    // Set the round winner: 0 == AXIS, 1 == ALLIED
    wm_setwinner 0
    // Set autospawn markers <team (0 = axis, 1 = allies)> <message key
of marker>
    // Spawns on siwa:
    // Axis Garrison
    // Allied Camp
    // Old City
    wait 150
    setautospawn "Old City"
    setautospawn "Old City" 0
```

The name of the spawnpoint used comes from the description key of the Team_Wolf_Objective entity. Both teams have their spawnpoints autoset for the Old City which makes people spawn as close to the old city flag as possible. Effectively, this means that as soon as a team captures the forward flag, you'll automatically spawn there.

```
wait 2000
}
trigger 75mm_gun_counter
{
accum 5 inc 1 // Increase game counter
accum 5 abort_if_not_equal 2 // All guns destroyed ?
wm_announce "Allies have destroyed both Anti-Tank Guns!"
wait 8000
accum 1 set 1 // Both pak guns destroyed
// Call function to check if the round has been won
trigger game_manager checkgame
}
```

This is a simple objective counter script that counts how many objectives (Pak 75mm guns) have been destroyed. Whenever the North or South gun is destroyed, it calls the gun_counter script trigger checkgame

```
{
accum 1 abort_if_not_equal 1
// Set the round winner: 0 == AXIS, 1 == ALLIED
wm_setwinner 1
// End the round
wm_endround

}
```

Each time a gun is destroyed, we check to see if the game has been won by checking if accum 1 is equal to 1. Accum 1 is set to 1 in the gun_counter routine only when both guns have been destroyed. If so, the Allied team is declared the winner. If not, the script stops and the game continues.

1.1 Script Writing Examples

Here are a couple simple scripts, three of which, we will combine to determine if the game is over after an objective has been completed.

1.3.1 Objective Scripts

For the sake argument, let's limit our game objectives to one of three types. The attacking team either has to build something, destroy something, or return an objective. So, before we go on, first read thru the following 5 scripts and at least be comfortable with reading thru them. Scripts 1a –1c are entity scripts that I won't cover here. Scripts 2 and 3 are similar to the *75mm_gun_counter* and *checkgame* scripts in the example *game_manager* from Oasis that we looked at previously

```
    1a. Func_constructible
    //Axis safehouse. Axis team has to build a gate in front of a
safehouse

    safehouse_script
    {
    spawn
    {
    wait 200
    constructible_class 2
    trigger self startup
    }

    buildstart final
    {
    }

    built final
    {
    setstate safehouse default
    setstate safehouse_materials invisible
    wm_announce "The Axis safehouse door has been constructed"
    }

    decayed final
    {
    trigger self startup
    }

    death
    {
    trigger self startup
    wm_announce "The Axis safehouse has been breached"
    setstate safehouse_materials default
    }

    trigger startup
    {
```

```
setstate safehouse invisible
setstate safehouse_materials default
}
}

1b. Func_destructible
//Door to room containing Axis gold – Allied have to dynamite a door protecting a box of Axis gold

axisvault_script
{

spawn
{
wait 200
constructible_class 3
}

death
{
wm_announce "Allies breached the Axis gold vault"
setstate axisvault invisible
}
}

1c. Team_CTF_redflag
//Radar. Axis team must capture and secure an Allied radar module containing critical codes

allied_radar //enter this as the scriptname value for the team_CTF_red/blueflag entity
{
spawn
{
wait 200
setstate allied_radar_captured invisible
}

trigger stolen
{
wm_announce 0 "Return the Allied radar set to the getaway truck"
wm_announce 1 "The Axis have stolen the Allied radar set"
setstate allied_radar_cm_marker invisible
}

trigger returned
{
wm_announce 0 "The Allies have retrieved the radar set"
wm_announce 1 "Radar set returned! Protect the radar set"
```

```
setstate allied_radar_cm_marker default
}

trigger captured
{
wm_announce "The Axis have secured the Allied radar components"
setstate allied_radar_red invisible
setstate allied_radar_captured default
}
}
```

1.3.2 Objective Counter

```
trigger objective_counter //Counts allied objectives completed
{
accum 1 inc 1
trigger game_manager checkgame
}
```

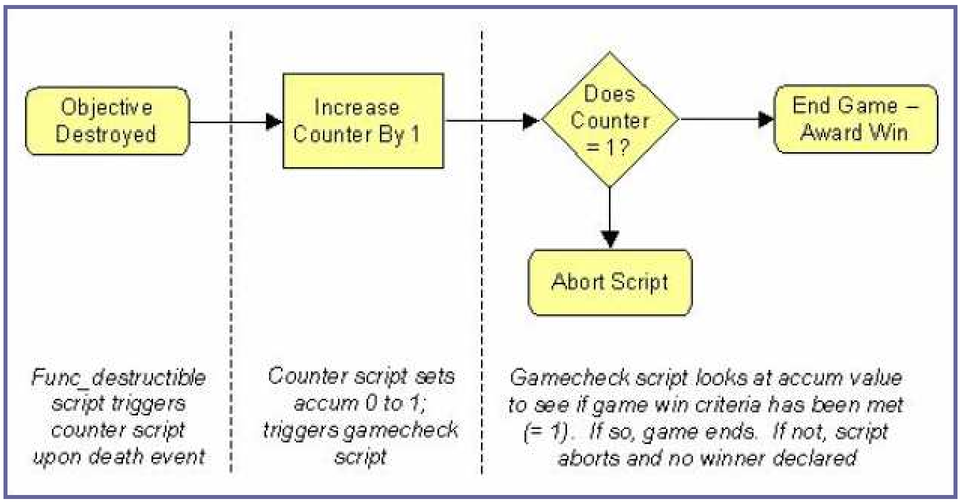
1.3.3 Game Win Check Script

```
trigger checkgame
{
accum 1 abort_if_not_equal 1
wm_setwinner 0
wait 1500
wm_endround
}
```

1.3.4 Combining these Scripts to determine if the game is over

Here are 3 examples in order of increasing complexity.
Single objective, which must be destroyed by Allied team
Two objectives, which must be destroyed by Allied team
Dual objective, both teams have two destructible objectives
We notice that all three of them really use the same scripts, just with more objectives, different variables, and slightly different organizations. For each, we have included a flow chart, the script, and an explanation of how it's changed from previous scripts.

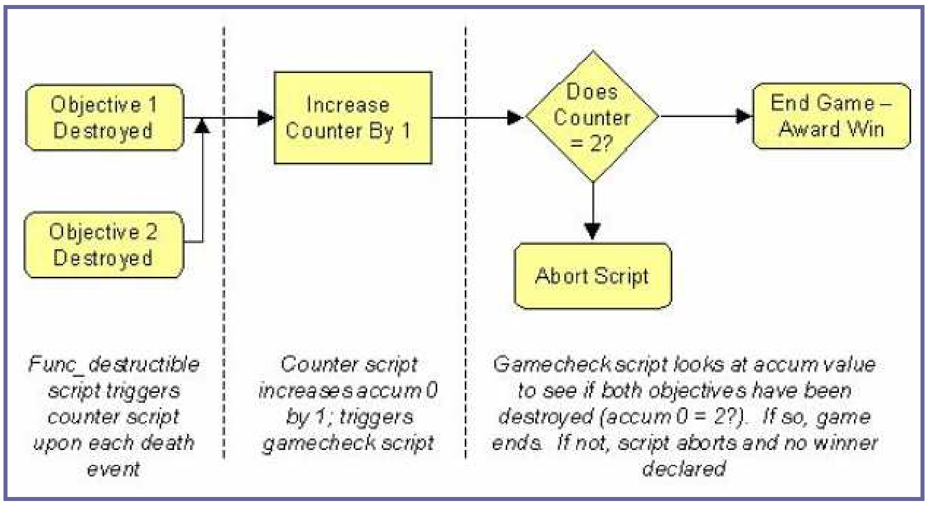
Single objective, which must be destroyed by Allied team


Objective Destroyed
Increase Counter By 1
Does Counter = 1?
End Game – Award Win
Abort Script
Func_destructible script triggers counter script upon death event
Counter script sets accum 0 to 1; triggers gamecheck script
Gamecheck script looks at accum value to see if game win criteria has been met (= 1). If so, game ends. If not, script aborts and no winner declared

In this example, we used scripts 1b, 3.2, and 3.3. We have a standard func_destructible script, which has an additional line, "trigger game_manager objective_counter", in the death subroutine. This is the only addition to the standard func_destructible script you find in most tutorials and means that whenever the target is destroyed, it automatically executes the script, 'objective_counter'. When call another script via a trigger statement, we have to specify in which routine it is contained (game_manager) and the name of the script itself (objective_counter).
'Objective_counter' increases the value of Accum 0 by 1, and then executes the script named 'gamecheck' to see if the game is over.
'Gamecheck' looks at the value of Accum 0 and if it is 1, it says, "yep, victory conditions are met, Allies win, and the game is over." If the value of Accum 0 is anything other than 1, the script just stops and does nothing.
Here are the scripts:

```
Game_manager
{
spawn
{
}
trigger objective_counter //Counts allied objectives completed
{
accum 1 inc 1
trigger game_manager checkgame
}

trigger checkgame
{
accum 1 abort_if_not_equal 1
wm_setwinner 0
wait 1500
wm_endround
}
}

//Door to room containing Axis gold
axisvault_script
{
spawn
{
wait 200
constructible_class 3
}
death
{
wm_announce "Allies breached the Axis gold vault"
setstate axisvault invisible
trigger game_manager objective_counter
}
}
```

Two objectives, which must be destroyed by Allied team

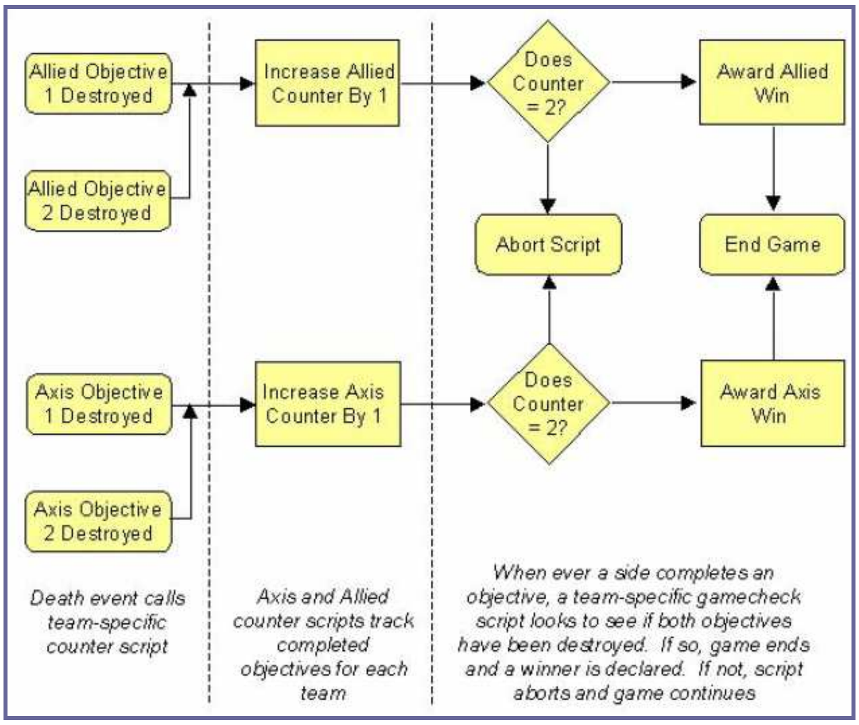

Objective 1 Destroyed
Objective 2 Destroyed
Increase Counter By 1
Does Counter = 2?
End Game – Award Win
Abort Script
Func_destructible script triggers counter script upon each death event
Counter script increases accum 0 by 1; triggers gamecheck script
Gamecheck script looks at accum value to see if both objectives have been destroyed (accum 0 = 2?). If so, game ends. If not, script aborts and no winner declared

This uses the exact same script as in case #1, except that the gamecheck script now looks for an Accum 0 value of 2. This makes sense, because each time one of the two objectives is blown up, the objective_counter script increases the value of Accum 0 by 1 until it has a value of 2, which signifies that both objectives have been blown up.

Here are the scripts:

```
Game_manager
{
spawn
{
}

trigger objective_counter //Counts allied objectives completed
{
accum 1 inc 1
trigger game_manager checkgame
}

trigger checkgame
{
accum 1 abort_if_not_equal 2
wm_setwinner 0
wait 1500
wm_endround
}
}

//Door to room containing Axis gold - Allied objective 1
axisvault_script
{
spawn
{
wait 200
constructible_class 3
}

death
{
wm_announce "Allies have breached the gold vault"
setstate axisvault invisible
trigger game_manager objective_counter
}
}

//Axis radar unit – Allied objective 2
axisradar_script
{
spawn
{
wait 200
constructible_class 3
}

death
{
wm_announce "Allies have destroyed the Radar unit"
setstate axisradar invisible
trigger game_manager objective_counter
}
}
```

Dual objective, both teams have two destructible objectives


Allied Objective 1 Destroyed
Allied Objective 2 Destroyed
Increase Allied Counter By 1
Does Counter = 2?
Award Allied Win
Abort Script
End Game
Axis Objective 1 Destroyed
Axis Objective 2 Destroyed
Increase Axis Counter By 1
Does Counter = 2?
Award Axis Win
Death event calls team-specific counter script
Axis and Allied counter scripts track completed objectives for each team
When ever a side completes an objective, a team-specific gamecheck script looks to see if both objectives have been destroyed. If so, game ends and a winner is declared. If not, script aborts and game continues

Again, almost no fundamental change from the previous script. The only change is that we've totally duplicated objective_counter and gamecheck and made them specific to each side. That is, there is one copy of the two scripts (allied_objective_counter and allied_gamecheck) which checks to see if the Allies have won the game whenever they blow up an objective and another copy of the two scripts (axis_objective_counter and axis_gamecheck) which checks to see if the Axis have won the game whenever they blow up an objective. Now, it's just a race to see which team can blow up their two objectives first.

1.2 The Accum Value

Accum values are the heart of your map's game logic. They are like a register or variable that are assigned or change based on the occurrence of selected game events. We used this earlier to count the number of objectives completed and based on that, determine whether or not the game was over.
They allow you to program events into your map that happen or don't happen based on the accum values and some limited set of accum math. For example, you can trigger an event if your accum value is:
equal to a stated target (e.g., IF number of completed objectives = 2, THEN execute win routines)
not equal to a stated target (e.g., IF number of completed objectives not = 2, THEN abort script)
less than a stated target
greater than a stated target
and many more
If you've ever used MS-Excel, this is very much like writing a function with IF statements. They can also be nested to allow for multiple paths, though the scripting required is a little more complex.

1.3 Accum vs. GlobalAccum

Accum values are specific to each entity and routine. So, I could use accum 1 in my objective_counter script as well as in my forward_spawn and command_post scripts. This could also lead to some confusion if you use the same scriptname with multiple entities. While they share the same scriptname, each will have its own, unique set of accum values. As a result, you might end up with unexpected results.
A globalaccum can be used by multiple scripts and entities and therefore can be very useful for sharing information across scripts and entities.

1.4 Set vs. Bitset

You can use accum values in one of two ways:
as a single discreet value, which ranges from x to y, or
as a 32-character bitset, which has a total of 32 separate on/off states.
'Set' allows you to set the value of accum x to any number. If you use set, you are limited to 8 discrete values per entity or script. An example of set might be used to count the number of completed objectives or flags captured. 'Bitset', allows you to set the value of bit (I won't explain binary math here) X to 1, and 'bitreset' allows you to set it to 0, this allows you to store 32 different on or off states. So, if all you need to know about parts something is whether they are on/off, dead/alive, etc, then you can use bitset and bitreset + the various check functions to your advantage, rather than using a single accum value for each one. An example of this might be a tank with 32 checkpoints that it needs to pass. Each can be thought of, in black and white, as passed/not passed.
I guess you can think of it this way, if you can need a range of values (up to 8) in your script, use 'set'. If your values can be expressed as yes/no or on/off states (up to 32) then you should use 'bitset'.

1.5 Commonly Commands

A lot of the commands you will see in a script are pretty self explanatory, but I thought I'd include them for the sake of completeness:

```
accum x set y – Set the value of accum #x to y
accum x inc y – Increase the value of accum #x by y
```

accum x bitset y – Turn bit #y of accum #x to the ON position (e.g., 1)

accum x bitreset y – Turn bit #y of accum #x to the OFF position (e.g., 0)

alertentity - triggers the entity with the given targetname

constructible class n (1-3) – Determines how much chargebar is required to finish construction and by what

weapons it can be destroyed

disablespeaker – Disables a speaker from playing

enablespeaker – Enables a speaker to start playing

gotomarker – Tells a mover to go to a specified point

remove – Removes an entity from the game (e.g., the Axis Old City Spawn on Oasis and the Axis Forward Bunker Spawn on Radar)

setautospawn "Team_Wolf_Objective description” [team] – Make players from Axis (team=0) or Allies (team=1) spawn as close to the selected spawn point as possible.

setchargetimefactor [team] [class] [% of default time required for a full charge] – Tells game to set time required for

a full recharge by [% of default time required for a full charge] for everyone who is a [class] on Axis (team=0) or Allies (team=1)

sethqstatus [team] [status] – Identifies command post as built (status = 1) or unbuilt (status = 0) for Axis (team=0) or Allies (team=1)

setstate [entity name] default – Make [entity name] visible

setstate [entity name] invisible - Make [entity name] invisible

startanimation – Starts a model animation

trigger [routine name] [subroutine name] – Execute the script named [subroutine name], which can be found in the

routine called [routine name]

wait x – wait for x milliseconds before continuing to the next step

wm_allied_respawntime x – Allow Allies to respawn every x seconds

wm_announce [team] “message” – Send the text “message” to Axis only (team=0, Allies only (team=1), or everyone (team=blank)

wm_objective_status [objective number] [team] [status] – Set the status of objective #[objective] to not completed (status=0) completed (status=1), or failed (status=2) for Axis (team=0) or Allies (team=1)

wm_axis_respawntime x – Allow Axis to respawn every x seconds

wm_endround – Ends the game

wm_number_of_objectives x – Set number of objectives to X

wm_removeteamvoiceannounce [team] “[name of voice over]”

wm_set_defending_team [team] – Set Axis (team=0) or Allies (team=1) as the defending team

wm_set_main_objective [objective number] [team]

wm_set_round_timelimit x – Let the game play for X minutes

wm_setwinner [team] – if the time limit expires, then award the win to Axis (team=0) or Allies (team=1)

2. Valve Hammer Editor

2.1 Introduction

Valve Hammer Editor, formerly known as Worldcraft and now commonly called Hammer, is Valve Software's map creation program for their first-person shooter computer game *Half-Life* and for all of its mods, sequels, and expansions, as well as all games using the Source engine. Old versions of Worldcraft also supported *Quake* and *Quake II*. There are currently two versions- version 3.5, a beta version (still available to the public, but no longer in development), which supports only *Half-Life* and its mods, but can be run without Steam. The current version, version 4.1, supports *Half-Life 2* for the *Source engine* as well as *Half-Life*, but requires Steam to be operating.
The Valve source engine programming features are:
All code written in C/C++ using Visual Studio 6.0. Easily and quickly derive new entities from existing base classes.
Internal context sensitive performance monitoring system
Graphics performance measurement tools built into the engine
Modular code design (via DLL's) allows swapping out of core components for easy upgrading or code replacement
Dx9 shaders all written in HLSL

2.2 Basics

Developer Console Control

Printing to the console

Printing text to the console is done the same way in all modules, since the Tier0 debug layer provides these routines. The 3 most common functions are Msg(), DevMsg() and Warning() which all support VarArgs, like sprintf():

```
DevMsg (char const* pMsg, ... )    - only in developer mode
Msg(char const* pMsg, ... )   - always, white
Warning(char const *pMsg, ... )    - always, red
```

For backward compatibility with HL1 source code you can still use Con_Printf() and Con_DPrintf().

Executing commands

The engine provides interface to server and client code to execute commands (strings) from code as if the user would have entered them. The server has to use the interface IVEngineServer::ServerCommand():

```
engine->ServerCommand("changelevel de_dust\n");
```

Client code has to use the interface IVEngineClient and can choose here between two functions, depending if the command should be executed on the client first or be sent directly to the server:

```
engine->ServerCmd( "say hello\n" ); // send command to server
```

or

```
engine->ClientCmd( "say hello\n" ); // execute command on client
```

2.3 Adding New Commands & Variables

The developer console is a subsystem provided by the Source engine and is accessible for all other modules via a public interface ICvar (see \public\icvar.h). This interface allows registering new commands and finding/iterating existing commands. This interface is

available via the global CVAR ingame server/client code (cv in engine code). Console commands are implemented in ConCommand and console variables in ConVar, which both derive from the base class ConCommandBase (see \public\convar.h).
Adding a new command or variable is fairly simple and the same code in both server and client (even engine) modules. The constructor of these classes automatically registers the command at the console system. This short example code adds a new command my_function and a new variable my_variable initialized to 42:

```
    #include <convar.h>
    ConVar my_variable( "my_variable", "42", FCVAR_ARCHIVE, "My favorite
 number" );


    void MyFunction_f( void )
    {
        Msg("This is my function\n");
    }


    ConCommand my_function( "my_function", MyFunction_f, "Shows a
 message.", FCVAR_CHEAT );
```

It is common use that the object name and the command name are the same and variables used only in a single source file are declared as static.

Using the class ConVar
First let's take a look at the most used ConVar constructor:

```
    ConVar( char const *pName,
      char const *pDefaultValue,
      int flags,
      char const *pHelpString )
```

The first argument pName is the variable name (no spaces), followed by pDefaultValue, which is always given as a string even for ConVars with numerical values. Flags specify special characteristics of the variable, all flag definitions start with FCVAR_*, more about these flags later. It's always good to provide a pHelpString, so users get an idea what this variable is about. ConVars are not bound to a certain type, their value can be an integer or a float or string and you may use it however you like. As long as you have the ConVars object itself or as a pointer, you can access and modify its values directly. All these examples are valid and have the same result:

```
    if ( my_variable.GetInt() == 42 ) DoSomething();
    if ( my_variable.GetFloat() == 42.0f ) DoSomething();
    if ( strcmp(my_variable.GetString(), "42")==0 ) DoSomething();
```

To set the value of a ConVar you should use the SetValue() function, which also allows all types of data:

```
    my_variable.SetValue( 42 );
    my_variable.SetValue( 42.0f );
    my_variable.SetValue( "42" );
```

At any time you can revert a ConVar back to it's original default value by using the Revert() function.
If a ConVar is created in a different module, the ICvar interface function FindVar() can be used to get a pointer to this object, if the variable name is known. This is an expensive search function and the pointer should be cached if reused often. Here is an example how to check the ConVars sv_cheats defined in the engine module:

```
    ConVar *pCheats  = cvar->FindVar( "sv_cheats" );

        if ( pCheats && pCheats->GetInt() == 1 ) AllowCheating();
```

A range of valid values can be specified for numerical ConVars using a different constructor. Then a ConVar is automatically checked by the console system whenever changed manually. If the entered number is out of range, it's rounded to the next valid value. Setting valid range from 1 to 100:

```
    ConVar  my_variable(  "my_variable",  "42",  0,  "helptext",  true,  1,
 true, 100 );
```

Sometimes you also want a notification when a user's or another subsystem changes your ConVar value, therefore a callback function can be installed:

```
    static void OnChangeMyVariable ( ConVar *var, char const *pOldString
 )
    {
        DevMsg( "ConVar %s was changed from %s to %s\n", var->GetName(),
 pOldString, var->GetString() );
    }

    ConVar  my_variable(  "my_variable",  "42",  0,  "My  favorite  number",
 OnChangeMyVariable );
```

Using the class ConCommand

The class ConCommand is simpler than the ConVar and has just one constructor:

```
    ConCommand( char const *pName,
        FnCommandCallback callback,
        char const *pHelpString = 0,
        int flags = 0,
        FnCommandCompletionCallback completionFunc = 0 );
```

As in ConVar pName specifies the command name (no spaces!). callback is the function executed when a user runs this command, pHelpString and flags have the same function as in ConVar. ConCommands supports auto completion for the first parameter, which is usefully especially for commands that process files. For example, if you have a command loadtext lt;textfilegt; that expects a .txt file as input, the console scans for all available .txt files and allows the user to choose one from a list. If a valid completionFunc is passed, it will be called whenever the console system needs a list of available arguments.

When the callback function is executed, the parameters entered in console are not passed as function arguments. The callback function has to query the engine how many arguments where given using the engine interface function Cmd_Argc(). Then you can look at single arguments using Cmd_Argv(index), where index 1 is the first argument. The arguments are always returned as strings.

```
    void MySay_f ( void )
    {
        if ( engine->Cmd_Argc() < 1 )
        {
            Msg(""Usage: my_say <text>\n");
            return;
        }

        Msg("I say: %s\n", engine->Cmd_Argv(1) );
    }

    ConCommand my_say( "my_say", MySay_f, "say something", 0);
```

Here an example how to build a simple auto complete list. The partial parameter isn't used here; it contains the characters entered so far (including the command name itself) :

```
    static int MySayAutoComplete ( char const *partial,
    char         commands[         COMMAND_COMPLETION_MAXITEMS         ][
COMMAND_COMPLETION_ITEM_LENGTH ] )
    {
        strcpy( commands[0], "hello" );
        strcpy( commands[1], "goodbye" );
        return 2; // number of entries
    }

    ConCommand   my_say(   "my_say",   MySay_f,   "say   something",   0,
MySayAutoComplete);
```

The FCVAR_ flags

The console command/variable flags can specify quite powerful characteristics and must be handled with care. These flags are usually set in the constructor but may be modified with ConCommandBase::AddFlags() (not used very often). It's not possible to change these flags other than in source code to avoid cheating. Some flags must be set manually, others are set automatically by then console system:

```
    FCVAR_LAUNCHER,             FCVAR_GAMEDLL,             FCVAR_CLIENTDLL,
FCVAR_MATERIAL_SYSTEM, FCVAR_STUDIORENDER
```

These flags are set by the registration process and specify the module, where the command was created (you don't need to set them). The following flags must be set manually:

```
    FCVAR_CHEAT
```

Most commands and variables are for debugging proposes and not removed in release builds since they are useful 3rd party developers and map makers too. Unfortunately we cannot allow normal players to use these debugging tools since it's an unfair advantage over other players (cheating). A good rule is to add FCVAR_CHEAT basically to every new console command you add unless it's an explicit and legitimate options setting for players. Experience shows that even the most harmless looking debugging command can be misused as a cheat somehow.

The game server's setting of sv_cheats decides if cheats are enabled or not. If a client connects to a server where cheats are disabled (should be the default case), all client side console variables labeled as FCVAR_CHEAT are reverted to their default values and can't be changed as long as the client stays connected. Console commands marked as FCVAR_CHEAT can't be executed either.

```
    FCVAR_USERINFO
```

Some console variables contain client information the server needs to know about, like the player's name or his network settings. These variables must be flagged as FCVAR_USERINFO, so they get transmitted to the server and updated every time the user changes them. When the player changes on of these variables the engine notifies the server code via ClientSettingsChanged(). Then the game server can query the engine for specific client settings with GetClientConVarValue().

```
FCVAR_REPLICATED
```

Game server and client are using shared code where it's important that both sides run the exact some path using the same data (e.g. predicted movement/weapons, game rules). If this code uses console variables, they must match the same values on both sides. The flag FCVAR_REPLICATED ensures that by broadcasting these values to all clients. While connected, clients can't change these values and are force to use the server side values.

```
FCVAR_ARCHIVE
```

Some console variables contain user specific settings we want to restore each time the game is started (like name or network_rate). If a console variable is labeled as FCVAR_ARCHIVE, it is saved in the file config.cfg when the game shuts down and is reloaded with the next start. (Also the command host_writeconfig stores all FCVAR_ARCHIVE variables to a file).

```
FCVAR_NOTIFY
```

If a console variable is flagged as FCVAR_NOTIFY, a server sends a notification message to all clients whenever this variable is changed. This should be used for variables that change game play rules, which are important for all players (mp_friendlyfire etc).

```
FCVAR_PROTECTED
```

If console variables contain private information (passwords etc), we don't want them to be visible to other players. Then the FCVAR_PROTECTED flag should be set to label this information as confidential.

```
FCVAR_SPONLY
```

Sometimes executing commands or changing variables may be valid only in single player mode, then label these commands as FCVAR_SPONLY.

```
FCVAR_PRINTABLEONLY
```

Some important variables are logged or broadcasted (gamerules etc) and it is important that they contain only printable characters (no control chars etc).

```
FCVAR_NEVER_AS_STRING
```

The flag tells the engine never to print this variable as string since it contains control sequences.

```
FCVAR_DEMO
```

When starting recording a demo file, some console variables must explicitly added to the recording to ensure a correct playback.

```
FCVAR_DONTRECORD
```

This is the opposite of FCVAR_DEMO, some console commands shouldn't be recorded in demo files.

2.4 The Force Game Data (FGD)

FGD stands for Forge Game Data. It is the file extension for Hammer's game definition files. They define all of the *entities* of a game so mappers can select them from within the editor. It is a key point to understand that an FGD is nothing more than a reference. You cannot create or modify entities by editing a FGD, you merely give Hammer different information about what it expects to find within the game. Sometime editing reveals hidden or unused features or even entities, but they were always there and could be used even without the updated FGD.

History

While Hammer was originally called Worldcraft, it was developed under the name The Forge (hence the name Forge Game Data). However, due to trademark issues, the name Forge couldn't be used for the final version of Hammer. Still, the file extension stayed.

File format

The FGD file follows a fairly simple format. It is a script file that sets up entity structures and relationships for Hammer. The various parts of a game FGD (found in your SDK binary directory, for example: (path_to_steam)/SteamApps/SteamUsername/sourcesdk/bin) are explained below.

```
//====== Copyright © 1996-2005, Valve Corporation, All rights reserved. =======
//
// Purpose: Test game definition file (.fgd)
//
//=============================================================================
```

Comments are defined simply by starting a line with //. They can be preceded by spaces or tabs.

```
@include "base.fgd"
```

If the game you are writing your FGD for has a lot in common with another game (Half-Life 2 and Counter-Strike: Source, for example), you can include a file that has all of the common structures defined in it. The FGDs for Half-Life 2 and Counter-Strike: Source both include the base.fgd file, and the FGD for Half-Life 2 Deathmatch includes the halflife2.fgd file.

```
@BaseClass base(BaseNPC) = TalkNPC
[
	UseSentence(string) : "Use Sentence"
	UnUseSentence(string) : "Un-Use Sentence"
]
```

A BaseClass is used to setup structures that are used by several different entities. They are referenced in an entity structure by adding base(BaseClassName) to the main definition line of the structure. The BaseClass structure is defined just like a normal entity, in all respects. The only difference is that it doesn't appear in the entity lists in Hammer. (We'll discuss the complete entity structure below).

```
@PointClass base(Targetname, Origin) = example_entity : "example"
[
	spawnflags(flags) =
	[
		32 : "A flag"
		64 : "Another flag" : 1
	]

	foobarname(string) : "Name" : : "Name of foobar"
	foobargroup(string) : "Group" : "Squad1" : "Name of foobar group"
	foo(float) "Floating point number" : "100.7" : "Decimal points = fun"
```

```
          something(integer) : "first number" : 0 : "This is a number"
          something2(choices) :  "second  number" :  0 :  "Your  choice  of
numbers!" =
          [
                0 : "Default"
                1 : "Something"
                2 : "Another Thing"
          ]

          // Outputs
          output   OnSomethingHappened(void) :   "Fires   when   something
happens"
          output  OnSomethingElse(void) :  "Fires  when  something  else
happens"

          // Inputs
          input DoSomething(void) : "Do something"
      ]
```

Above is a generic example of an entity structure as defined in the FGD. Let's break it down bit by bit, starting with the first line:
@PointClass - The class type of an entity tells Hammer how this entity can be placed.
@PointClass - This entity exists at a certain non-arbitrary point. It is typically referred to as a "point entity". The entities are placed within Hammer by using the Entity tool (Shift+E).
@NPCClass - This is a special form of point entity tailored for NPC (non-player character) entities. It is useful in conjunction with the npcclass property type, below.
@SolidClass - This entity's area is defined by the solid (also referred to as a brush) that it is attached to. It is typically referred to as a "brush entity" or "solid entity".
base(Targetname, Origin) - Things between the type declaration and the "=" character help to define properties of the entity and how it will act and be displayed in Hammer. There are a number of different things that can used here. (More than one of these can be used, each separated by a space.)
base(BaseClass1, BaseClass2, ...) - This lets you attach previously defined BaseClasses (see above) to an entity. You can specify multiple BaseClasses, separated by a comma.
color(rrr ggg bbb) This setting will change the color of the wireframe box in the Hammer 2D views. If this isn't present, the color will default to magenta. The values specified here are the RGB values of a color, and each number has a range from 0 to 255.
iconsprite("path/sprite.vmt") - If this is used, the specified sprite will be shown in the Hammer 3D view instead of a flat-shaded colored box.
sphere(propertyname) - If an entity has a radius of effect, like a sound for example, a sphere will be displayed in Hammer's 2D and 3D views. You need to specify the property that will control the sphere size. If no property is specified, it will look for a radius property. Multiple spheres can be defined for one entity.
studio("path/model.mdl) - If this is used, the entity will be displayed in the 3D view as the specified model. If no model is specified, the value of the entity's "model" property will be used, if available.
studioprop("path/model.mdl) - Similar to studio(), but prop specific.
example_entity : "example" - This is the entity's name followed by a description. The description is displayed in Hammer when you click on the Help button inside the entity property dialog. For visual ease, the description can span multiple lines by joining "blocks of text" with the plus (+) character. For example:

```
      @PointClass = example_entity :
```

```
"This is an example description for"+
"this example entity. It will appear"+
" in the help dialog for this entity"
[
entity properties go here
]
```

entity properties - Everything between the main set of [/] brackets is used to define the entity's properties, including their inputs and outputs. Individual property structures consist of a name, a type declaration, a short description, a default value, and a long description. The most common properties are:
string - This creates a property of the string type.
name(string) : "Short description" : "Default" : "Long description"
integer - This creates a property of the integer type.
name(integer) : "Short description" : 1 : "Long description"
float - This creates a property of the float type. Although it deals with numbers, the structure of it is similar to the string type. The default value must have quotes around it.
name(float) : "Short description" : "1.5" : "Long description"
There are also two common special case property types, choices and flags, that follow a slightly different format.
choices - A property of this type lets you setup a number of distinct choices. Their format is similar to the other types:

```
    name(choices) :       "Short       description" :       "1"       =
[
    0 : "something"
    1 : "something else (default)"
    2 : "something completely different"
    ]
    You can also use strings (or floats) as values, instead of integers,
like this:
    name(choices) :  "Short  description" :  "models/something02.mdl"  =
[
    "models/something01.mdl" : "something"
    "models/something02.mdl" : "something else (default)"
    "models/something03.mdl" : "something completely different"
    ]
```

flags - The flags property type lets you setup what will appear in the Flags portion of the entity property dialog. It is setup similar to the choices property type. The flags are all powers of 2 - 20, 21, 22, etc, and their values are either 0 (off) or 1 (on). If no default is specified for a flag, it is considered to be off.

```
    spawnflags(flags)                                                    =
[
    1 : "something clever" : 1
    2 : "something else" : 0
    4 : "you said what now?" : 0
    8 : "nothing" : 1
    ]
```

Note that spawnflags is always the name of this property.
There are also a number of special purpose property types that modify the entity properties dialog UI to allow for easy browsing for files or easier manipulation of complex properties (like colors or angles).
axis - adds a relative 2 point axis helper
angle - adds an angle widget for this property to the entity dialog UI.

color255 - adds a button that brings up the color choosing UI, which takes the color you select and translated it into the three number RGB value. Allows extra parameters (i.e. brightness)
color1 - adds a color button, but uses a float [0,1] to represent RGB. Allows extra parameters (i.e. brightness)
filterclass - marks property as being the name of the filter to use
material - adds a button that brings up the material browser.
node_dest - adds an eyedropper to select a node in the 3d view
npcclass - adds a drop-down selection list populated by entities of the NPCClass type.
origin - origin
pointentityclass - adds a drop-down selection list populated by entities of the PointClass type.
scene - adds a button that brings up the file browser, letting you browse for scene files.
sidelist - adds a side selection eyedropper that allows you to choose sides (multiple with ctrl).
sound - adds a button that brings up the file browser, letting you browse for sound files.
sprite - adds a button that brings up the file browser, letting you browse for sprite files. It is advised that you use the material datatype instead; it has the same functionality, but it uses the material browser.
studio - adds a button that brings up the file browser, letting you browse for model files.
target_destination - marks property as another entity's targetname.
target_name_or_class - marks property as another entity's targetname or classname.
target_source - marks property as being the name that other entities may target.
vecline - adds an absolute 1 point axis helper
vector - 3d vector

1.5 Entities

One of the easiest ways to begin your scripting/mod is by creating new entities inside the Source engine. Entities make up the objects in the world: NPCs, weapons, tin cans, or triggers; they're the means by which we add life and interaction into the static geometry of the world. Entities fall into three general categories: logical, model, and brush.

Logical Entity

These entities are the simplest of entities, because they have no position in the world, no visual component, and only exist to service input from the game map and make decisions based on the state of the world. Logical entities do not move and they do not have a model, they simply receive inputs and send outputs depending on their exact utility. An example would be a logic_counter entity that stores a value that can be added to or subtracted from. Other entities in the map can access the entity via inputs or get information from it via outputs. The entity's position is irrelevant in this case, which is the major factor in declaring an entity logical.

Model Entity

Model entities are what we would most typically think of entities as being: they have a visual component, they can move around the map and often they are interactive. NPCs are an example of this sort of entity.

Brush Entity

These entities are constructed in Hammer out of *brushes*. Brush entities are most often *triggers:* volumes in space that fire outputs when certain other entities (like the player) enter or exit them. These can also be moving entities, like doors and platforms.
Using these three major types of entities, we can express almost all the flora and fauna of the Source universe.

Authoring a Logical Entity

Create a file named sdk_logicalentity.cpp. The file should go under the dlls folder under your source code folder. For example, if you installed the source code into C:\MyMod\src, then you would create a file called C:\MyMod\src\dlls\sdk_logicalentity.cpp.
Next, copy *Logical Entity code* and paste it into this new file.
Last, add this file to your server.dll project.

Walking Through The code

Class Definition

```
class CMyLogicalEntity : public CLogicalEntity
{
public:
     DECLARE_CLASS( CMyLogicalEntity, CLogicalEntity );
};
```

We descend our new entity from the CLogicalEntity class. This class is a server-side only entity and will not transmit data to the client. We also use the helper macro DECLARE_CLASS to assist in some behind the scenes bookkeeping. By declaring the relationship between CMyLogicalEntity and CLogicalEntity we are able to use the BaseClass type within this class. This will be useful later for calling up to the CLogicalEntity class we're descended from.

Linking the class to an entity name

Next we'll link the CMyLogicalEntity class to an actual entity classname that the game engine can reference.

```
LINK_ENTITY_TO_CLASS( my_logical_entity, CMyLogicalEntity );
```

Here our class CMyLogicalEntity declares its *classname* as "my_logical_entity". This is the name that Hammer will use to refer to the entity type in the editor and is also how other entities will search and find our entity type. The classname varies from the targetname of the entity, which is a string that labels an individual or group of entities. The classname refers to all entities of a type, while the targetname can span multiple different types of entities (i.e. You may have an entity with a classname of env_splash belonging to a group of entities, all with the targetname of splash_group).

Adding member variables

```
int     m_nThreshold;  // Count at which to fire our output
int     m_nCounter;    // Internal counter
```

Here we declare two integer values, to be used later.

Declaring a data description for the entity

```
 . . .

public:
       DECLARE_CLASS( CMyLogicalEntity, CLogicalEntity);
       DECLARE_DATADESC();

 . . .

LINK_ENTITY_TO_CLASS( my_logical_entity, CMyLogicalEntity );
BEGIN_DATADESC( CMyLogicalEntity )
```

```
DEFINE_FIELD( m_nCounter, FIELD_INTEGER ),
DEFINE_KEYFIELD( m_nThreshold, FIELD_INTEGER, "threshold" ),

END_DATADESC()
```

The DECLARE_DATADESC macro must be included to let the compiler know that we'll be adding a data description table later in the class implementation. The data description holds various definitions for the data members and special functions for this class. In this case, m_nCounter is defined for save/load functionality, and m_nThreshold is defined to tell the game to use the value named "threshold" to link this member variable to the entity keyvalue from Hammer. See the Data Descriptions Table Document for more information.

Creating the output event

```
COutputEvent   m_OnThreshold;
DEFINE_OUTPUT( m_OnThreshold, "OnThreshold" ),
```

This output event will be triggered when we meet the defined threshold. For more information on outputs, see Entity Inputs and Outputs.

Creating the input function

```
void InputTick( inputdata_t &inputData );

void CMyLogicalEntity::InputTick( inputdata_t &inputData )
{
	// Increment our counter
	m_nCounter++;

	// See if we've met or crossed our threshold value
	if ( m_nCounter >= m_nThreshold )
	{
		// Fire an output event
		m_OnThreshold.FireOutput( inputData.pActivator, this );

		// Reset our counter
		m_nCounter = 0;
	}
}
```

This function simply increments a counter and fires an output when that counter reaches or exceeds a certain threshold value, as specified in the entity inside of Hammer. The function takes no parameters from Hammer.

Create The FGD Entry

To use the entity within Hammer, we'll need to create an entry in our FGD file. Hammer will use this data to interpret the various keyvalues and functions the entity is exposing. See the FGD Format document for more information about FGD files.

In this case we declare the "threshold" value we have linked to the m_nThreshold data member, the input function Tick and the OnThreshold output function.

```
    @PointClass base(Targetname) = my_logical_entity : "Tutorial logical entity."
    [
           threshold(integer) : "Threshold" : 1 : "Threshold value."
           input Tick(void) : "Adds one tick to the entity's count."
           output OnThreshold(void) : "Threshold was hit."
    ]
```

If your .FGD is otherwise empty, be sure to add the line @include "base.fgd", which provides Hammer with some necessary functions.(That is appropriate for a total conversion. For a mod based on existing content, include the appropriate FGD instead; eg, for an HL2 mod, include halflife2.fgd instead of base.fgd.)

2.6 Authoring a Model Entity

Create a file named sdk_modelentity.cpp. The file should go under the dlls folder under your source code folder. For example, if you installed the source code into C:\MyMod\src, then you would create a file called C:\MyMod\src\dlls\sdk_modelentity.cpp
Next, copy the *Model Entity code* and paste it into this new file.
Last, add this file to your server.dll project.

Walking Through The code

Creating The Class Definition

```
class CMyModelEntity : public CBaseAnimating
{
public:
     DECLARE_CLASS( CMyModelEntity, CBaseAnimating );
     DECLARE_DATADESC();

     void Spawn( void );
     void Precache( void );
     void MoveThink( void );

     // Input function
     void InputToggle( inputdata_t &inputData );

private:
     bool         m_bActive;
     float        m_flNextChangeTime;
};
```

We descend our new entity from the CBaseAnimating class. This allows us to use models and animate. Also new to this entity is the Spawn() and Precache() function.

Defining The Data Description

```
LINK_ENTITY_TO_CLASS( my_model_entity, CMyModelEntity );

// Start of our data description for the class
BEGIN_DATADESC( CMyModelEntity )

     // Save/restore our active state
```

```
          DEFINE_FIELD( m_bActive, FIELD_BOOLEAN ),
          DEFINE_FIELD( m_flNextChangeTime, FIELD_TIME ),

          // Links our input name from Hammer to our input member function
          DEFINE_INPUTFUNC( FIELD_VOID, "Toggle", InputToggle ),

          // Declare our think function
          DEFINE_THINKFUNC( MoveThink ),

     END_DATADESC()
```

Much like our logical entity, we must declare the variables used by the entity so that the engine knows their intention.
It's important to note that the MoveThink() function must be declared as an entity think function in the entity's data description table using the DEFINE_THINKFUNC macro.

Creating The Precache() Function

```
#define ENTITY_MODEL     "models/gibs/airboat_broken_engine.mdl"

void CMyModelEntity::Precache( void )
{
     PrecacheModel( ENTITY_MODEL );
}
```

The Precache() function is where all asset precaching must be done.
In this example, we call PrecacheModel() to precache our model. Without this step the entity's model would not appear in the world and the engine would complain of a missed precache.

Creating The Spawn() Function

```
void CMyModelEntity::Spawn( void )
{
       Precache();

       SetModel( ENTITY_MODEL );
       SetSolid( SOLID_BBOX );
       UTIL_SetSize( this, -Vector(20,20,20), Vector(20,20,20) );

       m_bActive = false;
}
```

The Spawn() function is called after the entity is first created. This function can be thought of as the game's constructor method for the entity. In this function the entity can setup its initial state, including what model to use for itself, its method of movement and its solidity. It's important to note that the Spawn() function is called immediately after the allocation of the entity and that if this has occurred at the beginning of a map, there is no guarantee that other entities have been spawned yet. Therefore, any code which requires the entity to search or otherwise link itself to other named entities must do so in the Activate() function of the entity. Activate() is called when all entities have been spawned and had their Spawn() function called. Searching for entities before the Activate() function will rely on the spawning order of the entities and is unreliable.

In this example, we first call the Precache() function to be sure all of our assets are precached properly. After that, we use the SetModel() function to set our entity's model to the one we defined previously.

Next, we set the solidity of the entity via the SetSolid() function. There are multiple possible solid types, defined as:

SOLID_NONE	Not solid.
SOLID_BSP	Uses the BSP tree to determine solidity (used for brush models)
SOLID_BBOX	Uses an axis-aligned bounding box.
SOLID_CUSTOM	Entity defines its own functions for testing collisions.
SOLID_VPHYSICS	Uses the vcollide object for the entity to test collisions.

For this example, we'll make the entity use a bounding box. The UTIL_SetSize() function allows us to set the size of that bounding box. Here we set it to a 40x40x40 cube.

Creating The MoveThink() Function

Entities have the ability to update internal state and make decisions via a think function, which will be called at a rate specified by the entity. Here we will create a think function that we will have called up to 20 times a second. We will use this think function to randomly update our movement and direction in the world.

```
void CMyModelEntity::MoveThink( void )
{
     // See if we should change direction again
     if ( m_flNextChangeTime < gpGlobals->curtime )
     {
            // Randomly take a new direction and speed
            Vector vecNewVelocity = RandomVector( -64.0f, 64.0f );
            SetAbsVelocity( vecNewVelocity );

            // Randomly change it again within one to three seconds
            m_flNextChangeTime    =    gpGlobals->curtime    +    random-
>RandomFloat(1.0f,3.0f);
     }

     // Snap our facing to where we are heading
     Vector velFacing = GetAbsVelocity();
     QAngle angFacing;
     VectorAngles( velFacing, angFacing );
     SetAbsAngles( angFacing );

     // Think every 20Hz
     SetNextThink( gpGlobals->curtime + 0.05f );
}
```

While a lot of code is packed into this function, its outcome is fairly simple: once a random time interval has elapsed, the entity will choose a new, random direction and speed to travel at. It will also update its angles to face towards this direction of travel.

The call to SetNextThink() is important in this function, because it tells the entity when next to think. Here it is set to think again 1/20th of a second in the future. Most entities will only need to think at a rate of 1/10th of a second, depending on their behaviors. It's important to note that failure to update the next think time of the entity will cause it to stop thinking (which is sometimes desired).

Create the InputToggle() function

For this entity, we'll use an input to toggle its movement on and off. To do so, we declare the input function like any other.

```
void CMyModelEntity::InputToggle( inputdata_t &inputData )
{
	// Toggle our active state
	if ( !m_bActive )
	{
		// Start thinking
		SetThink( MoveThink );
		SetNextThink( gpGlobals->curtime + 0.05f );

		// Start flying
		SetMoveType( MOVETYPE_FLY );
		// Set our next time for changing our speed and
direction
		m_flNextChangeTime = gpGlobals->curtime;
		m_bActive = true;
	}
	else
	{
		// Stop thinking
		SetThink( NULL );

		// Stop moving
		SetAbsVelocity( vec3_origin );
		SetMoveType( MOVETYPE_NONE );

		m_bActive = false;
	}
}
```

To start the entity thinking, we use the SetThink() function in conjunction with the SetNextThink() function. This tells the entity to use our MoveThink() function and to call it 1/20th of a second in the future. It's important to note that an entity can have any number of think functions and use the SetThink() function to choose between them. Entities can even have multiple think functions running at the same time using the SetContextThink() covered in another document.

We also set the entity's movement type to MOVETYPE_FLY. This allows the entity to move along a direction without gravity.

In the second portion of this function we stop the entity from moving. The think function is set to NULL to stop all thinking. Its movement type is also set to MOVETYPE_NONE to keep it from moving.

Create The FGD Entry

To use the entity within Hammer, we'll need to create an entry in our FGD file. Hammer will use this data to interpret the various keyvalues and functions the entity is exposing. The FGD entry for this entity simply displays the model in hammer and allows you to send the input "Toggle" to it.

```
      @PointClass                                          base(Targetname)
   studio("models/gibs/airboat_broken_engine.mdl")=       my_model_entity :
   "Tutorial model entity."
      [
              input Toggle(void) : "Toggle movement."
      ]
```

Be sure to add the line @include "base.fgd" at the top of the FGD file, which provides Hammer with some necessary functions. (That is appropriate for a total conversion. For a mod based on existing content, include the appropriate FGD instead; eg, for an *HL2* mod, include halflife2.fgd instead of base.fgd.)

2.7 Authoring a Brush Entity

Create a file named sdk_brushentity.cpp. The file should go under the dlls folder under your source code folder. For example, if you installed the source code into C:\MyMod\src, then you would create a file called C:\MyMod\src\dlls\sdk\sdk_brushentity.cpp.
Next, copy *Authoring Brush Entity Code* and paste it into this new file.
Last, add this file to your server.dll project. If you opened the game_sdk.sln solution, then you can right-click on the hl project in the Solution Explorer window and choose Add, then Add Existing Item.

Walking through the code

Creating the Class Definition

```
class CMyBrushEntity : public CBaseToggle
{
public:
       DECLARE_CLASS( CMyBrushEntity, CBaseToggle );
       DECLARE_DATADESC();

       void Spawn( void );
       bool CreateVPhysics( void );

       void BrushTouch( CBaseEntity *pOther );
};
```

We descend our new entity from the CBaseToggle class. This class has some basic functions to help us move our brush entity through the world.

Defining the Data Description

```
LINK_ENTITY_TO_CLASS( my_brush_entity, CMyBrushEntity );

// Start of our data description for the class
BEGIN_DATADESC( CMyBrushEntity )

     // Declare this function as being a touch function
     DEFINE_ENTITYFUNC( BrushTouch ),
	END_DATADESC()
```

Here we simply declare our touch function that we'll use.

Create the Spawn() function

```
	void CMyBrushEntity::Spawn( void )
	{
		// We want to capture touches from other entities
		SetTouch( &CMyBrushEntity::BrushTouch );

		// We should collide with physics
		SetSolid( SOLID_VPHYSICS );

		// We push things out of our way
		SetMoveType( MOVETYPE_PUSH );

		// Use our brushmodel
		SetModel( STRING( GetModelName() ) );

		// Create our physics hull information
		CreateVPhysics();
	}
```

The first thing we do in this block is setup our touch function to point to BrushTouch() where we'll do our movement code. Next we tell the entity to use SOLID_VPHYSICS so we'll use our exact bounds to collide. Setting the entity to MOVETYPE_PUSH means that we'll attempt to move entities out of our way, instead of just being blocked.
In this example we use the SetModel() with our model name from the editor. In this case it tells the entity to use its brush model, as defined in the map.

```
	bool CMyBrushEntity::CreateVPhysics( void )
	{
		// For collisions with physics objects
		VPhysicsInitShadow( false, false );

		return true;
	}
```

Finally, we call CreateVPhysics() to setup our collision shadow. This is what we'll use to collide with physics objects in the world. Without this, the brush would pass through those objects.

Create the BrushTouch() function

The entity has been told to notify us when its been touched, via the BrushTouch() function. When we receive this notification, we'll cause the entity to move away from the entity that touched it. To do this, we'll need information about the events surrounding the touch. This information is provided in the trace_t structure, returned by the GetTouchTrace() function. This returns the actual trace collision that generated the event.

```
	void CMyBrushEntity::BrushTouch( CBaseEntity *pOther )
	{
        // Get the collision information
        const trace_t &tr = GetTouchTrace();

        // We want to move away from the impact point along our surface
        Vector vecPushDir = tr.plane.normal;
        vecPushDir.Negate();
        vecPushDir.z = 0.0f;

        // Move slowly in that direction
        LinearMove( GetAbsOrigin() + ( vecPushDir * 64.0f ), 32.0f );
    }
```

First we retrieve the normal of the surface that was hit. In our case, this will be one of the planes of the brush entity. We negate that value to point towards the direction of the impact, and then remove the Z component of the direction to keep us parallel to the floor.
Finally, we use the LinearMove() function to cause the brush to move to a location at a given speed. The LinearMove() function is implemented by CBaseToggle and takes care of behind-the-scenes maintenance in how the brush model moves.

Create The FGD Entry

To use the entity within Hammer, we'll need to create an entry in our FGD file. Hammer will use this data to interpret the various keyvalues and functions the entity is exposing. The FGD entry allows you to assign the entity to a brush.

```
    @SolidClass  base(Targetname)  =  my_brush_entity :  "Tutorial  brush
entity."
    [

    ]
```

Be sure to add the line @include "base.fgd" at the top of the FGD file, which provides Hammer with some necessary functions.

2.8 Sample Code

```
    Logical Entity Code

    //=====  Copyright  ©  1996-2005,  Valve  Corporation,  All  rights
reserved. ========
    //
    // Purpose: Simple logical entity that counts up to a threshold
value, then
    //               fires an output when reached.
    //
    //=================================================================
==========

    #include "cbase.h"

    class CMyLogicalEntity : public CLogicalEntity
```

```
   {
   public:
         DECLARE_CLASS( CMyLogicalEntity , CLogicalEntity );
         DECLARE_DATADESC();

         // Constructor
         CMyLogicalEntity ( void ) : m_nCounter( 0 ) {}

         // Input function
         void InputTick( inputdata_t &inputData );

   private:

         int                    m_nThreshold;     // Count at which to
fire our output
         int                    m_nCounter;       // Internal counter

         COutputEvent     m_OnThreshold;    // Output even when the
counter reaches the threshold
   };

   LINK_ENTITY_TO_CLASS( my_logical_entity, CMyLogicalEntity  );

   // Start of our data description for the class
   BEGIN_DATADESC( CMyLogicalEntity  )

         // For save/load
         DEFINE_FIELD( m_nCounter, FIELD_INTEGER ),

         // Links our member variable to our keyvalue from Hammer
         DEFINE_KEYFIELD( m_nThreshold, FIELD_INTEGER, "threshold" ),

         // Links our input name from Hammer to our input member
function
         DEFINE_INPUTFUNC( FIELD_VOID, "Tick", InputTick ),

         // Links our output member to the output name used by Hammer
         DEFINE_OUTPUT( m_OnThreshold, "OnThreshold" ),

   END_DATADESC()

   //-----------------------------------------------------------------
----------
   // Purpose: Handle a tick input from another entity
   //-----------------------------------------------------------------
----------
   void CMyLogicalEntity ::InputTick( inputdata_t &inputData )
   {
         // Increment our counter
         m_nCounter++;
```

```
	// See if we've met or crossed our threshold value
	if ( m_nCounter >= m_nThreshold )
	{
		// Fire an output event
		m_OnThreshold.FireOutput( inputData.pActivator, this );

		// Reset our counter
		m_nCounter = 0;
	}
}
```

Model Entity Code

```
//===== Copyright © 1996-2005, Valve Corporation, All rights reserved. ========
//
// Purpose: Simple model entity that randomly moves and changes direction
//		when activated.
//
//==========================================================================

#include "cbase.h"

class CMyModelEntity : public CBaseAnimating
{
public:
	DECLARE_CLASS( CMyModelEntity, CBaseAnimating );
	DECLARE_DATADESC();

	void Spawn( void );
	void Precache( void );

	void MoveThink( void );

	// Input function
	void InputToggle( inputdata_t &inputData );

private:

	bool	m_bActive;
	float m_flNextChangeTime;
};

LINK_ENTITY_TO_CLASS( my_model_entity, CMyModelEntity );

// Start of our data description for the class
```

```
   BEGIN_DATADESC( CMyModelEntity )

         // Save/restore our active state
         DEFINE_FIELD( m_bActive, FIELD_BOOLEAN ),
         DEFINE_FIELD( m_flNextChangeTime, FIELD_TIME ),

         //  Links  our  input  name  from  Hammer  to  our  input  member
function
         DEFINE_INPUTFUNC( FIELD_VOID, "Toggle", InputToggle ),

         // Declare our think function
         DEFINE_THINKFUNC( MoveThink ),

   END_DATADESC()

   // Name of our entity's model
   #define     ENTITY_MODEL      "models/gibs/airboat_broken_engine.mdl"

   //-----------------------------------------------------------------
----------
   // Purpose: Precache assets used by the entity
   //-----------------------------------------------------------------
----------
   void CMyModelEntity::Precache( void )
   {
         PrecacheModel( ENTITY_MODEL );
   }

   //-----------------------------------------------------------------
----------
   // Purpose: Sets up the entity's initial state
   //-----------------------------------------------------------------
----------
   void CMyModelEntity::Spawn( void )
   {
         Precache();

         SetModel( ENTITY_MODEL );
         SetSolid( SOLID_BBOX );
         UTIL_SetSize( this, -Vector(20,20,20), Vector(20,20,20) );

         m_bActive = false;
   }

   //-----------------------------------------------------------------
----------
   // Purpose: Think function to randomly move the entity
   //-----------------------------------------------------------------
----------
   void CMyModelEntity::MoveThink( void )
```

```
{
	// See if we should change direction again
	if ( m_flNextChangeTime < gpGlobals->curtime )
	{
		// Randomly take a new direction and speed
		Vector vecNewVelocity = RandomVector( -64.0f, 64.0f );
		SetAbsVelocity( vecNewVelocity );

		// Randomly change it again within one to three seconds
		m_flNextChangeTime   =   gpGlobals->curtime   +   random-
>RandomFloat( 1.0f, 3.0f );
	}

	// Snap our facing to where we're heading
	Vector velFacing = GetAbsVelocity();
	QAngle angFacing;
	VectorAngles( velFacing, angFacing );
	SetAbsAngles( angFacing );

	// Think every 20Hz
	SetNextThink( gpGlobals->curtime + 0.05f );
}

//-----------------------------------------------------------------
----------
// Purpose: Toggle the movement of the entity
//-----------------------------------------------------------------
----------
void CMyModelEntity::InputToggle( inputdata_t &inputData )
{
	// Toggle our active state
	if ( !m_bActive )
	{
		// Start thinking
		SetThink( MoveThink );
		SetNextThink( gpGlobals->curtime + 0.05f );

		// Start flying
		SetMoveType( MOVETYPE_FLY );

		// Set our next time for changing our speed and direction
		m_flNextChangeTime = gpGlobals->curtime;
		m_bActive = true;
	}
	else
	{
		// Stop thinking
		SetThink( NULL );
```

```
            // Stop moving
            SetAbsVelocity( vec3_origin );
            SetMoveType( MOVETYPE_NONE );

            m_bActive = false;
      }
}
```

Authoring Bush Entity Code

```
//===== Copyright © 1996-2005, Valve Corporation, All rights reserved. ========
//
// Purpose: Simple brush entity that moves when touched
//
//=========================================================================

#include "cbase.h"

class CMyBrushEntity : public CBaseToggle
{
public:
      DECLARE_CLASS( CMyBrushEntity, CBaseToggle );
      DECLARE_DATADESC();

      void Spawn( void );
      bool CreateVPhysics( void );

      void BrushTouch( CBaseEntity *pOther );
};

LINK_ENTITY_TO_CLASS( my_brush_entity, CMyBrushEntity );

// Start of our data description for the class
BEGIN_DATADESC( CMyBrushEntity )

      // Declare this function as being a touch function
      DEFINE_ENTITYFUNC( BrushTouch ),

END_DATADESC()

//-------------------------------------------------------------------------
// Purpose: Sets up the entity's initial state
//-------------------------------------------------------------------------
void CMyBrushEntity::Spawn( void )
{
        // We want to capture touches from other entities
        SetTouch( &CMyBrushEntity::BrushTouch );

        // We should collide with physics
        SetSolid( SOLID_VPHYSICS );

        // We push things out of our way
        SetMoveType( MOVETYPE_PUSH );

        // Use our brushmodel
        SetModel( STRING( GetModelName() ) );

        // Create our physics hull information
        CreateVPhysics();
   }

   //----------------------------------------------------------------------
----------
   // Purpose: Setup our physics information so we collide properly
   //----------------------------------------------------------------------
----------
   bool CMyBrushEntity::CreateVPhysics( void )
   {
        // For collisions with physics objects
        VPhysicsInitShadow( false, false );

        return true;
   }

   //----------------------------------------------------------------------
----------
   // Purpose: Move away from an entity that touched us
   // Input  : *pOther - the entity we touched
   //----------------------------------------------------------------------
----------
   void CMyBrushEntity::BrushTouch( CBaseEntity *pOther )
   {
        // Get the collision information
        const trace_t &tr = GetTouchTrace();

        // We want to move away from the impact point along our surface
        Vector      vecPushDir = tr.plane.normal;
        vecPushDir.Negate();
        vecPushDir.z = 0.0f;

        // Move slowly in that direction
        LinearMove( GetAbsOrigin() + ( vecPushDir * 64.0f ), 32.0f );
   }
```

3. UnrealEngine

3.1 Introduction

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++ or Java, is familiar with object-oriented programming, has played Unreal and has used the UnrealEd editing environment. 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-inheritance 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.

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 extends 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  &amp;amp;amp;&amp;amp;amp;  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 "extends" (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. These properties are designated using the var() syntax, instead of var.
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 that 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.

3.2 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 that 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 (the so called "state code"), not from code within a function (that includes functions define within a state). 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 that 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.

3.3 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 extension 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 (extends 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 (extends Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.
Class (extends 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 extends 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 inherits 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 extends 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.

```
nativereplication
```

Indicates that this class uses C++ code to replicate state across the network.

```
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 relevant 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".

`placeable`

Identifies this class as being available for placement in UnrealEd. Classes with this designation can be placed into the world (level).

`notplaceable`

Prevents this class from being placed in UnrealEd.

`hidecategories(category list)`

Tells UnrealEd to not display the specified property categories for editing within the property browser (has effect on subclasses).

`showcategories(category list)`

`opposite of hidecategories`

noexport

for native classes only, this will prevent the class from being exported automatically to the header file. The native class will have to be defined by hand in order to be used in native code.

`exportstructs`

all structs declared in this class will be exported to the header file. Without this only structs declared native export will be exported to the header file.

`CacheExempt`

this class will be exempt for exporting to a cache file.

`HideDropDown`

the class won't show up in various dropdown menus in UnrealEd (v3323 and up).

`collapsecategories`

All categories will be collapsed to a single category, this has effect on all subclasses in order to uncollapse the categories for a subclass use dontcollapsecategories

`editinlinenew`

Advanced. allows an instance of this object to be created inline within UnrealEd; instead of using a reference to an existing object a new instance can be created from within a property inspector. This setting also has effect on all subclasses, to negate this option in a subclass use noteditinlinenew

`instanced`

Advanced. This has effect on an object when it is defined as a subobject in the defaultproperties section. When this subobject is assigned to a variable in the defaultproperties section the subobject declaration will be used as a template to create a new instance of that object. This way the same subobject declaration can be used to for multiple instances. Right now this is only used for the GUI system. This setting has effect on all subclasses and it can't be negated. (v3323 and up)

`dependsOn(ClassName)`

this will make sure this class is compiled after ClassName, because this class depends on ClassName to be compiled first. This only works for classes defined in the same package. You can use multiple DependsOn(...) specifiers in a class declaration.

`within ClassName`

means that this object class (can not be used with Actor classes) resides within an other Class, the outer of this class will be assumed to be of type ClassName instead of object. This keyword is used by PlayerInput and CheatManager.

3.4 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 PlayerName; // Declare a max 32-character string.
var actor Other; // Declare a variable referencing an actor.
var() float MaxTargetDist; // Can be set in <nop>UnrealEd
```

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 declared with the local keyword, such as:

```
function int Foo()
{
   local int Count;
   Count = 1;
   return Count;
}
```

```
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 an index into the global name table. Names correspond to simple strings of up to 64 characters. Names are not like strings in that they are immutable once created (see UnrealStrings for more information).

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.

Variable specifiers

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:

config

This variable will be made configurable. The current value can be saved to the ini file and will be loaded when created.

globalconfig

Works just like config except that is has effect on all subclasses, unless configuration of the subclass overrides the value.

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).

editconst

Advanced. The variable can be seen in UnrealEd but not edited. A variable that is editconst is *not* implictly "const".

editconstarray

Advanced. Only usefull for dynamic arrays. This will prevent the user from changing the length of an array.

edfindable

Advanced. Allows this actor to be findable within the editor (using the "find" dialog).

editinline

Advanced. This object reference can be edited inline within UnrealEd's property inspector (only usefull for object references).

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.

protected

The variable can only be accessed from the class and it's subclasses, not from other classes.

travel

Advanced. "Travel" is supported only for the player's PlayerPawn derived class and Inventory-derived classes in his linked Inventory list. It means "this variable should be saved and restored when switching levels, so that it isn't forgotten". This is the mechanism that keeps your health and ammo counts intact when switching levels. The combination of "transient" and "travel" isn't sensible (and isn't used in Unreal), since transient objects are non persistent betweeen savegames, and "travel" implies the thing should persist between levels.

skip

Advanced. Only used for logical operators like && and ||.

export

Advanced. Only meaningful for the Brush variable in Actor classes.

localized

The value of this variable will have a localized value defined. Mostly used for strings. Read more about this in the LocalizationReference.

Arrays

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.

Editability

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".
```

You can also declare a variable as editconst, which means that the variable should be visible but *not* editable UnrealEd. Note that this only prevents the variable from being changed in the editor, not in script. If you want a variable that is truly const but still visible in the editor, you must declare it const editconst:

```
// MyBool is visible but not editable in UnrealEd
var(MyCategory) editconst bool MyBool;

// MyBool is visible but not editable in UnrealEd and
// not changeable in script
var(MyCategory) const editconst bool MyBool;

// MyBool is visible and can be set in UnrealEd but
// not changeable in script
var(MyCategory) const bool MyBool;
```

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 referring to an actor is the Enemy variable in the Pawn class, which refers to the actor that 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 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 (referred to as an "instance" of class C). To give a really shaky analogy, a class is like a pepper grinder, and an instance of that class 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).

When declaring variables that reference class objects, you can optionally use the syntax **class<metaclass>** to limit the classes that can be referenced by the varible to classes of type *metaclass* (and its child classes). For example, in the declaration:

var class<actor> ActorClass;

The variable ActorClass may only reference a class that extends 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:

```
    // casts the result of SomeFunctionCall() a class of type Actor (or
subclasses of Actor)
    class<actor>( SomeFunctionCall() )
```

3.5 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.
```

3.6 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 much like C structs, in that they can contain variables, arrays, and other structs, but UnrealScript structs cannot contain functions.

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.

3.7 Expressions

Constants

In UnrealScript, you can specify constant literal values for 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 hexadecimal 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: ect(1.0,2.0,4.0)
Rotation constants contain Pitch, Yaw, and Roll values like this: Rot(0x8000,0x4000,0)
The None constant refers to "no object" (or equivalently, "no actor").
The Self constant refers to "this object" (or equivalently, "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 static array, for example: ArrayCount(Touching)
You can use the "const" keyword to declare constants that 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 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, a "float"), and you specify a different type of data (for example, an "int), the compiler will try to automatically 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 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; True 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;

      // Cast Target to Pawn and assign the result to P.  If Target is
   not a Pawn (or subclass of Pawn), then the value assigned to P will be
   None.
      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 conversions 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 and allow you access the properties and methods declared in the more derived class.

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 (because Actor.Owner is a variable of type 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 );
}
```

3.8 Functions

Declaring Functions

In UnrealScript, you can declare new functions and write new versions of existing functions (overwrite 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", with the exception on formatting rules). 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 &amp;&amp; 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 &amp;&amp; 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. One important note about the out keyword (does not apply to natvie functions): the value isn't really passed by reference, but the changed value is copied back to the variable. So there is no speed gain in using the out keyword, in had the opposite effect (copy before and after the function call).

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, extends 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 extends an existing class. Then, all you need to do is override the

functions that 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 that 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 behavior, 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 implemented (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), and is only required for operator functions. 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 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.

Iterator

Declares that a 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 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 code similar to the following in EngineClasses.h:
void eventTouch(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;
SomeActor->eventTouch(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").

```
    function BreakExample()
    {
        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
```

Continue

The "continue" command will jump back to the beginning of the loop, so everything after the continue command isn't executed. this can be used to skip the loop code in certain cases.

```
    function ContinueExample()
    {
        local int i;
        log( "Demonstrating continue" );
        for( i=0; i<10; i++ )
        {
            if( i == 2 )
                continue;
            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 3
Completed with i=4
```

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.

```
   // Example of switch-case.
   function TestSwitch2()
   {
      switch( LightType )
      {
         case LT_None:
            log( "There is no lighting" );
            break;
         case LT_Steady:   // will "fall though" to the LT_Backdrop case
         case LT_Backdrop:
            log( "There is lighting" );
            break;
         default:
            log( "Something else" );
            break;
        }
    }
```

3.9 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 that 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 the entire regular UnrealScript commands plus several special functions known as "latent functions". A latent function is a function that executes "slowly" (i.e. non-blocking), 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 elapses. Latent functions can only be called from within state code, and not from within functions. Latent functions let you manage complex chains of events that 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 the specified 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 inheritance and scoping rules

In UnrealScript, when you subclass an existing class, your new class inherits 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 inheritance and scoping rules. The complete inheritance rules are:

A new class inherits all of the variables from its parent class.

A new class inherits all of its parent class's non-state functions. You can override any of those inherited non-state functions. You can add entirely new non-state functions.

A new class inherits all of its parent class's states, including the functions and labels within those states. You can override any of the inherited state functions, and you can override any of the inherited 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 extends 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 extends 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 that resolve 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 "extends" keyword to make the state expand on an existing state in the current class. This is useful, for example, in a situation where you have a group of similar states (such as MeleeAttacking and RangeAttacking) that 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 extends Attacking
{
   // Stick specialized functions here...
}

// Attacking from a distance.
state RangeAttacking extends 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.
Whenever 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 that your state may require.

3.10 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, comparing 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, with an additional space between the two strings. "string1"@"string2" = "string1 string2"
@=	string	String concatenation, with an additional space between the two strings, concat and assign (v3323 and up)
$	string	String concatenation
$=	string	String concatenation, concat and assign (v3323 and up)
*=	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
^^	bool	Exclusive or
&	int	Bitwise and
|	int	Bitwise or
^	int	Bitwise exlusive or (XOR)
=	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)
>>		same as >>
+	byte, int, float, vector	Add
-	byte, int, float,	Subtract

	vector	
%	float, int, byte	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
ClockwiseFrom	int(rotator lements)	returns true when the 1st is clockwise from the 2nd argument

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).
```

New operators are added to the engine from time to time. For a complete list of operators, check the latest UnrealScript source - specifically the Object class.

3.11 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 ASin(float A); Returns the inverse sine of the number expressed in radius.
float ACos(float A); Returns the inverse cosine of the number expressed in radius.
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.
float Ceil (float A); Rounds up
float Round (float A); Rounds normally
Unreal's string functions have a distinct Basic look and feel:
int Len(coerce string S); Returns the length of a string.
int InStr(coerce string S, coerce string t); Returns the offset into the first string of the second string if it exists, or -1 if not.
string Mid (coerce string 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 Left (coerce string S, int i); Returns the i leftmost characters of s.
string Right (coerce string] S, int i); Returns the i rightmost characters of s.
string Caps (coerce string S); Returns S converted to uppercase.
string Locs (coerce string S); Returns the lowercase representation of S (v3323 and up)
string Chr (int i); Returns a character from the ASCII table
int Asc (string S); Returns the ASCII value of a character (only the first character from the string is used)
bool Divide (coerce string Src, string Divider, out string LeftPart?, out string RightPart?); Divide a string into a left and right part of the divider, returns true when divided.
int Split (coerce string Src, string Divider, out array Parts); Split a string into parts, returns the number of items the string was split into.
int
StrCmp (coerce string S, coerce string T, optional int Count, optional bool bCaseSensitive); Perform a C like string compare. Count defines the max number of characters to compare. (v3323 and up)
string Repl (coerce string Src, coerce string Match, coerce string With, optional bool bCaseSensitive); Replace Match with With in the source. (v3323 and up)
string Eval (bool Condition, coerce string

ResultIfTrue, coerce string

ResultIfFalse); A short hand for a string assignment based on a condition. (v3323 and up)

Vector functions:

float VSize(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.
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 the foreach command is a variable that is assigned an actor for the duration of each iteration through the foreach loop.
Here are all of the iterator functions that work with "foreach".
AllObjects(class baseClass, out Object obj); Iterates through all objects.
AllActors (class<actor> BaseClass, out actor Actor, optional name MatchTag)
Iterates through all actors in the level. If you specify an optional MatchTag, only includes actors that have a "Tag" variable matching the tag you specified.
DynamicActors(class<actor> BaseClass, out actor Actor)
Iterates through all the actors that have been spawned since the level started, ignoring the ones placed in the level.
ChildActors(class<actor> BaseClass, out actor Actor)
Iterates through all actors owned by this actor.

BasedActors(class<actor> BaseClass, out actor Actor)
Iterates through all actors which use this actor as a base.
TouchingActors(class<actor> BaseClass, out actor Actor)
Iterates through all actors which are touching (interpenetrating) this actor.
TraceActors(class<actor> BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent)
Iterates through all actors 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<actor> 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<actor> 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).
VisibleCollidingActors (class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc, optional bool bIgnoreHidden);
returns all colliding (bCollideActors==true) actors within a certain radius for which a trace from Loc (which defaults to caller's Location) to that actor's Location does not hit the world. Much faster than *AllActors()* since it uses the collision hash.
CollidingActors (class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc);
returns colliding (bCollideActors==true) actors within a certain radius. Much faster than *AllActors()* for reasonably small radii since it uses the collision hash
ZoneActors(class BaseClass, out actor Actor)
Iterates through a list of all actors who are in the same zone as the object upon which you are calling the iterator.
Note: The iterator functions are all members of particular class (ZoneInfo in the case of ZoneActors(), and Actor in all other cases). So if you want to use an iterator from within a function in a non-Actor, you must have an actor variable and use the following syntax:
foreach ActorVar.DynamicActors(class'Pawn', P)
foreach ActorVar.Region.Zone.ZoneActors(class'Pawn', P)
So, from with an Interaction class, you could do:
foreach ViewportOwner.Actor.DynamicActors(class'Pawn', P)

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 extends 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 extends Pawn;

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

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

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 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.
```

3.12 Dynamic Arrays

Previously, we covered Arrays, which were static. What that means is that the size (how many elements are in the array) is set at compile time and cannot be changed. Dynamic Arrays and Static Arrays share the following common characteristics :
constant seek time - the time code spends accessing any given element of the array is the same, regardless of how many elements are in the array
unlimited element type - you can have an array of anything (other than bools) - ints, vectors, Actors, etc.
access behavior - you can access any element with an index into the array, and conversely, attempting to access an element at an index that is outside the bounds of the array will throw an accessed none.
Dynamic Arrays provide a way of having Static Array functionality with the ability to change the number of elements during run-time, in order to accommodate changing needs. In order use Dynamic Arrays, we need to know a few things.
The first is variable declaration. In order to declare a dynamic array, the syntax is: array<VARIABLE_TYPE> VARIABLE_NAME which is preceded by the appropriate identifiers (var or local, for example). An example would be: var array<int> IntList[7] . When script starts, IntList will start with 0 elements. There are methods supported by Dynamic Arrays that allow us to add elements to the array, take elements out, and increase or decrease the length of the array arbitrarily. The syntax for calling these methods is (using our IntList example): IntList.MethodName(). The following is a breakdown of the supported methods and a brief explanation of their parameters :
Insert(int index_to_insert_at, int how_many_elements_to_insert) - this allows us to tell the array to create more elements and create them starting at a specific location in the array. Inserting 5 elements at index 3 will shift up (in index value) all elements in the array starting at index 3 and up (shifting them up by the number of elements to insert).

Remove(int index_to_begin_removing_at, int how_many_elements_to_remove) - this allows us to remove a group of elements from the array starting at any valid index within the array. Note that any indexes that are higher than the range to be removed will have their index values changed, keep this in mind if you store index values into dynamic arrays.

Dynamic Arrays also have a variable called Length, which is the current length (number of elements) of the dynamic array. To access Length, using our example array, we would say: IntList.Length . We can not only read the Length variable, but we can also directly set it, allowing us to modify the number of elements in the array. When you modify the Length variable directly, all changes in array length happen at the 'end' of the array. For example, if we set IntList.Length = 5, and then we set IntList.Length = 10, the extra 5 elements we just added were added to the end of the array, maintaining our original 5 elements and their values. If we decreased the Length, the elements would be taken off the end as well. Note that when you add elements to the array, either by Insert() or by increasing Length, the elements are initialized to the variable type's default value (0 for ints, None for class references, etc). It is also noteworthy to know that you can increase the length of a dynamic array by setting an element index that is greater than the array's current Length value. This will extend the array just as if you had set Length to the larger value.

A word of caution - the Length member of a dynamic array should never be incremented / decremented by '++', '--', '+=', or '-=', nor should you pass Length to a function as an out parameter (where the function can change the value of it). Doing these things will result in memory leaks and crashes due to Length not being accurate any more; only setting the Length via the '=' operator (and setting an element at an index larger than Length) modifies the actual length of the dynamic array properly.

A final note - dynamic arrays are **not** replicated. You could get around this by having a function that replicates and has two arguments, an index into the dynamic array and the value to store there. However, you would also have to consider consequences of elements not being the same within a space of a tick on client and server.

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. 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 determine 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.

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 three-pass

Unlike C++, UnrealScript is compiled in three distinct passes. In the first pass, variable, struct, enum, const, state and function declarations are parsed and remembered; the skeleton of each class is built. 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. The third phase parses and imports default properties for the class using the values specified in the defaultproperties block in the .uc file.

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 only at times when all actors are at their lowest possible UnrealScript 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.

Unreal files

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 are being covered 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 about 20X faster than UnrealScript. 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