About

Team

Old News

Mods
- OSP Tourney Q3A
- OSP Tourney Q2
- King of the Hill
- OSP Rocket Olympics
- OSP Wolf

- Download

DM Maps

Old DOOM Page

Tutorials

Extras
- Mame Front End
- Cycling

Contact Us

Home

GreenMarine's Object Oriented Logic Tutorial

(with specific references to OO's place in UnrealScript)

> "For the layprogrammer." - GreenMarine
>
> "You can use the pepper grinder to create pepper by turning the crank...BUT, a pepper grinder is _not_ pepper, so you MUST NOT TRY TO EAT IT!" - Tim Sweeney, Epic MegaGames, Inc.

You can always find the latest version of this document at http://www.OrangeSmoothie.org/tuts/GM-OOtutorial.html.

Drafted by: Brandon "**GreenMarine**" Reinhart
Contact: ~~greenmarine@planetunreal.com~~

Version: 1.0

ToDo

- Constructors
- Destructors
- State
- Advanced Techniques

Introduction

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

Legal

The Way Programmers Think

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

Breaking It Up

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

Getting Organized

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

The Class

In OO logic, a "class" is a description of what a thing might be if it were created. A class is a generic template. It defines abstract properties

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

Class Mechanics

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

```
class MyClass expands MyParentClass;
```

The class specifier tells UnrealScript that you are starting to define a new class. MyClass is the name of this class (it can be whatever you want, preferably something meaningful, like CarPart). We'll get to the rest in a bit.

After you've made your class declaration in UnrealScript you are ready to define the class's instance variables. This is done by listing a series of property variables:

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

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

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

This is just a matter of taste, but it won't hurt. Its usually a _good thing_ to make your code more readable, especially if you want to share it with others or come back to it later.

Defining methods for your new class are simular to defining instance variables, just make a list of functions:

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

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

Once again, you might separate the method area of your class from the rest of the code by using a comment line. It is also highly suggested that you name your methods after their functions. For example, a function that cleans your socks might be called cleanSocks().

In UnrealScript, we call object instantiation "spawning." As such, you use the Spawn() function to create a new object from a class template:

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

A Brief Discussion of Methods

Objects are independent collections of "interactive" data that sit in memory. The keyword is _independent_. Each object does its own thing. If you have a class called MyBot and you spawn two objects from that class called BotA and BotB those two instantiations don't know about each other. This leads us to the next concept of Object Oriented Programming: objects edit themselves.
When an object is spawned it is usually stuffed into a hash table in memory, given a lookup key, and forgotten about. The system uses the lookup key to find the specific object in the hash table if you want to do something to it, but for all intents and purposes the object is just sort of unavailable. Unavailable in the since that, unlike a normal variable, you can't just change it. You cannot, for example, rip open an object and change the contents of the instance variables. (This is one of the ways in which objects are different from structs in C++.) Instead, you have to tell the object to change itself. This is done through the methods.
The methods of a class define the way an object will act when it is spawned. If you want to change an instance variable of an object, you have to have written a method in the class that allows this to behavior to take place. Our CarPart class might have a method that looks like this:

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

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

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

Methods, Variables, and Object Security

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

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

Notice the difference. In the method case, we tell the object to change itself and in the assignment case we change the nature of the object directly. This presents another one of those "How Programmers Think" issues. You're probably asking yourself if I can just alter an object directly, why would I ever use a method to do it? Well, think about it.
What if there were very special restrictions on the instance variable "color"? For example, you might want the color variable to only contain values from 1 to 10. Anything outside of that range would maybe cause your program to act unpredictably. In that case, it just wouldn't be a good idea to allow the object's user to edit the color variable directly. Right? Even though _you_ might know that 1 to 10 are the only correct values for color, someone else who uses your code might not. The solution is to make the instance variable private, so that only the class itself can change it:

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

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

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

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

Class Families

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

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

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

Fundamental I: Inheritance

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

```
Object
| expanded by
Actor
| expanded by
```

```
CarPart
| expanded by
SteeringWheel
```

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

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

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

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

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

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

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

Fundamental II: Polymorphism

Poly what? Its more of that crazy programmer speak. (If you've understood everything up until now, you are more a programmer than you think.) Polymorphism is another one of the fundamentals of object oriented programming. In inheritance, the child class gains the instance variables, methods, and states of the parent class... but what if we want to change those inherited elements? In our car example, we might have a Pedal class that defines a method called pushPedal(). When pushPedal() is called, the

method preforms a default behavior (maybe it activates the breaks.) If we expand the Pedal class with a new class called AcceleratorPedal, the pushPedal() method suddenly becomes incorrect. An accelerator certainly shouldn't turn on the breaks! (Or you're gonna have a lot of lawsuits when you release your program, believe you me).
In this situation, we have to replace the behavior we inherited from Pedal with something new. This is done through a process called "Polymorphism" or "Function Overloading." You'll run into this all the time when you write UnrealScript. To borrow an explanation from Tim Sweeney:

> "[Function overloading] refers to writing a new version of a function in a subclass. For example, say you're writing a script for a new kind of monster called a Demon. The Demon class, which you just created, expands the Pawn class. Now, when a pawn sees a player for the first time, the pawn's [SeePlayer()] function is called, so that the pawn can start attacking the player. This is a nice concept, but say you wanted to handle [SeePlayer()] differently in your new Demon class."

To do this, just redefine the function in the child class. When the class is instantiated, the object will have the new behavior, and not the parent behavior. If you don't want anyone to redefine a function you have added to a class, add the "final" specifier to the function's declaration:

```
function final SeePlayer()
```

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

Bringing It All Together

So now you know the fundamentals of Object Oriented design and have a good idea of how objects relate to one another. What do you do next?
The best advice is to get hacking. Dive into the code and don't come up for air even if the promise of food, sleep, or sex looms near. Seek the zone. Or...you can always read Tim Sweeney's guide to UnrealScript. It goes into much greater detail about the syntax surrounding OO in Unreal. In addition, I suggest you find other resources on the net discussing OO. As I develop this paper, I'll try to come up with some good link, which I will list below. There are a lot of subtle elements of OO that can only be learned. Some aren't really supported by Unreal, some are. Some are merely ways of thinking.
And that brings me to my closing point. OO is as much a way of thinking as it is a way of programming. As you walk to school or drive to work, try imagining the relationship between things you see (a tree has leaves, a rose is a flower). This will greatly enhance your understanding of OO. Create complex relationships in your mind and then find ways of representing them in code. To those who really understand it and really enjoy it, programming is a mental, physical, and spiritual task. It might sound wierd, but programming touches the fundamental ways in which we think and solve problems. If you can think in OO, then your mind is unrestricted when it comes to solving problems in OO. The more you use it, the more you will come to realize it is

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

- Brandon "**GreenMarine**" Reinhart, May, 1998

Notes

- SkinDoggy notes that a class MUST extend some parent class in UnrealScript.

Useful Resources

- UnrealScript Language Reference
- The OO FAQ

About the Author

Brandon Reinhart, known as GreenMarine in the Quake/Quake2/Unreal scene is a dedicated mod hacker obsessed with game and graphics programming. Brandon has authored the King of the Hill modification for Quake2 and currently has multiple Unreal projects planned, in addition to a technology demonstration. Brandon is currently looking for employment in the games industry.

Special Thanks

- Epic MegaGames
- Tim Sweeney
- Dr. Pepper
- Mr. Pibb
- Cabaret Voltaire
- Front Line Assembly
- Nick Cave
- SkinDoggy from #UnrealED for Corrections

History

May 27, 1998 - Made corrective changes.
May 27, 1998 - Version 1.0 in PlainText and HTML
May 27, 1998 - First Revision
May 27, 1998 - First Draft