Chapter 3: Developing the Racing Game

Table of Contents

Introduction

With this project we'll get into the basics of how to develop a new gametype for the UDK platform. In other words we're going to start not by developing a whole new game from scratch, but by changing the rules for an Unreal Tournament gametype, which will give us a firm idea of how Unrealscript is organized and where to go when we want to make more major changes or when we want to develop brand new content in the future. Topics that will be covered in this project include:

- Changing basic rules by developing a new gametype script
- Communication between gametype scripts, player data, and in-game actors

Part 1 - Collecting our Goals

When you begin coding it's best to have a very concrete task in mind, which means breaking down the goals you need to meet in order to make your project functional. What variables do we need to have around in order to keep track of the data we need? Who needs to know about it, the game at large or the player? What's the logic for moving it around?

In our case we're developing a racing mod for Unreal. I picked this project because it's a relatively straightforward modification that covers all of the basics of this workflow and because the Unreal Tournament 3 content included in UDK handily provides some nice, functional vehicles for us to use. What we need to focus on, though, is what exactly the rules of a racing game are meant to be and what tasks our scripts have to handle that aren't already in UDK right now. I've made a quick list describing these rules:

- Players need to make three laps around a racing track
    - Whoever does this first, wins
- Players must make full laps in the intended direction--no going back-and-forth over the finish line or circumventing the structure of the track
- The game must be able to keep track of what player is in what place in the race
- We must display the player's current laps and their current place in the race to them
- A mini-map displaying all players in a race and their vehicles must be created
- We must spawn players directly into vehicles
- We must insure that players re-spawn in reasonable places on the track

Now that we've collected some concrete goals to tackle, we can start thinking of which ones need to be done first in order to make our new gametype playable and give it its basic functionality.

Don't feel too married to this list as new tasks and challenges will always crop up on the way to completing a coding task. A good example of this is the second bullet -- "Players must make full laps in the intended direction." This problem is especially fluid and may require us to develop several more sub-tasks, but we'll worry about it when we get there. In the meantime, let's start with the basics--getting new base rules going in our game.

Part 2 - Setting up the Gametype

The Basics of Gametypes

Any game has a class that acts as the "brain" of the game, called a gametype script. Whenever we start a level, an instance of one of these classes is called into being to track the game, and 90% of the functionality we need to track the rules of the game is stored in here. UDK has a whole hierarchy of scripts devoted to tracking player and game data. Going from the top down it looks like this:

- Info
    - GameInfo
        - FrameworkGame
            - UDKGame
                - UTGame
                    - UTDeathmatch/UTCTF/UTTeamGame/Other UT Gametypes

Each level of this hierarchy stores different information regarding the rules of the game.

- **GameInfo** is the lowest-level code we have for doing this, and stores all the basics on where objects are and what state the game is in.
- **UTGame**, meanwhile, stores all information and functions pertaining *specifically* to Unreal Tournament, and the scripts that extend off of that are new Unreal Tournament game modes like Capture the Flag and Deathmatch.

These two scripts are what we most want to pay attention to, but for the sake of thoroughness I'll explain the other sets of scripts as well:

- **UDKGame** is blank except for one piece of code devoted to assigning gametypes based on level prefixes like DM- and CTF- (IE: DM-Divinity, CTF-FacingWorlds).
- **FrameworkGame** is also blank except for a handful of items pertaining to mobile game input. It was most likely added recently.

- **Info** is the root class for holding any kind of information. Only the most basic kinds of information, like player number, map name, et cetera are stored here; it has no information pertaining to collision or ingame movement.

Anything from GameInfo lower on will give you the information you need to track a game, though it seems that Epic wants us to extend the UDKGame class when we start a brand new game as it includes the vital bit of text that makes classifying maps and assigning gametypes easy.

If you want to know more about Gametypes, UDN has a handy Gametype Technical Guide that more fully lays out the where and what of all these classes and their functions.

Basic Setup

We want to create an Unreal Tournament gametype, so we're going to extend UTGame. So, start a new script, and write it like this:

```
class UTRaceGame extends UTGame
config(game);

DefaultProperties
{

}
```

The "config" command has been added to our class declaration in order to make this class configurable in the future; IE, we can change certain parts of it with an INI file later on.

We now have a new gametype, although a lot of questions have yet to be answered, the most important of which right now is: how do we work with this gametype in the UDK editor, and how do we assign it in a map?

Remember that bit of code I mentioned in **UDKGame**? That one that deals with acronyms? We're going to be using that. Among the common variables of a gametype script are the level prefixes. All we have to do is assign them:

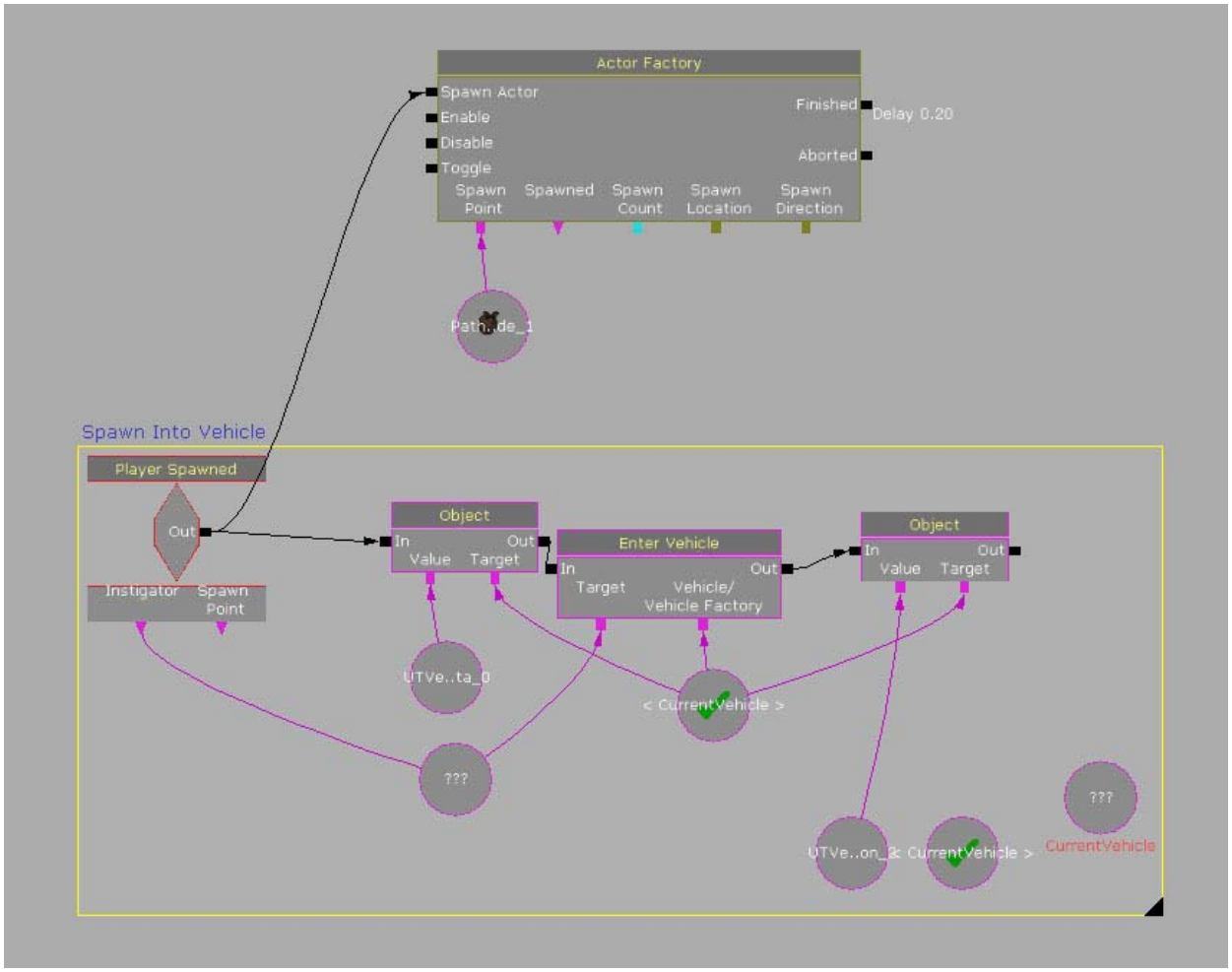
```
DefaultProperties
{

      Acronym = "RC"
      MapPrefixes[0] = "RC"

}
```

Done. Save your script, start up UDK, let it compile, then start up UDK again and make a new map. Now when you go into View --> World Properties and scroll down to "Game Types Supported on this Map," you'll be able to select the gametype for your map, like so:

When you name your map, name it RC_MyLevel, and then when you start it up, it will start up with the UTRaceGame gametype.

Getting the Map Ready

It's a good idea to do a little bit of work on our map before we go much further so that we can test it reliably in the near future. This needn't take you very long; just get a PlayerSpawn and a bit of light in there and create a ring-shaped racing track. If you like, you can put some additional Kismet to spawn the player into a vehicle. One way or another, I suggest having an additional bot around just as a dummy so that the player has something to compare to when they start taking laps and whatnot.



Customizing the Gametype

It's all well and good that we have a Gametype running, but now we need to add some functionality to it so that we can start differentiating it from UDK's default gametypes a bit more and verify that it actually works.

We'll start by knocking out a few of our simpler goals. We don't want the player spawning with UDK's physics gun, and we want the player who takes laps in the race to win--*not* the player who racks up the most kills.

To start off, let's open up the Gametype Technical Guide and start looking around. If we scroll down to "GameInfo Settings," it has a whole list of settings that fall under the DefaultProperties section that we can edit. If we look under the "GameInfo Functions" header, we also see detailed for us a summary of the many functions that drive an Unreal gametype. You should be able to pick out several that will act as good starting points. There's even more available in the UTGame Class, which I also suggest you open up in your scripting editor and browse through since that's what contains the rules that we're extending.

The DefaultProperties values we want to pay attention to are bScoreDeaths, bGivePhysicsGun, and bIgnoreTeamForVoiceChat. These are UTGame-related DefaultProperties. Set bScoreDeaths and bGivePhysicsGun to false and bIgnoreTeamForVoiceChat to true.

```
DefaultProperties
{

	//Don't want to score with kills
	bScoreDeaths = false;

	//No need for team voice chat as this is no team gametype!
	bIgnoreTeamForVoiceChat=true

	//We also don't need the physics gun or any other gun.
	bGivePhysicsGun=false

}
```

This takes care of a couple of our tasks very neatly, but let's make absolutely sure. In browsing the documentation on GameInfo you should have seen a function called ScoreKill. Let's take advantage of our extensibility and re-write it in the body of our code.

```
function ScoreKill(Controller Killer, Controller Other)
{
return;
}
```

ScoreKill is now totally blank and whenever a player kills someone all it does is return a void value--for *this* gametype, anyway. We could call all the stuff from the original ScoreKill function if we really wanted, but as stated before we do not want winning the game to be dependent on this.

Compile your game, run it with the dummy bot, and test this out. You should spawn without the physics gun, and killing the dummy bot should net you no points. We now know that our new gametype works.

Part 3 - Implementing a New Rules System

Now we need to actually develop our *own* way of winning the game. In terms of programming, game rules are *completely* arbitrary. If we take a close look at UTGame, where the basic rules for all of Unreal Tournament's Deathmatch-based games are defined, we find out that the scoring system that it makes such a big deal over ingame is actually virtually meaningless and has no relation to any scripts outside of UTGame aside from player data that it has to pass around in order to work.

The scoring array doesn't do anything, it doesn't last longer than the player's time in a match, and the ability to win the game through a score is very loosely defined and can be easily re-defined. In fact, the game doesn't even *really* declare an actual winner when it sets a winner. It just stops the game and focuses the camera on whoever is arbitrarily deemed the winner. If we wanted to, we could write a function that calls "SetEndGameFocus" on anybody we wanted to, regardless of score. It just so happens that it's written to work when someone *does* achieve the desired number of kills in a match.

In some of the more gametype-specific scripts there *is* network code for applying these scores to online leaderboards, but there is absolutely nothing in a multiplayer gametype itself that strongly enforces rules outside of a big if/else tree and a couple of Booleans. In other words, we're dealing with simple Command Prompt-level programming.

To sum it up, how we define our gametype isn't in the least bit as dependent on what's written in UTGame as one might think. UTGame is an arbitrary collection of code designed to create the presentation and system for enforcing rules, therefore so is our script. What we do with it is then more about what code we choose to *ignore* than whose rules we have to live by. We are the programmers, so we define our own rules however we want.

In fact, let's start by creating a "SetWinner" function that'll do exactly what we just said and declare a winner on whoever we want.

```
function SetWinner(PlayerReplicationInfo Winner)
{
	//setting the end of game time
	EndTime = WorldInfo.TimeSeconds + EndTimeDelay;
	//Setting the winner in Game Replication Info
	GameReplicationInfo.Winner = Winner;
	//Aaaand money shot of the winner!
	SetEndGameFocus(Winner);
}
```

The code inside this function is borrowed from the bottom of the UTGame script's "CheckEndGame" function and defines all the basic data that we need in order to properly set the winner of a game. As we can see, "Winner" is defined as a type called "PlayerReplicationInfo," which is a valuable resource for us and which I recommend you look at. I'll explain more about Player Replication Info later. For now we have a function that we can call on anybody we want, for whatever reason.

Developing a Goal Trigger

As with anything in programming we want to take this one thing at a time. We can create a configurable system of laps and checkpoints later. For our first task we'll just make sure whoever crosses the finish line first wins.

The easiest way that we can create a finish line for our game is to use a trigger volume; this lets us re-size and re-define the area of our goal as needed in our level editor and provides us all the functionality necessary to process actors passing through it right off the bat. The only trouble is making that goal trigger communicate the right information between our player and our custom gametype. To do that we're going to need to script a *new* trigger that adds what we need for that purpose.

Start a new script up and call it "RC_GoalTrigger," extending it off of "TriggerVolume." You can find TriggerVolume under the "engine" folder as it is one of the most basic objects that a UDK game can have.

There are a few extra things we need to do with this class before we can start making it work for us. First, there's something we have to add in the header:

```
class RC_GoalTrigger extends TriggerVolume
      placeable;
```

The "placeable" keyword means exactly what it says; this Class is placeable within the level editor. Any time you want to create a class that you can manipulate like this, add this keyword to the Class declaration.

Creating a "Touch" Event

Next, we need to define the event for when an actor enters the trigger. To do this, we're going to re-define the "Touch" event.

```
simulated event Touch(Actor Other, PrimitiveComponent OtherComp, vector
HitLocation, vector HitNormal)
{

}
```

To break this down: "simulated" is a prefix we can put on functions and events. An "event" is something that's triggered as opposed to an action that we create, as with functions. When you write an event, you're saying "when this happens, do what's inside the brackets" instead of storing functionality for yourself to call up later. Other events would include keyboard input or being hit with a shot from a weapon. The stuff inside the parenthases is the parameters of this event, and they are as follows:

```
Actor Other
```

The object touching our trigger. "Other" is a very common word that scripters use to describe "any other object that hits this."

```
PrimitiveComponent OtherComp
```

Not completely sure what this actually is, but it likely has to do with hitboxes.

```
vector HitLocation
```

The place on the trigger where the other actor hit.

```
vector HitNormal
```

The direction in which the object was passing when it hit the trigger.

The only one of these parameters we really need to concern ourselves with is the Actor, thankfully, and 90% of the time that we're dealing with triggers, unless we're doing something a lot more complicated and specialized--like, say, a fluid volume that makes ripples wherever an object lands in it.

Using the "Super" Command

There's another piece of code we're going to want to put inside this event--namely, the "super" command. What this command does is call the *original* function from the script that we're extending.

```
super.Touch(Other,OtherComp,HitLocation,HitNormal);
```

To sum it up, all you do is type "super," separate the function or event name with a ., and then fill in the names of each of its parameters as originally defined--not the variable types, just the names. This allows us to keep the original functionality of the touch event, so if we want to, say, use Kismet to add even *more* reactions when a player passes through one of our goal triggers, we could definitely do so.

Giving our Goal Trigger Guts

Now, let's start thinking of our custom functionality. This trigger has to be able to do two things: first, it has to get the actor that's passing through the trigger—which it already does just by virtue of being a trigger. Second, it has to pass that actor's player data along to the gametype script, which is the part that can be a bit difficult to understand since the gametype script defines game rules and not a physical object that we can easily find. It's attached to the map that we're playing—obviously, because we took time in the UDK editor to assign it to our map in the World Properties editor—but where is it?

Getting the Gametype and Casting

Well, we just answered that question! The gametype script is attached to the World Info for whatever game we're running. All we have to do, then, is figure out how to obtain the World Info in our script and snag the gametype from that. Fortunately, this boils down to one very handy command, "WorldInfo," which returns for us the world properties of our current game. The gametype is one of those properties, and we can call it with "WorldInfo.Game," like so:

```
WorldInfo.Game
```

Now that we know how to call the gametype script from within a custom object, we can call any function that's attached to it... if we jump through a couple of extra hoops, anyway. WorldInfo.Game is a very general class call that returns whatever gametype class is assigned to the map's world info. If we use it in one of our racing game maps it will give us our custom gametype class and all the data that comes with it since we defined our gametype in our map as "UTRaceGame," but Unrealscript *itself* has no way of knowing that we did that until the game actually starts running. Therefore if you tried to call our SetWinner function...

```
WorldInfo.Game.SetWinner(Other);
```

... all you would accomplish is throwing an error telling you that the function we made doesn't exist. So, we need to help Unrealscript a little bit by casting the GameInfo that we get back from WorldInfo.Game as our racing gametype.

Casting is basically converting one variable or class type to another. In our case we want to convert what we get out of WorldInfo.Game *specifically* to the UTRaceGame gametype that we've defined. Since we're only using this trigger inside of maps that take our racing gametype, this doesn't *actually* do

anything to our game's data, it just makes it very explicit in our code that we mean this specific gametype and want to use its functions. To cast WorldInfo.Game, simply put it inside parenthases following the class name, like so:

```
UTRaceGame(WorldInfo.Game);
```

And now, we have our gametype, *as a racing game*, and can call our SetWinner function on whoever triggered the touch event.

```
UTRaceGame(WorldInfo.Game).SetWinner(Other);
```

Organizing Information with Local Variables

This can get a little bit confusing, but you would be surprised how often we have to do things like this--using a function call to get a variable of some kind, then casting it as another type of variable, then calling a function off of *that*. It's just how the computer moves information like this around.

It's most efficient to write this code the way it's defined above in one quick declaration so that you're not storing unnecessary data, but you can make things easier for yourself to understand by using **local** variables to store information.

Simply put, a local variable is a variable that exists *only* inside the function that you're currently writing and *only* when that function gets called; the computer cleans up after it after the function finishes running. They're declared the same way most normal variables are declared, except that you write "local" instead of var, like so:

```
local int MyLocalInteger;
```

Right now, we're looking at using them to break down the SetWinner call we just wrote. If we break it apart, we have three variables total:

- A UTRaceGame
- "Other," which is already defined as a parameter that our function takes in
- WorldInfo.Game

So, we can re-write THIS...

```
simulated event Touch(Actor Other, PrimitiveComponent OtherComp, vector
HitLocation, vector HitNormal)
{
//Cast our gametype, set the winner all in one command...
UTRaceGame(WorldInfo.Game).SetWinner(Other);
}
```

... like THIS:

```
simulated event Touch(Actor Other, PrimitiveComponent OtherComp, vector
HitLocation, vector HitNormal)
```

```
{

//Make a place to store our racing game's information...
local UTRaceGame MyRaceGame;

//Cast the gametype and store it...
MyRaceGame = UTRaceGame(WorldInfo.Game);

//Set the current game's winner
MyRaceGame.SetWinner(Other);
}
```

As you can see, it becomes a lot clearer when we separate these commands out, if a bit lengthier.

Testing our Trigger

At this point we now have a goal trigger that can call a function inside our gametype script. All that's left to do is open up UDK, let our code compile, and give it a test run. You ought to be able to find your new trigger volume in the "volumes" dialog now, and passing a player pawn through it should cause the game to end.

Congratulations, you've just changed the rules of UDK's default game! We have an awful lot of refining to do, however, and to do that we're going to need to start looking elsewhere.

Why do we have to call our functions from the gametype script?

We could offload a lot of the functions in our gametype script to our goal trigger or any of the other special actors that we create for our game, but then we'd be calling all that code for every instance of that object. Every goal trigger would completely repeat the SetWinner function, which is unnecessary and can get confusing.

If we keep our major functions in the gametype script, though, and use it as the "brain" that controls our game, then there's one object that has those important scripts at any given time. That code and data gets instanced *one* time as opposed to several, which is a lot more efficient, and the functions are kept closer to where the important data is kept.

Part 4 - Creating Custom Player Data

We've got our game script communicating with an ingame actor we've created, but at this point we've reached a bit of an impasse as we can't keep track of individual players' laps or placement in the race.

Introducing Player Replication Info

To do that we're going to need to create a new place to store that information. One would think it would be part of our player pawn, but this is not the case. If you recall earlier, when we wrote the SetWinner function, we referred to a Class called PlayerReplicationInfo.

```
function SetWinner(PlayerReplicationInfo Winner)
{
	//setting the end of game time
	EndTime = WorldInfo.TimeSeconds + EndTimeDelay;
	//Setting the winner in Game Replication Info
	GameReplicationInfo.Winner = Winner;
	//Aaaand money shot of the winner!
	SetEndGameFocus(Winner);
}
```

PlayerReplicationInfo is to the player what GameInfo or a gametype script is to our map and the game rules. It's basically a private storage space that's attached to a player's pawn to keep track of important metagame information like score, number of deaths accumulated, team number, and whether the player is controlled by an AI bot or not.

This is as opposed to the information that's kept in the Pawn and UTPawn Class, which all pertains to the animations and physical behavior of the player's character in the game world, including their current health value, movement speed, and jump height. While there's little reason that both sets of data couldn't be in the same script, organizationally it makes sense to separate out what the gametype specifically needs to communicate with and what dictates the way a character behaves and controls.

It's with the PlayerReplicationInfo that we'll be storing our new game data, so let's make a new script extending it:

```
class RCPlayerReplicationInfo extends UTPlayerReplicationInfo;
```

Like with anything, Unreal Tournament has its own Player Replication Info class for storing information specific to Unreal Tournament gametypes, and that's what we're extending since we're still working under the pretense of creating a new Unreal Tournament gametype.

We notably don't have to do a whole lot with this script itself; we just have to add in a variable for laps and define it in DefaultProperties.

```
class RCPlayerReplicationInfo extends UTPlayerReplicationInfo;

var int currentLap;

DefaultProperties
{
	currentLap = 0;
}
```

We can add more data to this script as needed as we add complexity to the gametype, but for now this is all the data that we need.

Adding the Player Data to the Player in the Gametype

The next question is: how do we add the player data *to* the player? How does the game know to assign this instead of UTPlayerReplicationInfo?

It's at this time we need to re-visit our gametype script, as this is something that has to happen when the game starts. If we look at the DefaultProperties of either GameInfo or UTGame, we'll see a few entries that look like this:

```
HUDType=class'UTGame.UTHUD'
PlayerControllerClass=class'UTGame.UTPlayerController'
ConsolePlayerControllerClass=class'UTGame.UTConsolePlayerController'
DefaultPawnClass=class'UTPawn'
PlayerReplicationInfoClass=class'UTGame.UTPlayerReplicationInfo'
GameReplicationInfoClass=class'UTGame.UTGameReplicationInfo'
DeathMessageClass=class'UTDeathMessage'
BotClass=class'UTBot'
```

The DefaultProperties for our gametype script, as we can see here, define a lot of default values for things like what type of player pawn is supposed to be used for the game, what type of player replication info to use, and even what type of HUD to use. We'll be re-visiting this often, but in the meantime we want to pay attention to the PlayerReplicationInfoClass entry and re-define it in our *own* gametype script, making our DefaultProperties look a bit more like this:

```
DefaultProperties
{
	Acronym = "RC"
	MapPrefixes[0] = "RC"
	//Don't want to score with kills
	bScoreDeaths = false;
	//Don't want to give the physics gun
	bGivePhysicsGun=false;

	//I reject your player replication info and substitute it with my own!
	PlayerReplicationInfoClass = class'UTRaceGame.RCPlayerReplicationInfo'
}
```

The format for re-defining one of these classes is, as you might have noticed, to preface it with the name of the folder that we put the script in. This isn't actually necessary and you *could* just write "RCPlayerReplicationInfo," but if we actually name the folder our class is in then the game doesn't have to do such a wide search for it. Thus, any time you want to assign a class like this, use the format "MyScriptFolder.MyCustomClass."

More importantly, though, we have the game assigning our player data to the player when it starts. Now player pawns will be carrying around the "Laps" variable that we defined in addition to all the other usual variables that go with them.

Adding Laps; Using Our Player Data

Now let's actually work with it. We want our gametype to tally up laps every time the player passes the GoalTrigger we made instead of just outright winning the game.

Now, go back to UTRaceGame and make a new function. We're going to call this one GoalTouched, and it will take in the actor that we get from touching the GoalTrigger as a parameter.

```
function goalTouched(Actor Other)
{

}
```

While we're at it, go back and put "goalTouched(Other)" into the GoalTrigger script in place of "setWinner(Other)" so that we're calling *this* function when the player passes through the checkpoint instead of just calling them the winner. Instead, we're going to have our goalTouched function set our winner.

Again, the reason that we're having GoalTrigger *call* the goalTouched function instead of contain the data we're about to put in it is because it's more efficient for the triggers to just be "dummy terminals" that activate the *one* function in the gametype script rather than having each trigger contain the full guts of the program. Plus which, it's that many fewer times that we have to get information from the Gametype script.

Now, inside our new function, we have to get the information of the player passing through it. The "Touch" event gives us general actors; we get all the information that comes with a Pawn when it passes through the trigger, but all Unrealscript knows to look for is the "Actor" class, so we have to do a lot of the same things we did when we were communicating between the gametype and the GoalTrigger.

```
function goalTouched(Actor Other)
{
	local Pawn OtherPawn;
	local RCPlayerReplicationInfo RCRep;

	//The Pawn we're storing.
	OtherPawn = Pawn(Other);

	//The Racing Replication info for that pawn.
	RCRep = RCPlayerReplicationInfo(OtherPawn.PlayerReplicationInfo);
}
```

This should start to look familiar now. We make a place to store the pawn that steps through the goal trigger and our custom Player Replication Info, and then we cast the actor we got from the goal trigger as a Pawn in order to get its functions and properties. We then do the same thing with its Replication Info property, casting it *specifically* as RCPlayerReplicationInfo.

Now that we have the RacePlayerReplicationInfo exposed and at our disposal, we can edit it, simply by adding:

```
RCRep.currentLap = RCRep.currentLap+1;
```

You could also write it like this:

```
RCRep.currentLap++;
```

Or like this:

```
RCRep+=1;
```

All we're doing here is telling it to add 1 to the current value of currentLap for the player that's passing through this goal. It's amazing how all the classes that have to communicate with each other make such a simple operation complicated, but that's the Kid's Meal version of why programmers get paid the big bucks. Fortunately, it isn't a *lot* of code, it's just slightly confusing. It isn't *complicated* code, it's just stupendously specific.

Setting a Number of Laps to Meet

Now we have one last thing we need to do in order to set a winner. Namely, our gametype needs to have a requisite number of laps to reach. All we have to do is define a "NumLaps" variable at the top of our script, then set a default value in DefaultProperties.

Then, in our GoalTouched function, we just need to add an if/else statement, saying that if the current number of laps that the player has equals the "NumLaps" variable that we set, they should be declared the winner; otherwise, we tally up a lap.

So, to reiterate, our goalTouched function should look like this:

```
function goalTouched(Actor Other)
{
	local Pawn OtherPawn;
	local RCPlayerReplicationInfo RCRep;

	//The Pawn we're storing.
	OtherPawn = Pawn(Other);

	//The Racing Replication info for that pawn.
	RCRep = RCPlayerReplicationInfo(OtherPawn.PlayerReplicationInfo);



	if (RCRep.currentLap == NoLaps)
		{
		//Set a winner if it's the final lap.
		SetWinner(OtherPawn.PlayerReplicationInfo);
		}

	else
		{

		//Add a Lap to the Tally
		RCRep.currentLap++;
		}

	OtherPawn.ClientMessage("Laps: " + RCRep.currentLap);

}
```

You'll notice we've added one last bit of script to this function for debugging purposes:

```
OtherPawn.ClientMessage("Laps: " + String(RCRep.currentLap));
```

This simply sends a message to the pawn who's touching the trigger, showing the number of laps that we've completed--casted as a string, since ClientMessage in UDK doesn't automatically do that for us like other debug systems and logs. We'll get rid of it when we've verified that this works.

Now, save your script, let UDK compile it for you, check for any errors, and run the game. The goal trigger **won't work** when you try to pass through it with a vehicle, but we'll be fixing that soon. For now, run through the trigger *without* a vehicle. You should get the ClientMessage showing the number of laps, and if you pass through a number of times equal to whatever you set NumLaps to, you should end the game.

Making the GoalTouched Function More Efficient

While all this code should work fine, it's not *completely* efficient as it's processing and casting *every* actor that passes through it. Every rigidbody, vehicle, and projectile going through the GoalTrigger is having laps added to it, which is going to lead to some very silly glitches.

We want to *only* do lap counting on player pawns, and nothing else, so let's wrap what we've got in the GoalTouched function inside an if/else statement that checks to see if there *is* in fact a pawn associated with the actor we're getting from the trigger.

```
local Pawn OtherPawn;
local RCPlayerReplicationInfo RCRep;

OtherPawn = Pawn(Other); //The Pawn we're storing.

//If there isn't a pawn associated with this..
if (OtherPawn == none)
{
      //Just don't bother with it.
      return;
}
else
{

RCRep=RCPlayerReplicationInfo(OtherPawn.PlayerReplicationInfo);

//Put everything else we've written so far in here

}
```

The "none" command is Unrealscript's version of "void," which is how you explicitly state that there is absolutely *nothing* stored in a variable. So, if there is in fact *no pawn* associated with this Actor we're trying to process, we use the "return" value to just cut the script off right there and not do anything else--no adding laps, no processing a winner, nothing. We aren't even processing the PlayerReplicationInfo unless we absolutely have a pawn to do so with.

But, we can add still yet *another* level of protection to this. Just below RCRep, we can do the exact same thing with the PlayerReplicationInfo:

```
	RCRep=RCPlayerReplicationInfo(OtherPawn.PlayerReplicationInfo);

		if (RCRep==none)
		{
			return;
		}
		else
		{
			//Put everything else we've written so far in here
		}
```

Now, if the Pawn *somehow* doesn't even have PlayerReplicationInfo associated with it, for whatever reason, we just ignore it. Otherwise, everything else we do still counts.

For reference's sake, the whole thing should look like this now:

```
local Pawn OtherPawn;
	local RCPlayerReplicationInfo RCRep;

	OtherPawn = Pawn(Other); //The Pawn we're storing.

	//If there isn't a pawn associated with this..
	if (OtherPawn == none)
	{
		//Just don't bother with it.
		return;
	}
	else
	{

		RCRep=RCPlayerReplicationInfo(OtherPawn.PlayerReplicationInfo);

		if (RCRep==none)
		{
			return;
		}
		else
		{
			if (RCRep.currentLap == NoLaps)
			{
				//Set a winner if it's the final lap.
				SetWinner(OtherPawn.PlayerReplicationInfo);
			}

			else
			{
				//Add a Lap to the Tally
				RCRep.currentLap++;
				}

		OtherPawn.ClientMessage("Laps: " + RCRep.currentLap);
		}

	}
```

The if/else statements within if/else statements can look *very* confusing, but get used to nested flow control, especially with what we have to deal with in the next part. Fortunately most code editors conveniently highlight companion brackets.

More to the point, though, this little modification will help us keep unwanted actors from being processed. It's best as you write code to be mindful of what you need versus what you *don't* need and to be prepared to trim out information like this to maintain the efficiency of your code *and* the integrity of your game rules.

Part 5 - Rule Refinements

We have a custom gametype; we have laps tallying up when we pass our goal; and we have custom rules and player data to keep track of it all, but this just *barely* passes mustard. There's still a lot of issues we have to work out if we want to make a proper racing game. Namely...

- The player should have to go *one way* around the track to get a lap. Going back and forth over the goal trigger shouldn't work.
- Vehicles should work with our goal trigger.
- We aren't actually keeping track of who's in what place in this game.

Each of these problems requires something more comprehensive than our current code provides. As with all the other tasks we've had before us it's best to divide and conquer, so let's start with the simplest task first--let's get what we already *have* working with vehicles.

Getting Vehicles Working with the Goal; Comparing Classes

We're tackling this first because we don't need to change all that much; we just have to check and see if the player passing through our goal is in a vehicle, and then pass the player *inside* the vehicle along to our GoalTouched function instead of the vehicle *itself*.

Referencing a pawn inside of a vehicle is a little bit of a challenge, not because it's so hard in itself--it's just that much of a challenge to find out whether the object passing through our trigger *is* a vehicle or not. To do this we need our goal trigger to compare classes, and the syntax for that isn't as immediately clear as everything else we've been working with. If we look back at our gametype script, though, and look at the PlayerReplicationInfoClass that we assigned in DefaultProperties...

```
PlayerReplicationInfoClass = class'UTRaceGame.RCPlayerReplicationInfo'
```

... we see a means of using class *names*, specifically, as variables. Likewise, all Unrealscript objects handily come with a parameter called "Class" which you can get like any other variable and which returns the Class name of whatever Actor you're getting in the script. This will let us add some flow control to the GoalTrigger script. For example:

```
if(Other.Class == Class'Pawn')
```

```
	{
	UTRaceGame(WorldInfo.Game).goalTouched(Other);
	}
```

You're probably thinking that all you have to do is check to see if the object passing through is of the class, "vehicle," like this:

```
if(Other.Class == Class'Vehicle')
	{
	UTRaceGame(WorldInfo.Game).goalTouched(Other.Driver);
	}
```

Unfortunately, this **doesn't work**--for a variety of reasons.

First, in this piece of code we're forgetting to cast the Actor "Other" as a vehicle. It should more rightly look like this:

```
... goalTouched(Vehicle(Other).Driver);
```

Even then, this doesn't work--because **no object, ever, is going to have the class "Vehicle."**

To explain: Every vehicle in Unreal Tournament, and very likely every Vehicle you'll ever work with, whether it's one you created yourself or one someone else made, is extended into its own Class from the Vehicle Class. Even though every vehicle should be recognizeable as a member of the class "Vehicle," we can't actually detect this with the method shown above.

The "Class" comparison is ridiculously specific about this because it's comparing the strings that make up the objects' class names. Therefore, we have to get the **exact** class name of whatever object is passing through. It's not enough to say "Vehicle," you have to name the exact *type* of Vehicle.

In our case, we really only have two vehicles to reasonably choose from--the Scorpion and the Manta, both from Unreal Tournament. UDK also comes with the Cicada, but that one doesn't tend to lend itself to racing too well, so we're going to ignore it. If you recall from our section on the folder structure of UDK, "UTGame_Content" contains all the actual, physical in-game objects for things like vehicles, so it's here that we find the classes we seek: "UTGame_Manta_Content" and "UTGame_Scorpion_Content." If we simply plug these two into our flow control--and yes, we do need to use *both,* it's that specific--we should be able to make this work.

```
if(Other.Class == Class'UTVehicle_Manta_Content')
{
UTRaceGame(WorldInfo.Game).goalTouched(UTVehicle_Manta_Content(Other).Driver)
;
}
else
if (Other.Class == Class'UTVehicle_Scorpion_Content')
{
UTRaceGame(WorldInfo.Game).goalTouched(UTVehicle_Scorpion_Content(Other).Driv
er);
```

```
}
else
{
UTRaceGame(WorldInfo.Game).goalTouched(CheckpointNum, Other);
}
```

This Doesn't Work Either

Unfortunately, we *still* aren't done. For whatever reason, "Driver" doesn't return a pawn the way we'd *like* it to. As a subtype of UTVehicle and not *just* a Vehicle, the pawn that Mantas and Scorpions give us are specifically cast as a UTPawn, meaning that we're not getting the type of pawn that our gametype takes or any of the data that comes with it, so our gametype *never* writes to the player data. Therefore, we have to go over the system's head a little.

Pawns, Controllers, and Drivers; Here's what ACTUALLY Works

Every Pawn and every Actor that's "pilotable" by the player has a "Controller" assigned to it. This includes vehicles as well. Therefore, we can get the Vehicle's Controller, and then we can get the Controller's pawn. So instead of using "Driver" in our flow control above, we put in something more along the lines of:

```
UTVehicle_Manta_Content(Other).Controller.Pawn
```

This gives us the more *general* definition of Pawn--IE, whatever kind of pawn our game has with whatever kind of replication data it's got attached to it. You will note that in all of the other script we've shown, I've been using "Pawn" instead of "UTPawn," and it's for this very reason.

It's a crazy workaround to have to do when one would think that the call for "Driver" was made *exactly* for this sort of thing, but if we compile this script and run it, our vehicle should work just fine running through our goal trigger and the player inside will be declared the winner.

Setting Up a Checkpoint System

Now that that's done, we need to actually enforce the direction of our track. To do this we're going to modify our Goal Trigger and turn it into a checkpoint. We'll then do a bit of magic to insure that the player has to run through every checkpoint *in order* before the final goal will work and generate a new lap.

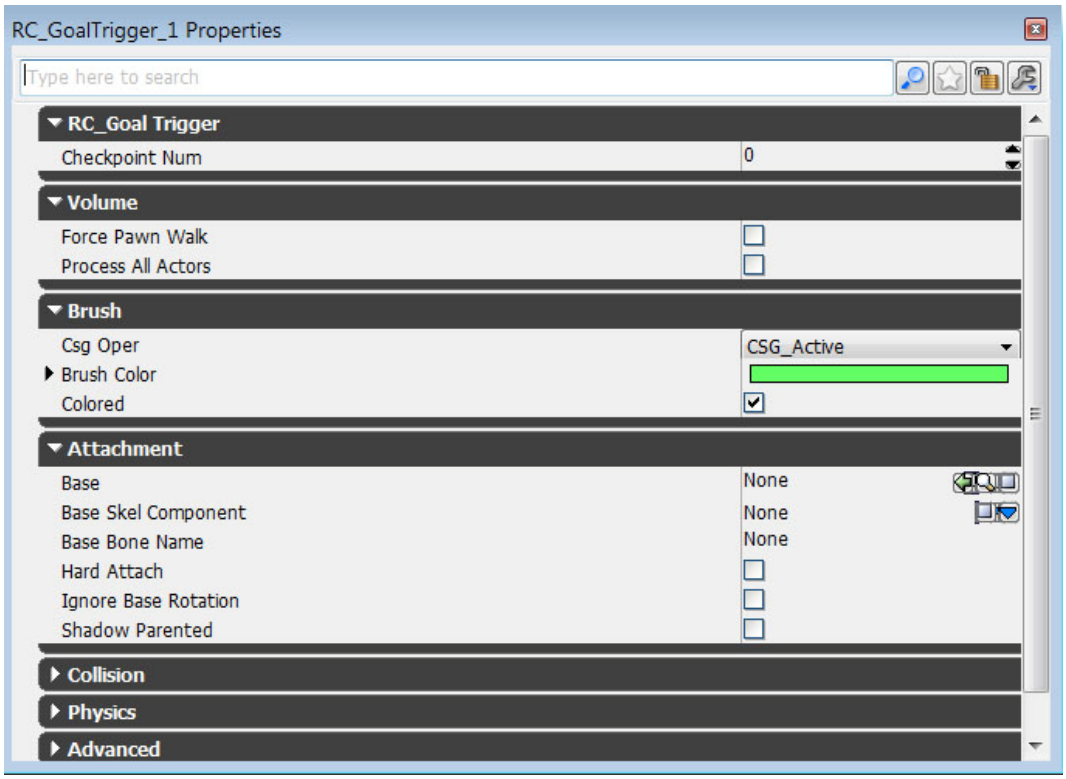
Adding Checkpoint Number in the Level Editor

To do that we have to have a desired order for the player to go through checkpoints, which is something that can't be set programically; only the level designer knows what order checkpoints should be set up in. Therefore, we need to provide the level editor with the ability to set that order.

All we need to do this is a single variable in our GoalTrigger Class, which we're going to call CheckpointNum, giving it a value of "0" in the DefaultValues section. We're going to declare it a little differently, though.

```
var() int CheckpointNum;
```

The parentheses in front of the "var" declaration indicate that this variable is to be exposed and configurable. Just make a Goal Trigger in the UDK editor and bring up its properties dialogue and you'll be able to edit this variable directly in the level builder.



Ideally, all your track designers have to do is number the checkpoints in ascending order from 0 up; our Gametype script will do the rest!

Using an Array and "foreach" to Track Checkpoint Order

Let's head back to the Gametype script so we can add that functionality. We have a complex task ahead of us now, having to both make sure that the Gametype understands what order it's supposed to put the player through the checkpoints in as well as creating even more flow control to make sure players are actually going in that order.

As the second task is dependent on the first, it's the first task we're going to tackle right now. As such, we'll be creating a new function, called "PopulateCheckpointArray," specifically made for this task. Once we do that we'll figure out where exactly in the scheme of the game we want to call it. It doesn't need to take in any parameters since it's just arranging information, so we won't give it any.

```
function populateCheckpointArray()
{

}
```

First, we need an array to store our checkpoints--however many there are in the level, so it will have to dynamically fill itself, and we'll have to always be able to access it. Therefore, we have to go back to the top of the whole UTRaceGame Class and declare it, with RC_GoalTrigger as its variable type.

```
class UTRaceGame extends UTGame
config(game);

var int NoLaps; //Number of laps to run in any given race.

var array<RC_GoalTrigger> Checkpoints;

function goalTouched(int CheckpointNum, Actor Other)
{

...
```

An array of is necessary so that we can keep track of every checkpoint individually. We can use a simple "for" loop to do this, but only if we know what the last checkpoint is, which we don't unless we restrict how many checkpoints the level designers use--and we don't want to do that.

Fortunately, we can find out. If we check the UDN documentation page we find that Unrealscript has a handy command built into it for just this sort of occasion called "foreach." What this command does is look through all actors of a certain type in the level, going down the list in order of their creation.

So, if we make a "foreach" command that looks through all of our Goal Triggers, it will go from GoalTrigger_0 to GoalTrigger_1 to GoalTrigger_2, et cetera. This is why UDK does not allow designers to edit instance names of objects; it depends heavily on the number that it automatically assigns for functions like this.

By the time this chapter is over we're going to use multiple variations on the "foreach" operation, as it can be made to look for specific classes and types of objects. In this case, we'll call the following in our PopulateCheckpointArray function:

```
function populateCheckpointArray()
{

//Somewhere to store the current checkpoint we're parsing.
local RC_GoalTrigger Checkpoint;

//The number of the last checkpoint in our array. This will give us the
length of the "Checkpoints" array.
local int finalCheckpoint;

//Customary number for doing a "for" loop. We need someplace to store the
number we're counting up every loop, so this is it.
local int i;

//We set finalCheckpoint to 0 so that we have a place to start counting from
finalCheckpoint = 0;
```

```
	//Counts up all the goal triggers in the level.
	foreach AllActors(class 'RC_GoalTrigger', Checkpoint)
	{

	}

}
```

This "foreach" operator looks at every actor in the level, but returns ones specifically of our RC_GoalTrigger class, one at a time, working a bit like a specialized "for" loop that does a lot of heavy lifting for us. Every time it loops, it gives us a new GoalTrigger and assigns it to the local variable name, "Checkpoint," as shown here. Any reference to "Checkpoint" inside the For Each statement will apply to every Checkpoint in the entire level.

Now, we just need to do a quick comparison as our function looks through all the checkpoints. If the current checkpoint that we're looking at has a higher value than "finalCheckpoint," we want "finalCheckpoint" to take the new value. When the For Each loop ends, we'll end up with the number of the last checkpoint in our array.

```
//Counts up all the goal triggers in the level.
foreach AllActors(class 'RC_GoalTrigger', Checkpoint)
{
	if (finalCheckpoint < Checkpoint.CheckpointNum)
	{
		//Updates what the last checkpoint in the array should be.

		finalCheckpoint = Checkpoint.CheckpointNum;
	}
}
```

And now, we have what we need to bring the array to the proper length. A quick "for" loop that inserts spaces until it counts up to the final checkpoint value will suffice quite nicely.

```
for (i=0; i<finalCheckpoint; i++)
	{
		//Inserts a number of spaces in our array equal to the number of
our last checkpoint, giving us a space for every one.
		Checkpoints.Insert(i, 1);
	}
```

And finally, we use another For Each loop to quickly put the checkpoints in their proper places in the array that we just made all that space in.

```
foreach AllActors(class 'RC_GoalTrigger',Checkpoint)
	{
		Checkpoint Array . [The current checkpoint we're looking at . the
number of the current checkpoint we're looking at]
		Checkpoints[Checkpoint.CheckpointNum]=Checkpoint;
	}
```

And now, we have a viable Checkpoint Array...

Populating the Array at Runtime

... if we were actually calling this anywhere in our game. Fortunately, GameInfo-derived Classes have a variety of different functions that call at various states of the game. "PostBeginPlay" is the one we want to pay attention to as it happens immediately after gameplay begins. So, we re-define PostBeginPlay, super it, and add our Populate Checkpoint Array function on top of it.

```
function PostBeginPlay()
{
super.PostBeginPlay();

populateCheckpointArray();
}
```

Done. Now we have the checkpoints processed in order, in a format that our gametype can more feasibly work with.

Player Flow Control

Next we actually have to enforce our checkpoint system on the player. To do so we'll have to open up RCPlayerReplicationInfo again and add a new variable to it. Since we're already getting each player's replication info every time we pass through a checkpoint, this makes it the natural place to store the checkpoint number that defines the player's destination.

```
class RCPlayerReplicationInfo extends UTPlayerReplicationInfo;

//Stores current lap
var int currentLap;

//Stores nect checkpoint the player has to go to
var int nextCheckpoint;
DefaultProperties
{
      currentLap=0;
      nextCheckpoint=0;
}
```

That concludes everything we have to do with replication info for this chapter. Now we just have to get our game updating it and looking at it, and the logic at hand is relatively simple if we just stop and make a list of our conditions one by one:

- The player starts having to go through checkpoint 0, AKA the starting line;
- The player's "next checkpoint" value has to count upwards every time they move through a checkpoint;
- A checkpoint will not trigger unless its current number equals their "next checkpoint" number;
- When the player reaches the last checkpoint in the array, AKA the "finish line," the "next checkpoint" value has to reset back to 0 so that the track can loop.

We have in our GoalTriggered function all the data we need to process this now that we've added these variables and have our checkpoint array. Therefore, all we have to do is add an "if/else" statement tree to compare these values. The logic we're using is a little bit more complex this time, so we need to be more specific.

Keeping Track of Player Placing

Who's in first, second, and third place? This is one of the most important questions of our gametype. While our game doesn't actually need to process this, it is relevant performance information that players would like to have displayed for them, so we need to keep it around in some fashion or another.

The question we need to answer, then, is: how do we objectively measure distance to the finish line *no matter what* our track layout is like? And then, more importantly, how do we order players by that information?

Let's think in terms of what we already have:

- A lap system
- Numbered checkpoints
- A system directing players to specific checkpoints in a specific order

Between those two things we have everything we need. Like with the checkpoint array system before, the logic we have to use is fairly simple if we just take the time to break it down. Think of it in terms of comparing yourself to every other player on the map, in terms of that information:

- If we have a higher lap value, we're ahead of them.
- If we have a higher "nextCheckpoint" value, we're ahead of them.
- If we're closer to the next checkpoint, we're ahead of them.

We're going to use this logic to build an if/else tree that builds an array for us--one that updates itself in real time, putting players in an order in the array that corresponds to their current place in the race.

Now we need to gather up a few ingredients we'll need to make this work.

The Tick Function

W need a special function--another standard function called "Tick" that exists for all Actors, as well as our gametype.

```
function Tick(float DeltaTime)
{
super.Tick(DeltaTime);
}
```

For Actionscript enthusiasts out there, Tick is the Unreal Engine equivalent of the "onEnterFrame" event; those of you who have worked with other game engines might understand this to be the equivalent of the "Update" function. In any case, the name sort of explains it. This function gets called every time Unreal's arbitrary time units do a "tick," which gives us a real-time update. Nearly any actor can call the Tick function, which makes it useful for all sorts of things.

Vectors

To put it simply, a Vector in this context is a three-dimensional variable that stores X,Y, and Z coordinates. They can be used for a variety of other purposes as well, including rotation and velocity. Vectors are used in place of computing X, Y, and Z coordinates separately because they store the same information more efficiently, representing the three numbers as a single variable.

We're going to be collecting three vectors--one for "ourselves," one for whatever player we're comparing ourselves to right now, and one for the next checkpoint we're heading towards, specifically its center point.

To be able to get this reliably we're going to have to return to the GoalTrigger and give it a new function, called "GetCenter." The reason we're doing this is because when level designers edit CSG brushes and the like they can sometimes move them far away from their actual pivot point, so we can't depend on the pivot point for our trigger volume; we want to find the absolute center of its bounding box no matter how irregularly shaped it is, so that's what we're going to do.

```
simulated function vector GetCenter(){
...
```

Now we need someplace to store the value of our centerpoint and someplace to store the dimensions of the brush that we're using to define our trigger volume. This is called a Box.

```
local Box TriggerBox;
local Vector CenterPoint;
```

Now, let's get those dimensions.

```
GetComponentsBoundingBox(TriggerBox);
```

... and perform a quick calculation with our vector.

```
CenterPoint=ActorBox.Min+(TriggerBox.Max-ActorBox.Min)*0.5f;
```

Now we'll have this function return that value. This means that anytime we call "GetCenter()" on a trigger volume, it'll give us that center point value in vector form.

```
return CenterPoint;
```

Now we have the functionality we need to compare distance with a trigger volume. We need one more thing if we want to do it relative to a player's next checkpoint.

Getting Checkpoints by Number Index

Namely, we have to be able to get checkpoints by number. Since our Player Replication Data only stores NextCheckpoint as an integer corresponding to the checkpoint number variable we assigned to it, we can't just call "NextCheckpoint.GetCenter()" or anything like that.

We have to be able to *find* the right checkpoint by that number index. Fortunately we don't need to do anything as drastic as making a new function. By the time we're sorting players by their placement in our race, we already have a populated array of checkpoints in the order that they're numbered. All we need is a simple bit of notation for retrieving from an Array, namely:

```
Checkpoints[RCRep.NextCheckpoint];
```

This tells the array to spit out the checkpoint that's held in the same value as the current player's next checkpoint. If we wanted to grab the third checkpoint in our list, we'd just do this:

```
Checkpoints[3];
```

This is one of the many benefits of having an array at our disposal.

Building The Dynamic Player Placement Array

This is our most complicated If/Else tree yet, but if you pay close attention to the logic detailed above, you should be able to trace it out. First, though, we're going to need some variables to do the heavy lifting for us. We're going to be employing loops once again.

```
//Stores player that our loop is currently looking at.
local Controller CurrentPlayer;
//Stores the current player's replication info as well as another replication
info to compare that to.
local RCPlayerReplicationInfo RCRep, CompareRCRep;
//Where we're going to put our fully sorted array.
local array <Controller> SortedControllers;
//An "i" for counting up a for loop we're using later.
local int i;
//The locations of three respective objects: The current player's pawn's
location, the location of "nextCheckpoint" for the current player, and the
location of the pawn we're comparing the current player to.
local Vector CheckpointLoc, PawnLoc, OtherPawnLoc;
```

It may be unclear what some of these variables are for since we've never quite worked with this many until now, but as we take it step by step, it should become clearer.

First, we go through all the player controllers. In this case, we don't use "allactors," but rather "WorldInfo.AllControllers," which is Unreal's handy shortcut for specifically getting all player controllers on a map.

```
foreach WorldInfo.AllControllers(class'Controller', CurrentPlayer)
{
...
```

Now, we define our player replication info as the replication info for the current player that we're looking at.

```
RCRep=RCPlayerReplicationInfo(CurrentPlayer.PlayerReplicationInfo);
```

If we actually *have* replication data to look at... or rather, if we *don't* have *nothing*...

```
if (RCRep != none)
{
...
```

... then we can start building our array. We start our "for" or "comparison" loop going through our array of players. If nobody's in it, then it does it *once* for "0" and just pops the current player into the array.

Otherwise, it'll count through however many players have been added to it, comparing our current player to each player that's already in the array until we break the loop.

When the loop breaks, we're going to insert a space for the current player that our "foreach" loop is looking at wherever this comparison loop stops counting, putting the player just ahead of the last person it compared them to.

So, let's start the loop.

```
for(i=0; i<SortedControllers.Length; i++)
      {
```

Once this loop starts, we define the pawn we're comparing our current player to right now.

```
CompareRCRep =
RCPlayerReplicationInfo(SortedControllers[i].PlayerReplicationInfo);
```

Now we start going through the cases we defined above.

If our current lap is bigger...

```
if (RCRep.currentLap > CompareRCRep.currentLap)
{
      break;
}
```

... then we break the loop. Since this loop is defined in terms of whoever's already been sorted, the first person in the array is whoever's in first place, the second person in the array is whoever's in second place, et cetera. This means that wherever we break this loop, our player is ahead of that person, and this is therefore as far ahead as they can possibly be in the array. We can then proceed on to adding them in.

Otherwise, if their laps are equal...

```
else if (RCRep.currentLap == CompareRCRep.currentLap)
{
```

We have to go to a deeper level, in another set of brackets.
If our current player's next checkpoint is bigger than the compared player's checkpoint...

```
if (RCRep.nextCheckpoint > CompareRCRep.nextCheckpoint)
{
break;
}
```

Then we're ahead of the player we're comparing to, and we're done with the loop. If they're the same, though, we have to go to an even *deeper* level.

```
else if (RCRep.NextCheckpoint == CompareRCRep.NextCheckpoint)
{
```

Only at this point do we worry about those vectors we defined above, so we define them all here.

```
CheckpointLoc =
Checkpoints[GetCheckpointNo(RCRep.nextCheckpoint)].GetCenter();

PawnLoc=CurrentPlayer.Pawn.Location;

OtherPawnLoc = SortedControllers[i].Pawn.Location;
```

Ift the difference between the player's current location and the center of their next checkpoint is higher than that of the compared player...

```
if (VSize(CheckpointLoc-PawnLoc) < VSize(CheckpointLoc-OtherPawnLoc))
{
break;
}
```

... then we get to the final break in our code. Our loop ends here, we're done comparing.

```
                        }
            }
}
```

We went several layers deep in nested if/else statements to do that, and we're still not out. Hopefully this format will keep it relatively clear how this code was blocked out. More importantly, though, now that our loop is broken and we know where the player belongs in this array, we can make a place for the current player our "for each" statement is looking at, and then add them into that spot.

```
SortedControllers.Insert(i,1);
SortedControllers[i]=CurrentPlayer;
```

In order to access this data *outside* of the Tick function, we need to have another array set up at the top of our gametype Class. I have such an array, called "EveryPlayer," which I re-define here as having the same values as SortedControllers. This is for if I have other scripts that need to access the current player placing.

```
EveryPlayer = SortedControllers;
```

Finally, we close out the first layer of if statements (RCRep!=none)...

```
	}
```

... and we close out our For Each statement.

```
}
```

Now there's one last thing we want to do before we close out the function. We want to be able to display this information in some kind of tangible form, so now that we're done sorting our player controllers, let's make one more for loop before the function ends.

```
for (i=0; i<SortedControllers.Length; i++)
{
RCRep=RCPlayerReplicationInfo(SortedControllers[i].PlayerReplicationInfo);
RCRep.Score=i+1;
}
```

All we're doing is looping up through the sorted list and assigning each player's position in the array as their score. Now when we hit the "F1" key ingame to view the scoreboard, we'll be able to see what place we're currently in. This is a quick and dirty way to represent current player placement, and it's not an entirely accurate report due to the way that the current UI interprets this data, but it gets the job done.

Now, start up UDK, let the script compile, debug if necessary, and give it a try. You should have a fully functional racing gametype!

Conclusion

At this point we now have a fully functional racing gametype! Many other additions can still be made to this system, most particularly a better means of spawning players in vehicles than relying on Kismet, a good respawn system that puts players back where their vehicles were destroyed, and a system that allows everyone to actually complete the race before a winner is declared.

Hopefully those will be added to the "Refinements" section soon, but for right now we want to focus on getting a decent feedback system implemented. Not that the Unreal HUD isn't pretty or anything, but the data it gives us isn't terribly relevant to this gametype. In the next lesson, then, we'll be implementing a custom GUI that will display our racing data for our players, making our game more tangible through stronger feedback.