Chapter 5: Unrealscript Integration with Scaleform

Contents

Introduction 2

HUD Class Organization 2

Initializing Our HUD Through Unrealscript 3

bUseClassicHUD - The Most Important Line of Code in This Tutorial 5

Building New Functionality 6

Setting up our FLA 7

Adding the Lap and Place Counters 8

Advanced Functionality - The Mini-Map 10

Understanding the Mini-Map's Functionality 10

Flash Setup 10

Populating the Mini Map 12

Updating the Minimap by Moving Icons 17

Updating the Minimap by Re-Initialization 19

Conclusion 19

Copyright Michael Scott Prinke 2011

Introduction

In our last chapter we developed a simple Scaleform HUD and imported it into UDK. Now, though, we want to make our UI more robust and we want it to display information pertinent to our custom gametype. That means we have to assign the HUD to the player in code instead of in Kismet, fetch information from both the gametype script *and* the player replication info script, then pass that information into our HUD.

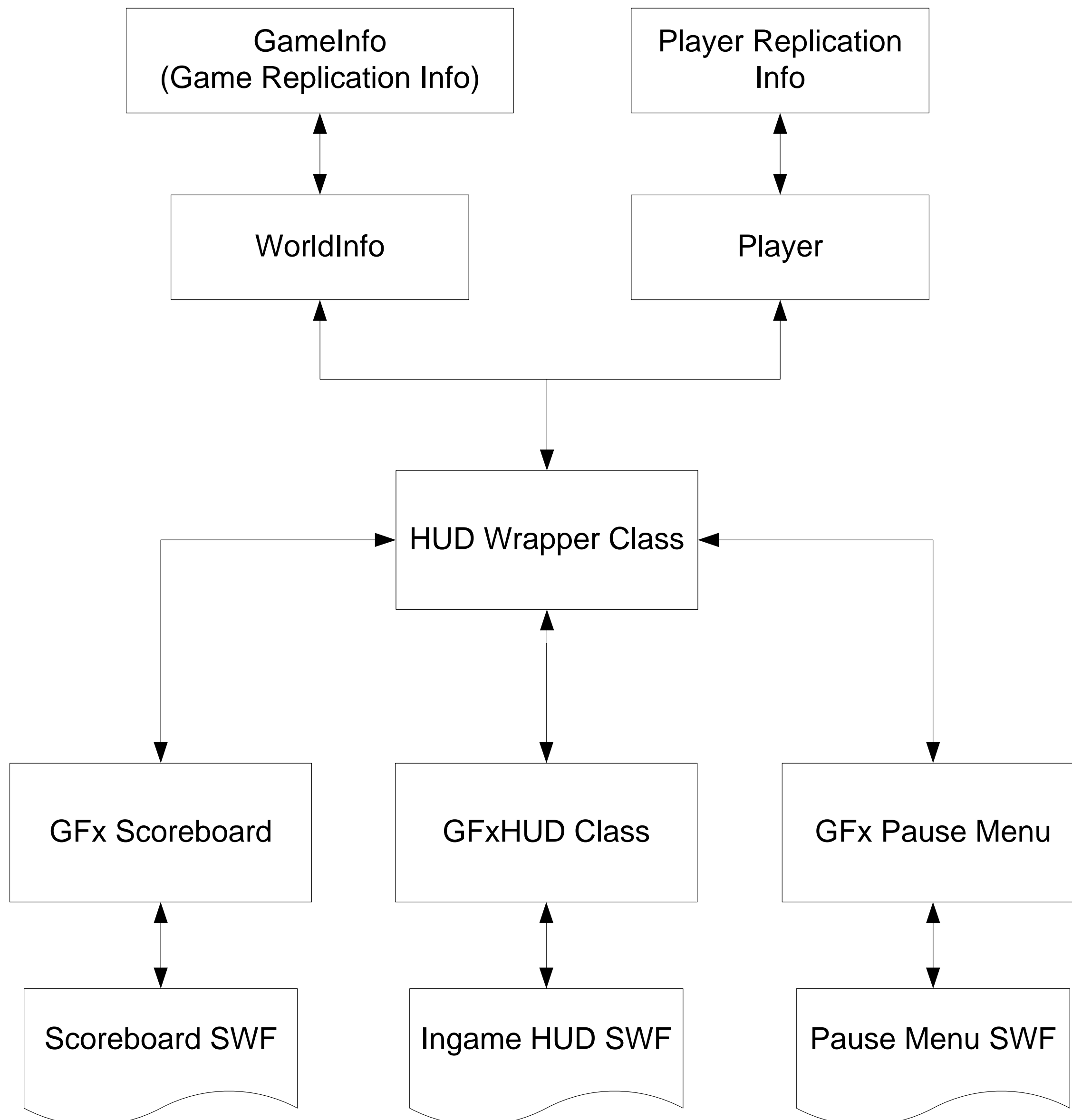
HUD Class Organization

First let's look at how HUDs are organized. There's two classes we want to pay attention to: UTGFxHUDWrapper, and GFxMiniMapHud, both of which are in the UTGame directory.

GFxMiniMapHUD is the Unrealscript that represents the sample Scaleform HUD that comes with UDK and displays ingame. It is responsible for playing the movie and executing all Unrealscript functionality for communicating with the SWF file. Where almost every other class in Unrealscript is an Actor, *this* one is not; rather, it's extended from a separate set of scripts in the "GFxUI" package, namely the GFxMoviePlayer class. Therefore it can't access things that require inheritance from the Actor class. This means that shortcuts like "foreach allactors" which we've been using up until now will not work inside *this script*, specifically. Nevertheless we want to process information here because this script runs *specifically* when our SWF for the ingame UI is running.

UTGFxHUDWrapper is an extension of the Unreal Tournament HUD's base scripts (UTHUDBase). It should be noted that UTGFxHUDWrapper doesn't *just* initialize GFxMinIMapHUD, but also multiple other GFxHUDs, including the scoreboard and pause menus. This can be thought of, then, as the script that acts as the "brain" that controls the entirety of the player's ingame UI, which is stored in separate SWFs and separate Unrealscript Classes per each different UI. This script actually *does* descend from the "Actor" class, and therefore *this one* can access information the way we're used to before passing it along to functions inside GFxMiniMapHUD.

Copyright Michael Scott Prinke 2011



These two classes thus encompass the ingredients we need to get our own HUD working--a HUD Wrapper class and a Gfx HUD class. We could make a scoreboard and pause HUD as well, but I'll leave you to do those in your own time as the processes for importing those and handling them through Unrealscript are practically identical to handling the Ingame HUD.

Initializing Our HUD Through Unrealscript

Start up two new Unrealscript classes. We'll call the first RCHudWrapper, extending it off UTHUDBase to get the functionality that the UT HUD has access to while avoiding the pre-defined functions and files that it's already linked to. The second one we'll call RCGFxHUD, which we're going to extend off GFxMoviePlayer.

Copyright Michael Scott Prinke 2011

RCGFxHUD.uc

Before we do anything, we need to make sure the GFxHUD class is fetching the SWF movie we want it to fetch. We can handle this easily by editing the DefaultProperties of this class.

```
DefaultProperties
{
 //The path to the swf asset
 MovieInfo=SwfMovie'RacingUIAssets.RacingUI'
	bDisplayWithHudOff=false
	bIgnoreMouseInput=true
	bAutoPlay=true
}
```

Here I've defined the path to my SWF asset as being named "RacingUI" inside a package called RacingUIAssets. Additionally, I've set the HUD to display only when HUD is turned on (IE, it won't display in cinematic mode or when the game is paused), to ignore mouse input, and to play automatically.

RCHudWrapper.uc

Next, let's set up the HUD wrapper to activate the GFx HUD. First, let's add a variable to store it.

```
class RCHudWrapper extends UTHUDBase;

var RCGFxHUD HUDMovie; //The name of our GFX UI movie.
```

Next, we need to initialize our HUD. We want it to start up right as the game starts, therefore we go once again to PostBeginPlay().

```
simulated function PostBeginPlay()
{

	// Grab all the normal initialization for the HUD class.
	super.PostBeginPlay();

	//Create a new instance of our custom GFX HUD class.
	HudMovie = new class'RCGFxHUD';

	//Set it to realtime updating.
	HudMovie.SetTimingMode(TM_Real);

	//Calls an initialization function inside the custom GFX HUD class
	HudMovie.Init();
}
```

This gives us the initialization of the HUD and its scripts, but it doesn't actually *draw* the HUD. To do that there's an event we need to rewrite as well, called PostRender, which is the equivalent of tje "Tick" function for other actors.

```
event PostRender()
{
	//Call all the other PostRender stuff from GFxMovie
	super.PostRender();
```

Copyright Michael Scott Prinke 2011

```
        //As long as the HUD is enabled, we want to draw it.
        if ( bShowHud && bEnableActorOverlays )
        {
              DrawHud();
        }

}
```

We haven't yet defined an Init() function, so we'll have to return to the GFxHUD class to do that.

RCGFxHUD.uc

```
function Init( optional LocalPlayer LocPlay )
{
        //Gets all the other intialization stuff we need.
        super.Init (LocPlay);

        //Starts the GFx Movie that's attached to this script (IE: our HUD).
        Start();

        //Advances the frame to the first one.
        Advance(0.f);
}
```

With that out of the way we've got one last old script we have to return to.

UTRaceGame.uc

Add the following to the DefaultProperties for the gametype script:

```
        //Changing HUD to our racing game's HUD
        HUDType = class'UTRaceGame.RCHudWrapper'
```

This controls the HUD that we assign to each player as they join the game, referring to the HUD Wrapper class we created. With that out of the way there's one more thing we need to do before we build our scripts and check to see that our custom HUD works...

bUseClassicHUD - The Most Important Line of Code in This Tutorial

UTRaceGame.uc

```
DefaultProperties
{
bUseClassicHUD = true;
...
```

"bUseclassicHUD" is a holdover from old versions of UDK, back when Epic was still working on integrating Scaleform in 2010. What it *actually* does, when set to false, is **force UDK to only use the sample Scaleform HUD for Unreal Tournament**. Setting it to "true" **tells it to use the HUD you designated**. If we don't set it to "true," our HUD will not work, **ever**, and we'll display the sample Scaleform HUD no matter what we do. This makes it among the most misleading, unintuitive booleans ever devised by modern computer scientists.

Copyright Michael Scott Prinke 2011

Fortunately for future endeavors we don't need to worry about this. bUseClassicHUD only applies to gametypes that extend UTGame. If you're extending UDKGame or GameInfo, all you have to do is define the HUDType.

Now you should be able to build your code. When you start up UDK and test out your gametype, you should find the UT HUD replaced with the new HUD that we developed.

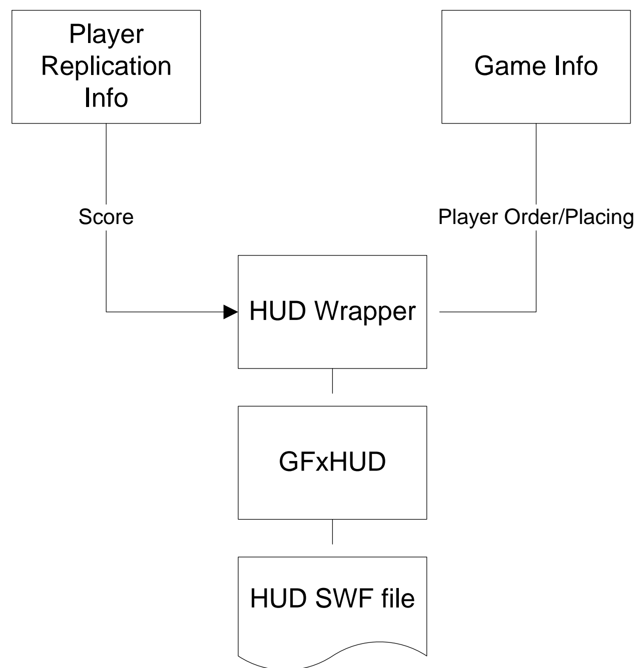
Building New Functionality

Now we want to actually access some data, but first, let's re-collect what our HUD needs to do, what data we need, and where that data has to come from.

We need our HUD to display...

- what place the player is in
- how many laps the player has taken
- a mini-map of the race track, differentiating the player's own icon from other players' icons

We can add other information, of course--a timer, a health bar, weapon displays for our vehicle, but for this tutorial we're just focusing on displaying the racing game's pertinent info. We know where to find this information already because we programmed the scripts that hold it, those being RCPlayerReplicationInfo.uc and UTRaceGame.uc.



As said before, GFxHUD is not extended from any Actor-based class, so it can't access a lot of the information for either our game or our players that we could get in other classes. Therefore we will be

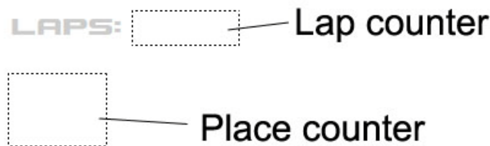
Copyright Michael Scott Prinke 2011

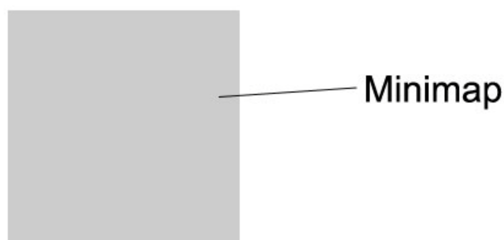
passing information from our Player Replication Info script and our GameInfo script directly into the HUD Wrapper, and *then* we will pass our info into the GFxHUD script. For organization's sake I put the majority of computation in the GFxHUD script; likewise any computations I'd do in the pause menu would be done in the GFxPauseMenu script. This means that the HUD Wrapper will only have to manage these HUDs and pass information to them, and otherwise they will only do computation when active.

Setting up our FLA

RacingUI.fla

First, we need to set up the proper UI elements in our flash file. That means dynamic text for the lap counter and the place counters and *something* to contain the mini-map. If you haven't already, clear away the counter that we made for the last tutorial--it's just cluttering up our FLA at this point.





For reference, the minimap displayed here is a simple grey box that I've converted to a movieclip. The instance names for these pieces are as follows:

Lap Counter: "CurrentLap"
Place Counter: "CurrentPlace"
Minimap Movieclip: "Minimap_MC"

Copyright Michael Scott Prinke 2011

While it may not seem it, making the minimap into a movieclip is going to give us a lot of valuable functions later on.

Adding the Lap and Place Counters

RacingUI.fla - "Actions" layer

Next we'll have to set up the actionscript to be able to update these variables. This is as straightforward as the counter we built before. We need variables to hold the current laps and current place, and on enter frame we need to update our dynamic text with those variables.

```
var Laps = 0
var Place = 0

this.onEnterFrame = function(){
	CurrentLap.text=""+Laps;
	CurrentPlace.text=""+Place;
}
```

To recap, we're just re-setting the text to an empty string and then adding the variables for laps and current place to it. This is literally all we have to do inside the FLA to update these numeric variables. Likewise, updating graphical representations such as health bars requires very little actual scripting work, if any at all, as we can access variables like this directly through our Unrealscript. If we wanted to, we could even manually assign the text for our counters with Unrealscript, but it's easier if we just have it automatically update when the movieclip enters a frame.

RCGFxHUD.uc

Next we have to pass this information along to the HUD. Fortunately the GFXUI classes have a nice shortcut for obtaining the player controller that owns the HUD movie in question: namely, the **GetPC()** function. First, though, we need to get our HUD to start getting ticks in real-time so that we can update our statistics in real-time as well.

```
class RCGFxHUD extends GFxMoviePlayer;

function TickHUD()
{

}
```

The TickHUD function is going to be our name for it. This won't work, however, without some set-up in the HUDWrapper.

RCHUDWrapper.uc

To get TickHUD happening, we're going to re-visit the PostRender event and add a little something to it:

```
event PostRender()//PostRender is the equivalent of a "tick" function for a
HUD wrapper class.
{
```

Copyright Michael Scott Prinke 2011

```
	local int i;
	super.PostRender();

	if (HudMovie != none){
		HudMovie.TickHUD();//As long as we have a HUD, we call the
TickHUD function on every tick.
	}

...
```

TickHUD won't work unless we start it up manually, but thankfully that's an easy thing. We just have to tell our GFxHUD to do the function on every tick via PostRender.

RCGFxHUD.uc

Now we just have to do a little light lifting inside the TickHUD, and we have our variables.

```
function TickHUD()//Calls every tick; we set this up ourselves in the
PostRender function in the HUD wrapper.
{
	local RCPlayerReplicationInfo RCRep;//Stores the current player's
replication info.
	local float thisLap; //Stores the current lap.
	local float thisScore; //Stores the current score.

	RCRep=RCPlayerReplicationInfo(GetPC().Pawn.PlayerReplicationInfo);
	//Gets the player's racing replication info.

	thisScore = RCRep.Score; //Now that we HAVE the player's replication
info, we can update the current score to that which is stored in the
replication info.
	thisLap = RCRep.currentLap; //Same with laps.

}
```

Acquiring the variables is straightforward stuff that we're already very familiar with from the gametype scripting. It's all there for us in the player's replication info, so all we have to do is get at it. As said before, "GetPC" is the function we need in order to ge tthe player this is attached to; what it does, specifically, is return the player controller, so it's no big jump to get the pawn, any info attached to it, and the player's replication info from there. Here I'm casting that info as my racing replication info so that I can get the score and laps from it, which I then assign to the corresponding variables.

Lastly, we need to pass these into the flash file. Fortunately that's very easy for us. At the bottom of the function, all we need is to add the following:

```
...
	SetVariableNumber("Laps",thisLap);
	SetVariableNumber("Place",thisScore);

}
```

There's an entire family of functions related to activating actionscript functions, acquiring actionscript variables, and passing information *into* actionscript variables. Those families are as follows:

Copyright Michael Scott Prinke 2011

- **GetVariable** - For obtaining actionscript variable values.
- **SetVariable -** For *changing* actionscript variables.
- **ActionScript -** For running Actionscript functions.

Each of these comes in specific flavors pertaining to different *types* of variables, and it's best to be as specific as possible when dealing with these functions. For instance, you *can* use "GetVariable()" or "SetVariable()" just by themselves to get any miscellaneous variable, but it's better practice to use "GetVariableNumber()" when you specifically are obtaining a float value, or "GetVariableString" when you're specifically obtaining a string.

```
SetVariableNumber("Laps",thisLap);
SetVariableNumber("Place",thisScore);
```

In this case, I'm using "SetVariableNumber" to set the "Laps" and "Score" values inside the Actionscript to the ones I just got done obtaining from the player's replication info. I name the two variables as strings, then supply the value to fill in. Easy-peezy. Now once we build this script and launch the game, we'll be seeing those statistics updating in glorious real-time onscreen!

Advanced Functionality - The Mini-Map

The vast majority of HUD functionality comes down to simple menu items, timeline manipulation, and responses to variable changes like the ones outlined above. What we're about to tackle next is significantly more problematic--namely, we're going to build a simple mini-map.

Because mini-maps in racing games don't tend to be dynamic maps--IE, the entire track is exposed as opposed to having a "radar" display that scrolls over a larger map--we're not going to worry about making the mini-map scroll as the player moves; it's just going to be a static display of our entire map.

Understanding the Mini-Map's Functionality

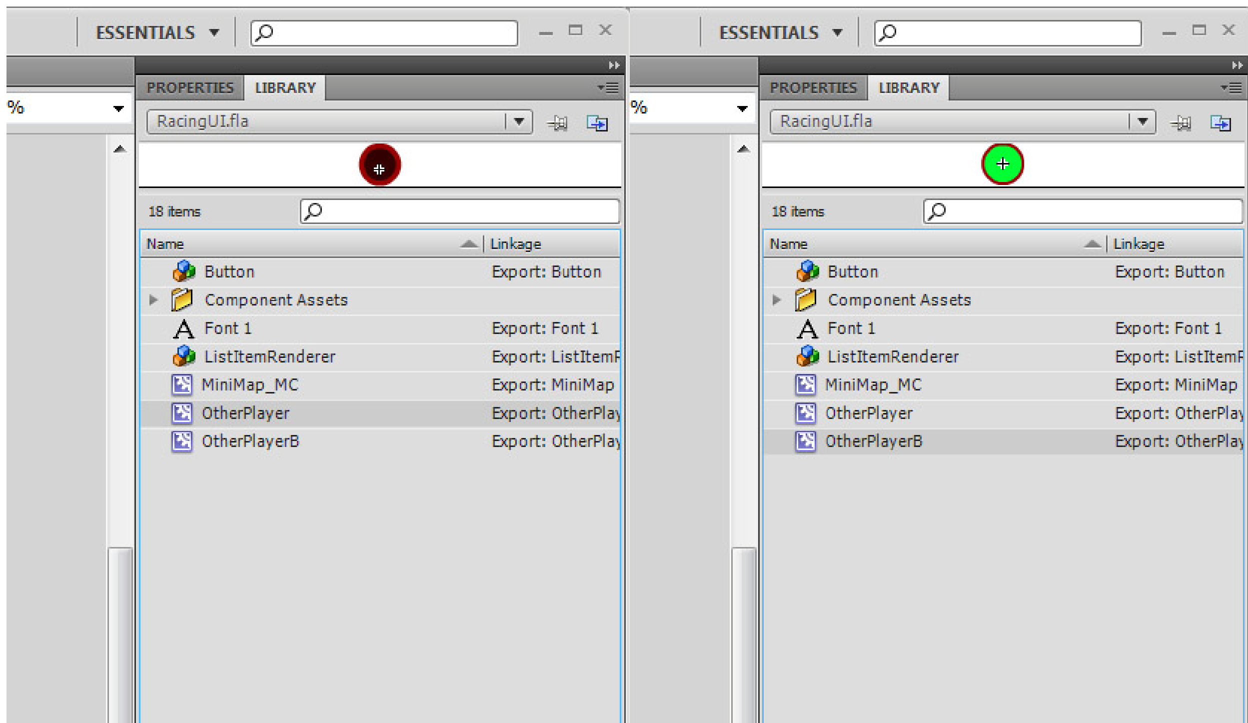
There's many ways we could make this work. Necessarily we'll have to insert the players' blips as movieclips *inside* the movieclip we set aside for the map, but how we move them around in the map is entirely up to our discretion. Again, we are the programmers, and our rules are the only rules that matter. Well. That and not dividing by zero.

Flash Setup

RacingUI.fla

Obviously we have to add a bit to the flash file for our HUD--namely, icons for the players and a method for inserting them dynamically into the minimap movieclip we created earlier. The icons are simple: all I did was make circles of different colors.

Copyright Michael Scott Prinke 2011



You could make them more complex than this--using arrows to provide a sense of direction, for instance--but for right now we'll settle on a large green blip for the player who owns the HUD and a small red one for everybody else. Because of a small shortsight, I've named the one for the player owner "OtherPlayerB" instead of something intuitive like "PlayerOwner," but for the rest of the tutorial we're going to refer to it as "PlayerOwner."

Next, we need a means of adding them into our minimap: namely, a set of functions that our Unrealscript can easily reference. Unrealscript can just reach into the Actionscript and add movieclips straight through the "attachmovie" function, but here I've written a couple of functions for the sake of organization. They are unnecessary, in all likelihood, but I found it helpful to have this condensed down to shorthand as I was learning to put this together.

Actions Layer

```
function InsertPlayer(param1:Number, param2:Number):Void
{
minimap_MC.attachMovie("OtherPlayerB", "firstplayer", 100);
        firstPlayer._x = param1;
        firstPlayer._y = param2;
}
```

Copyright Michael Scott Prinke 2011

```
function InsertOtherPlayer(param1:Number, param2:Number, param3:String, param4:Number):Void
{
	minimap_MC.attachMovie("OtherPlayer", param3, populatelayer);
	NPCPlayer._x=param1;
	NPCPlayer._y=param2;
	populatelayer++;
}
```

"InsertPlayer" here refers to the function that inserts the player owner's radar blip onto the map, while "InsertOtherPlayer" refers to the function that inserts any of the other players; we're going to go through a list of them in our Unrealscript and call this function for each player in the game.

In "InsertPlayer," "Param1" is the X position and "Param2" is the Y position; as far as I know it's entirely necessary to name parameters in this fashion, as it seems that the Scaleform GFx code looks for these parameter names. In any case it's good practice to keep a guide to these parameters in the form of comments. As you can see, all it does is add the Player Owner's blip to the minimap's movieclip, then sets its position to the paramters it took in. The number, "100," is a layer it's being added to. Every time we add something from the library like this, we have to provide a layer. There's no reason I picked 100 here, it just seemed like a good, high number that my counter wouldn't reach.

In "InsertOtherPlayer" we're adding a couple of other parameters. Param3 is the instance name we're giving each blip we insert, and refers to the name of the player, which we pass from Unrealscript to here. "Populatelayer" is, again, the layer it's being added to. You'll need to add this variable to the top of your script and set it at zero. After that, the function will increment it upwards every time it adds a new blip. This is necessary because otherwise "attachMovie" will overwrite all the other movieclips on the current layer whenever it adds a new one.

That's all the code we'll need inside our Actionscript to make this thing work. Now we need some Unrealscript to support it.

Populating the Mini Map

RCHudWrapper.uc

```
//"Populater" function. Counts through every player in our racing game info
class and adds them to the GFX HUD's minimap.
function Populater()

{
	local int i;

//Counts through the "everyplayer" array we made for the racing game info
class.
		for (i=0; i<UTRaceGame(WorldInfo.Game).EveryPlayer.Length; i++)
	{

	HUDMovie.PopulateMiniMap(UTRaceGame(WorldInfo.Game).EveryPlayer[i].Pawn
, i);
	}
```

Copyright Michael Scott Prinke 2011

```
}
```

This new function grabs the "everyplayer" array in the racing game info class for the current game, taking each player's pawn and putting it through another function inside the **RCGFxHUD.uc** class. The reason for this, as I explained earlier, is that RCGFxHUD is derived from the GFXMoviePlayer class and can't actually obtain information from the game, so we have to use the HUD wrapper to pass that information to it. If you remember, we designated HUDMovie as RCGFxHUD.

RCGFxHUD.uc

We need a few new functions to call the InsertPlayer and InsertOtherPlayer functions in the Actionscript. They're very easy to define, merely acting as shells to pass information to the Actionscript.

```
function InsertPlayer(float param1, float param2)
{
	ActionScriptVoid("_root.InsertPlayer");
}

function InsertOtherPlayer(float param1, float param2, string param3, float
param4)
{
	ActionScriptVoid("_root.InsertOtherPlayer");
}
```

We're using ActionScriptVoid because the Actionscript functions we wrote do not return anything. As far as I am aware, the parameters here need to be named "param1," "param2," et cetera, just like in the Actionscript in order for Scaleform to be able to pass them. Any other name, even if it matches between the Actionscript and the Unrealscript, will not work. If this isn't true someone feel free to Email me with a correction, but I wasn't able to get it to work otherwise.

Now we have a couple more functions to make. We can add players to the minimap, but we need to put it in terms of the minimap's coordinates. That means we need to convert the X and Y positions of the player in Unreal space to X and Y positions that match on the map. Fortunately we have the width and height of our map to work with, so that makes the next couple of functions very simple.

```
function float ConvertX(float ConvertA)
{
	local float ConvertedX;

	ConvertedX = (ConvertA/9000)*MiniMapWidth;
	return ConvertedX;
}

function float ConvertY(float ConvertB)
{
	local float ConvertedY;

	ConvertedY = -(ConvertB/8000)*MiniMapHeight;
	return ConvertedY;
}
```

Copyright Michael Scott Prinke 2011

I built my map such that the lower-left corner of it is at 0; thus I have to compensate by negatively multiplying my results by 1 since Flash's coordinate system is upside-down, moving positively as it goes downward.

Note my use of 9000 and 8000 here. I determined experimentally that these were roughly the width and height of my map. Ideally you'd want a more intelligent system for this, but under time constraints I picked the simplest way of doing this. What I'd do instead is create two new actors that I could place at the upper-right and lower-left corners of the map, then make my script run the difference between them on initialization. This is a simple operation--simpler even than the goal triggers--so it shouldn't take you much to figure out how to do it for yourself. You could also create a minimap volume that you can stretch over the map to acquire this data as well. Such a system would enable you to employ this minimap script on any map and not just this one.

The main point is, though, that I'm dividing the coordinates that each of these functions take in by the map's width and height, then multiplying them by the minimap's width and height instead, giving me the coordinates that the player would take at those points.

Now that we have what we need to get the players where we want them, we're going to head back to the Init function.

```
function Init( optional LocalPlayer LocPlay )
{
	super.Init (LocPlay);
	Start();
	Advance(0.f);

	MiniMapWidth = GetVariableNumber("minimap_MC._width");
	MiniMapHeight = GetVariableNumber("minimap_MC._height");

	InsertPlayer(ConvertX(GetPC().Pawn.Location.X),ConvertY(GetPC().Pawn.Lo
cation.Y));

	TickHUD();//Gets the HUD going on its first tick.
}
```

We're adding some variables to the top for the Mini Map's width and height, and in this function we're setting them to the width and height of the movieclip that serves as the minimap itself. If we were using a more dynamic movieclip that pans around a radar like in Grand Theft Auto, we'd use the movieclip that's being masked off and actually moving around underneath its container. In other words, always use the image that's acting as the map for these variables.

Next we're adding the InsertPlayer function, which adds the current player at its X and Y coordinates--but we're converting them before we're adding them. We're doing this now, at initialization, simply because we *can*. The other players are going to be handed with the PopulateMiniMap function we referenced in the HUD Wrapper.

Copyright Michael Scott Prinke 2011

```
function PopulateMiniMap(Pawn PlayerPawn, float Depth)

{
	local float Xposition;
	local float Yposition;
	//Self-explanitory. The X and Y positions of our pawn.

	if (PlayerPawn.Controller != GetPC())
//If the pawn that we pass into this function is NOT the player that owns
this HUD, we go ahead with the functionality.
	{
	Xposition=ConvertX(PlayerPawn.Location.X);
	Yposition=ConvertY(PlayerPawn.Location.Y);
	//Again, X and Y positions for the minimap are converted from the
location of our pawn.

	//We send the InsertOtherPlayer function the X and Y positions that we
want to insert the current player's icon at. We also throw in a string that
contains the current pawn's name, and the Depth value that we got passed from
the HUD wrapper.
	InsertOtherPlayer(Xposition, Yposition, String(PlayerPawn.Name),
Depth);
	}
}
```

To re-iterate the data passed into it: we're putting the pawn data itself in, along with the "i" value to act as the number of the layer we're putting the player on in the map. Then it's just a matter of converting the X and Y position and passing it through the InsertOtherPlayer function.

We need to do a few more things inside our HUD wrapper now to finish initializing the minimap.

RCHudWrapper.uc

Go back to the PostBeginPlay function and add this at the bottom:

```
SetTimer(0.5, false);
```

This sets a timer that delays populating the minimap. This is because the minimap is populating when the map is initialized and all the player pawns are present--IE in postbeginplay, and if we don't delay it by just a fraction of a second, we won't give all the OTHER Postbeginplay functions enough time to populate all the lists we're getting data from. Until I've got a better place to populate the minimap than here, this is a quick fix.

SetTimer references a "Timer" function in your class--which we need to write. It's in this function that we'll populate our map.

Copyright Michael Scott Prinke 2011

```
simulated function Timer(){

      local int i;

           for (i=0; i<UTRaceGame(WorldInfo.Game).EveryPlayer.Length; i++)
//Counts through the "everyplayer" array we made for the racing game info
class.
      {

      HUDMovie.PopulateMiniMap(UTRaceGame(WorldInfo.Game).EveryPlayer[i].Pawn
, i);
      }

}
```

When the timer finishes, the "PopulateMiniMap" function will run through all the players in our game. Simple as that.

Now our minimap is populated, but characters don't move. There's a few ways we could change this. We could clear and re-populate the mini-map every tick, which is probably a valid way of doing things and takes care of quite a lot of problems. From the player's point of view it's indistinguishable from the blips always being present and moving around. Or, we can always keep the blips present, reference them by their names (which we've built in), and then use Unrealscript to push their positions around on each tick.

What it comes down to is which one is more expensive for processing. Not being a computer scientist I'm not aware whether going through a loop to pick out the players each tick and send their positions to their blips or just re-initializing them is faster, but for the player controlling this map it's an easy task to just make it move and *definitely* less costly than re-initializing, so we'll do that first. Head back up to TickHUD().

```
SetVariableNumber("Laps",thisLap);
SetVariableNumber("Place",thisScore);

//The following set the player's position in the minimap.

SetVariableNumber("_root.minimap_MC.firstplayer._x",
ConvertX(GetPC().Pawn.Location.X));

SetVariableNumber("_root.minimap_MC.firstplayer._y",
ConvertY(GetPC().Pawn.Location.Y));
```

What we do next will depend on how we want to add the other players. We can use a loop to move them like this, or we can re-initialize the minimap every tick.

Re-initializing the minimap each tick seems like it should be costly, but consider what we'll have to do otherwise. Every time a player dies, we need to write a function that'll tell the minimap that they died and should be removed. Old players will be moved from where they die to where they respawn, but new ones won't be added. Every time a player enters the game, we'll need to write a function that adds them--otherwise this thing won't know past when the game initially starts how it should add a new

Copyright Michael Scott Prinke 2011

player. Finally, it won't be able to tell when the player gets in a vehicle, which is a big problem for us, so we'll have to add a function that'll tell the map when they've gotten in a vehicle.

Re-initialization, however, will take care of all these problems for us. If the map constantly re-adds players, then new players will come and go easily, vehicles won't be an issue since it will always reference where the current controller's player is instead of just looking for pawns, and dead players will not be re-initialized.

We can elliminate a lot of the cost by pulling out a few variables. If we're re-initializing all the time then we don't ever need to reference blips on the map by name, which means we can drop that out of the InsertOtherPlayer function and fill in gibberish instance names inside the Actionscript instead. We've already written the Actionscript not to accept the layer depth from the Unrealscript, so we can get rid of that too, streamlining the process of initializing other players.

On the other hand, these steps take place essentially at the time that these few aspects of the players' states change, which makes following this path and taking the extra steps markedly more efficient. It's more to code, but should run faster. *Should* run faster. To do this we still need to loop through every single player and check these things, at which point we may as well be re-initialize them. We can code it in at the level of each pawn or player controller's enter vehicle, death, and leave/enter game functions, which would be much more efficient, but that's a little out of scope for what we're doing here.

I'm going to meet you halfway on this. I'll cover how to move characters around on the map, but skip the exceptions above; you should be able to figure those out quickly for yourself. I'll also outline how to do re-initialization.

Updating the Minimap by Moving Icons

```
function UpdateMiniMap(Pawn PlayerPawn)
{
	local float Xposition, Yposition;
	local String Path, PathX, PathY;
```

These are the variables that we'll need: the players' X and Y positions as well as a set of strings to store the path to their blips inside the minimap. The function takes in each individual pawn; we'll set a loop later that'll take in everybody currently in the game.

```
	Path = "_root.minimap_MC.";
```

The path to any given blip on the radar starts out like this, going inside the minimap movieclip.

```
	Path $= String(PlayerPawn.Name);
```

$ is the "concatinate" character when working with strings. We use this instead of using "+" to add two strings together. In this case we're casting the name of the player pawn as a string and then adding it to the end of the last string we made. We've assigned this as the radar blip's name in "InsertOtherPlayer,"

Copyright Michael Scott Prinke 2011

so now we have the player's name. In full, the path should be something like "_root.minimap_MC.Player01" or something like that.

```
	PathX = Path$"._x";
	PathY = Path$"._y";
```

We add the designation "._x" and "._y" to the end to get the X and Y coordinates; thus "PathX" and "PathY" will allow us to quickly set those inside the flash file.

```
if (PlayerPawn.Controller != GetPC())
{
Xposition=ConvertX(PlayerPawn.Location.X);
Yposition=ConvertY(PlayerPawn.Location.Y);
```

As long as the player pawn we have now DOESN'T have the same controller as the one that this GFxHUD is assigned to, we assign the X and Y position variables we have in this function with--of course--the X and Y locations of the pawn in Unreal Space, converted with our custom ConvertX function to the coordinates inside our minimap movieclip.

```
	SetVariableNumber(PathX, Xposition);
	SetVariableNumber(PathY, Yposition);
	}

}
```

To recap; PathX is going to come out looking something like THIS: "_root.minimap_MC.Player01._x" -- So, throwing PathX in the "path" for our setvariablenumber call ends up assigning the X position that we calculated to that pawn's icon in the minimap.

That done, we need to set up the loop to update the minimap. We may as well do this inside the HUD Wrapper since it's the only place we *can* do it.

RCHUDWrapper.uc

Head back to the PostRender() function and add this at the bottom:

```
		for (i=0; i<UTRaceGame(WorldInfo.Game).EveryPlayer.Length; i++)
	{

	HUDMovie.UpdateMiniMap(UTRaceGame(WorldInfo.Game).EveryPlayer[i].Pawn);
//Updates the players' positions on the mini-map.
	}
```

And that concludes our work here. Characters in the minimap will now move according to their position. Again, you'll have to make fixes to insure that the map updates for new players, gets rid of ones that leave the game, and updates for ones inside vehicles, but this is a functional solution for the moment.

Copyright Michael Scott Prinke 2011

Updating the Minimap by Re-Initialization

The more powerful solution involves a little re-tooling to some things we've already written. I will summarize those changes here. You should be able to code them yourself:

Actions

- Don't bother with Param3 or Param4 in the "AddOtherPlayer" function; just have it assign a gibberish instance name based on a number.

RCHUDWrapper

- Instead of using the "Everyplayer" array from the gameinfo script, use the full command "foreach WorldInfo.AllControllers" when populating the map.
- Drop the "Timer" solution for the initial population of the minimap. Instead, put that solution inside PostRender. Since the minimap populates every time the HUD renders, we don't have to worry about that delay.
- The "depth" for each player should re-set every time a population loop finishes. Put in "HUDMovie.SetVariableFloat("populatelayer", 0) inside PostRender after it's done doing a populate loop.

RCGFxHUD

- Don't bother passing Param3 or Param4 in the "AddOtherPlayer" function.

These changes should enable the HUD to re-initialize player markers in the minimap. This is the simplest solution with the least coding and has the added benefit of keeping everybody's HUD consistent with one another.

Conclusion

By now you've got the hang of doing both some simple and advanced things with your HUD, ranging from statistical display to development of a functioning mini-map. Armed with this knowledge you should be able to tackle the needs of practically any HUD you develop for practically any game. There's other HUD extensions we could develop--the scoreboard, the pause menu, et cetera--but you should be able to handle these just as easily on your own and customize them to suit your needs.

Copyright Michael Scott Prinke 2011