

# Chapter 4: Introduction to Scaleform GfX

---

## Contents

- Introduction ..... 2
- What is Scaleform GfX UI? ..... 2
  - Why Use Scaleform? ..... 2
  - Caveat Emptor: On the Efficiency of Programming Languages ..... 3
- Setting Up Scaleform ..... 4
  - Importing Scaleform into Flash ..... 4
  - Setting up our Actionscript ..... 4
    - Directory Structure ..... 4
    - GfX Action Code ..... 4
    - Document Properties ..... 5
- Our First Scaleform Function ..... 5
  - Basics of Actionscript 2 ..... 6
  - Adding the Dynamic Text Box to the Stage ..... 7
  - Embedding Fonts ..... 7
  - Changing the Text Box in Real-Time ..... 8
- Publishing our File to UDK ..... 9
  - Exporting to SWF ..... 10
  - Importing to UDK ..... 10
    - Of Code and Content Browsers ..... 11
  - Displaying our HUD with Kismet ..... 11

## Introduction

A racing game without displays for lap counts, current placing in the race, minimap, and time is indistinguishable from just a bunch of cars going around a track, even *with* the game rules working. That's why we're next going to focus on adding a custom HUD to our gametype. UDK offers us two different systems for building graphic user interfaces (GUIs): Canvas, which is the Unreal Engine's native GUI support, and Scaleform GfX, a more recent addition to UDK that we'll be focusing on for this project. This chapter will be focusing on Scaleform GfX, providing some coding tips for moving from Unrealscript to Actionscript 2 as well as explaining the workflow and setup necessary to get the two communicating. A fully functional, Unrealscript HUD will be the topic of the next chapter.

## What is Scaleform GfX UI?

Simply put, Scaleform GfX is a piece of middleware developed by Scaleform Corporation that enables modern game engines to superimpose a Flash movie on top of normal rendering. It also enables the engine's scripting language to communicate with Flash Actionscript, which means we can build our interface's visual design and animation entirely in Flash Creative Suite to give it a pretty face, add minimal Actionscript for controlling the display, load it into our game, and let Unrealscript take over all the heavy lifting and calculations.

## Why Use Scaleform?

There are a few reasons we're working with Scaleform instead of Canvas.

First, it's the more modern interface solution. Many companies of many different scopes favor it over proprietary solutions, including but not limited to:

- Bioware (Mass Effect, Dragon Age 2)
- Rocksteady (Batman: Arkham Asylum)
- Gearbox (Borderlands)
- Crytek (Crysis)
- Capcom (Street Fighter IV, Lost Planet 2)
- Rare (Viva Pinata, Kinectimals)
- WB Games (DC Universe Online, Mortal Kombat)
- 2k Games (Civilization: Revolution, Civ IV)
- United Front (ModNation racers)

The list goes on. If you look it up, you'll see it was employed in virtually every major title released in 2010, with its usage numbers growing dramatically each year since it was introduced to the game industry. To put it simply, if you want to make game interfaces you almost *have* to be working in Flash and Scaleform.

Second, it provides a stronger sense of control in the long run. While it does carry a bit of a learning curve (unless you already know Flash), the Actionscript we need to learn is actually extremely light and

we don't have to fiddle with the guess-and-check that comes with programically throwing together a HUD with Canvas.

Ordinarily you'd have to load an image and guess at the on-screen coordinates, typing them in with code, re-building, and starting the game, only to see that it's wrong, adjust the code, try again, et cetera. Scaleform lets us develop our HUD with comprehensive visual tools and has all the features that come with Flash, including animations, transparencies, vector graphics, and even some 3D Flash support while also offering a way to preview the UI's functionality without booting up the game. For UI designers this is an invaluable asset that lets them make the most of their skills.

So, even though we *technically* aren't making a whole new game and the content we've developed would be more suited to an Unreal Tournament mod, it's this interface system we're going to focus on. Plus, this will prepare us for when we *do* get into developing a brand new game in the next project.

### **Caveat Emptor: On the Efficiency of Programming Languages**

One of the key guidelines for using Scaleform Gfx is that your Flash files should **absolutely not do any heavy lifting**; they should provide display *only*.

The reason for this is that certain languages tend to run faster than others. There's a couple of phrases used frequently with respect to programming called "low-level" and "high-level." A low-level programming language is closer to the inner workings of the hardware, carrying a big learning curve and being hugely time-consuming as they often require the programmer to *manually* move bits and bytes of data around but offering extremely fast performance and a great deal of control in return. C++ and Assembly are among such languages.

A high-level language is closer to plain English, offering lots of shortcuts, a low learning curve, and a faster development time since it's designed for a human to understand, but at the cost of speed and efficiency as it tends to over-generalize where data goes in the hardware and keeps it around longer than necessary a lot of the time. Java, Python, and Visual Basic are more high-level languages, as are scripting languages for game engines.

Speed is mainly a matter of what type of program is being run through what language and for what operating system, but to give a general idea, Python can sometimes be up to 80x slower than C++, and Java can be around 15x slower. If all you need is a quick and simple program these differences aren't an issue and you get a lot of benefit out of how quickly you can develop with a high-level language compared with something more time-consuming like C++. If you need something complex, though, it gets hairy and starts causing performance problems.

One of the reasons that some of these languages run slower than others is that low-level languages like C++ are "compiled" languages, meaning they get transformed by a compiler into something the computer can understand better and run faster. Scripting languages like Actionscript 2 *aren't* compiled. Rather, they get passed through an interpreter which sorts them out as the Adobe Flash player runs. It is only fair to note that Actionscript 3 is far more stable--too bad UDK doesn't have support for it yet.

To give a baseline for comparison: Epic claims that Unrealscript runs about 20x slower than C++, though I've seen data suggesting it's only about 10x slower most of the time. The version of Actionscript 2 that we're using is anywhere from 30-60x slower or worse; the more complex the task it's given, the worse it is, to the point that it can sometimes be *thousands* of times slower and take entire seconds to do things that Unrealscript will do in milliseconds. That said, it's obviously best not to take the risk of depending on it too much to do heavy lifting for us, especially when it comes to code executing in real-time. We'll be using it to re-size elements in our Flash movies, change dynamic text, provide rollover graphics, and not much else. The reason we do this is to keep our feedback as fast and accurate as possible. If Flash has to both do the math to display a statistic *and* display it, there will be a noticeable delay when the number should update, which is distracting and misleading to players. Just keep the math where it runs most efficiently--inside the game engine--and everything should be okay.

For fuller guidelines on how much Actionscript to use or not to use, check out Epic's Scaleform GfX guidelines.

## Setting Up Scaleform

Setting up Scaleform isn't as easy as just making a Flash AS2 file and sticking it into Unreal. Some special code in AS2 is required, and Scaleform's libraries are needed to make it work.

## Importing Scaleform into Flash

Fortunately UDN has comprehensive instructions on how to do this, such that I have practically no need to write it for you. Everything you need to know to import it and set it up is detailed under the [Scaleform GfX page](#), and all the files necessary to make it run come with your installation of UDK.

## Setting up our Actionscript

When making a Flash file for Scaleform, you'll need to choose "Actionscript 2.0" as your language and Flash Player 8 as your player. Follow UDN's instructions; add the Scaleform GfX extensions and launcher with the Actionscript 2 settings and the Adobe extension manager, then re-start Flash, set the GfX launcher to use the player in your UDK installation.

## Directory Structure

I highly advise putting your Flash project files in the UDKGame/Flash/ directory under a folder named for your project, then setting your publish settings to publish to this folder. In my case, my project is set up in UDKGame/Flash/RacingUI. The reason for this is that this is where UDK looks for SWF files, and is in fact the *only* place where it will accept SWFs when you try to import them into the Content Browser. If you put them anywhere else you will get an error when you try to import.

## GfX Action Code

We have one last step now before we can work with this the way we'd work with any other Flash project. You'll need to create two layers--one called "Actions" to hold the main body of our Actionscript, and one called "GfX Actions" to hold a special three lines of code that'll make this whole thing work. Those three lines are as follows:

```
_global.gfxExtensions = true
```

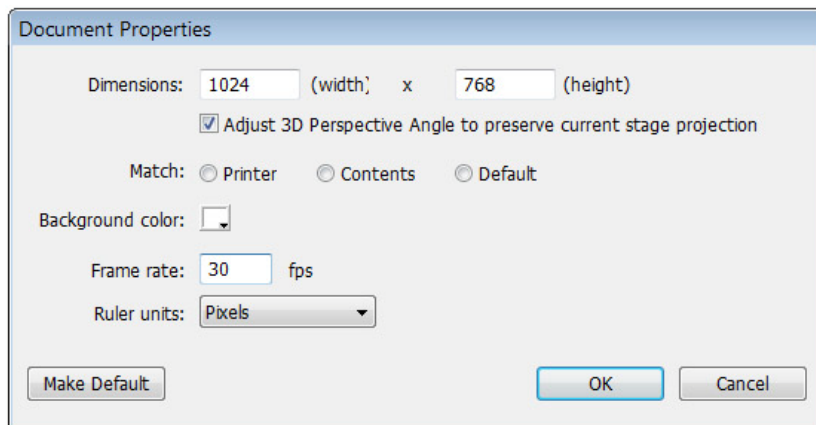
```
_perspfov=25
```

```
stop();
```

Add that under GfX Actions, then just forget about it. From here on out, when you publish a .SWF file of your Flash project it will be able to communicate with Unrealscript and Kismet both via Scaleform's libraries.

## Document Properties

One last thing. We need to set up a couple of our document properties to be appropriate to the screen resolution and fidelity that we want in our Unreal project. Right click the stage, click "document properties," and you'll be able to re-size the stage according to the resolution you want to build at.



In my case I'm building in 1024x768, which is UDK's default resolution, but you can make it anything you like. I've also switched to 30 frames per second to get a little bit more fidelity when my Actionscript updates.

## Our First Scaleform Function

Now, as usual, we need to verify that Scaleform is 1 - running, and 2 - functioning properly in Unreal. To do this we're going to add some Actions to our Flash movie and get a little bit of basic real-time functionality going; namely, we're going to have a timer displaying.

I'm going to assume some familiarity with Flash and its interface already--that you already know how to create dynamic and static text and shapes, how to work with the timeline, movieclips, and other symbols, and that you've worked with enough Actionscript to know how to add actions to the timeline. I'm *not* going to assume that you know Actionscript 2, however, as this tutorial is aimed at educational institutions--which exclusively teach Actionscript 3.

## Basics of Actionscript 2

Actionscript 2, fortunately, is still Java-based, which means that many features familiar to Unrealscript are at our disposal. Functions and variables are declared mostly the same, properties of objects are accessed with periods (firstplayer.\_x), and since we're avoiding anything computationally complex we have the benefit of ignoring the more confusing things that Actionscript 2 would throw at us in favor of the basic programming that's consistent with what we already know. There's a few key differences that you need to be aware of, which I've listed here.

- The main function that updates in real-time is the " enter frame" event, which is declared:

```
this.onEnterFrame = function()
{

}
```

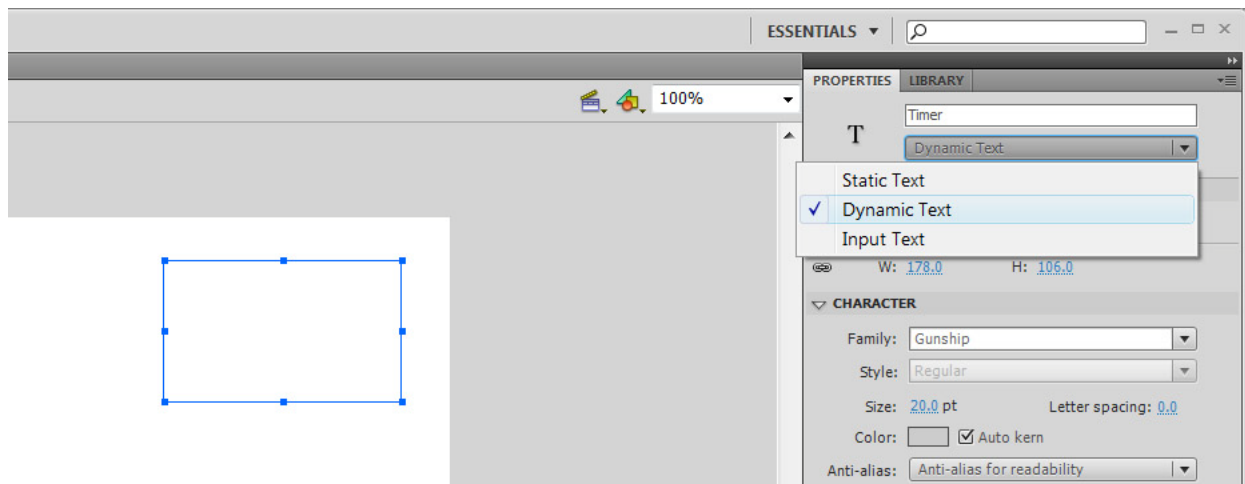
This is the only function that we declare differently from the rest; otherwise the way we've been defining them in Unrealscript still applies.

- Actionscript 2 likes to use the phrase "this" when referring to events. "This" in the preceding example, for instance, refers to the stage and triggers when it enters a frame. If we'd added it to a movieclip's Actions layer instead, it would refer specifically to that movieclip.
- When referencing the stage, the object is called "\_root" in code. So, if we want to get the size of the stage, we'd say "\_root.\_width" or "\_root.\_height". When we're running Flash code out of Unrealscript, bearing this in mind is a good way to guarantee that movieclips we've added get properly referenced.
- Native properties like position and width/height of elements are always preceded with an underscore. Thus the X-position of a movieclip called "PlayerBlip" would be "PlayerBlip.\_x" instead of "PlayerBlip.Position.X."
- Where Unrealscript stores most 3D information as Vectors--IE, three-dimensional variables that can store multiple values in a single place--Flash does *not*, simply keeping things like position and rotation out in the open and directly exposed. In other words, we don't have to go to the trouble of referencing "Player.position.x" like we would in Unreal. Rather, we just go straight to "Player.\_x".
- Where Unrealscript and virtually every other programming language in existence uses the word "float" to define floating point variables, IE variables that have decimal places and not just whole numbers, Actionscript uses word "number" instead. "Int" still refers to "integer."

Bearing this in mind it should be a straightforward transition from Unrealscript to Actionscript and vice versa. With that out of the way, we can finally start adding some functionality.

## Adding the Dynamic Text Box to the Stage

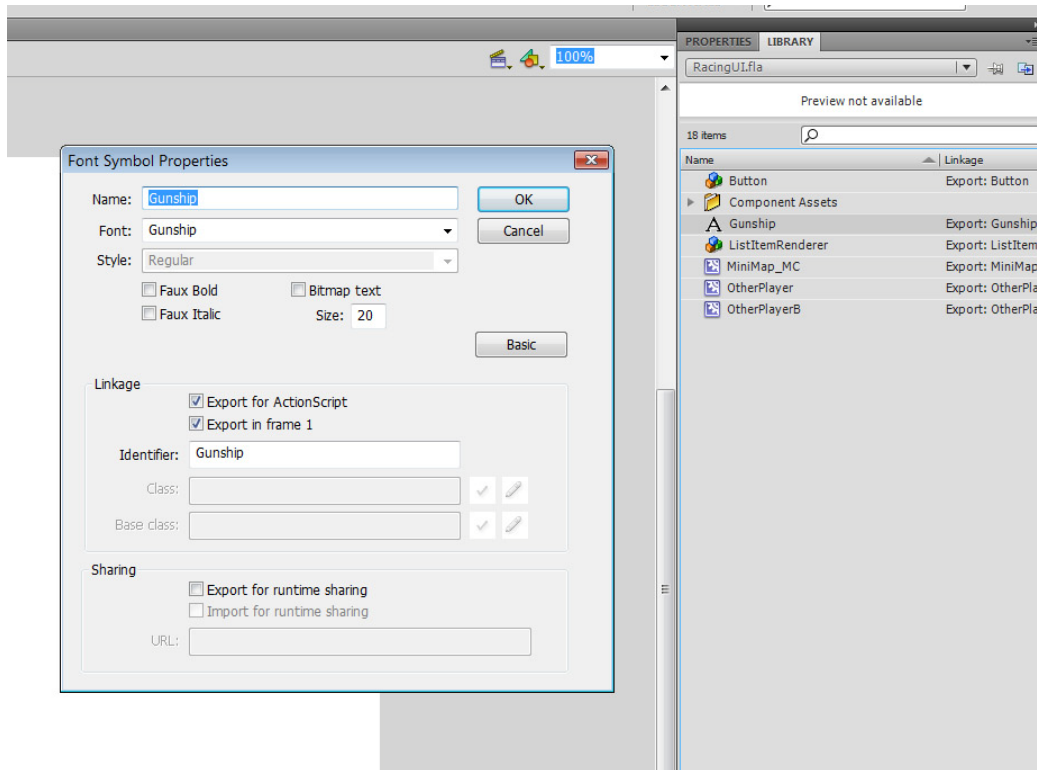
Create an empty text box on the stage. Under the "properties" editor, give it an instance name, "Timer," and switch its type to Dynamic Text. Giving it an instance name is an important step since this is what allows us to reference it in our code. Any time we want to do something to an object, we reference that instance name. Meanwhile, Dynamic Text is something we need in order to be able to change the text with code at all; otherwise it renders out as a static image.



## Embedding Fonts

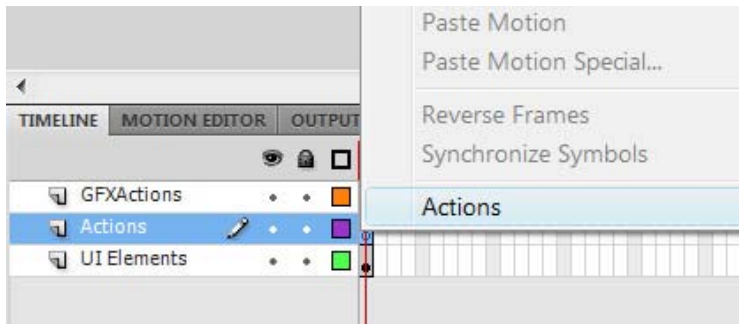
Any time you publish a SWF file, you need to embed your fonts as well--otherwise dynamic text simply will not display when the file is published.

This applies even to basic system fonts like Arial and Times New Roman. Create a new "font" symbol in the Library, set the font to the desired font family (I chose a custom font I downloaded called "Gunship"), name it, check "Export for Actionscript," and "Export in Frame 1," and call it done. Now not only will your font show up regardless of what fonts users have installed on their machine, it'll also display in dynamic and editable text boxes--so long as those boxes are set to use the same font you assigned to your font symbol.



## Changing the Text Box in Real-Time

Bring up the Actions window for the first frame of the "Actions" layer we created on the timeline.



First, add in the following at the top of your Actionscript:

```
var Time = 0;
```

```
Timer.focused = true;
```

The "Time" variable is where we're going to store how much time has passed since our UI was initialized, and "Timer.focused" sets focus on our text box, insuring that it will be updated when our code calls for it. While we can do without setting Timer to "focused," it's good form to set focus on any element that has to update in real-time, especially text boxes. Visually it also makes a handy reference to our text

box's instance name while we're editing code. We can be a lot more efficient about how we set focus, but since the player doesn't directly interact with this interface it doesn't make that big of a difference. Next, we need to add the Enter Frame event.

```
this.onEnterFrame = function()
{

}
```

This is how Flash processes things in real-time--not with arbitrary time units that measure when the data should update, but by the rendering of frames. Any time the program enters a new frame, everything contained in this function will be performed. This is why I'm not using 60 frames per second as the framerate for my Flash project, as Actionscript already chugs enough.

We want our text box to count every time the Flash movie enters a frame, which is a very simple operation.

```
this.onEnterFrame = function()
{
Time = Time+1
Timer.text=""+Time;
}
```

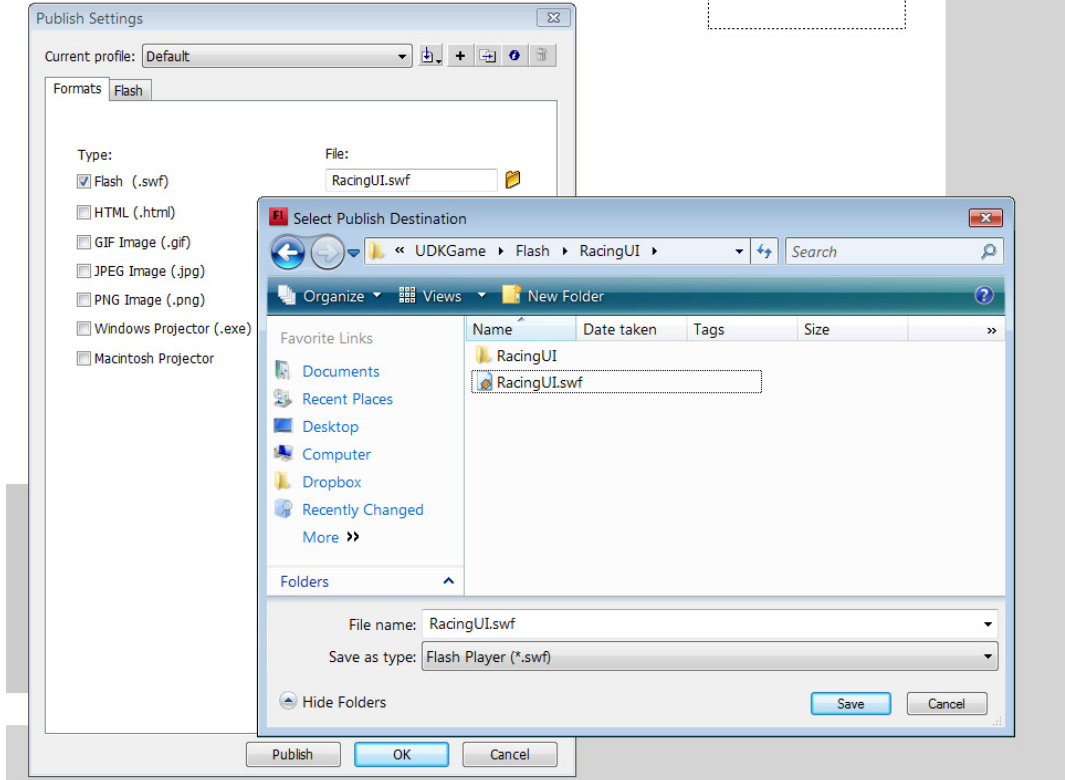
Now, when our movie enters a frame it will 1) add 1 on top of the current time that's passed, and 2) create a new, empty string (""), and then add the current time to it.

Save your file, then hit the "Test with FxMediaPlayer" button in the Scaleform launcher. When it comes up, you should see the dynamic text box you created rapidly cycling through numbers. If not, take care of any errors and double check to make sure that you set up your project correctly, but otherwise, congratulations! You've created your first Scaleform UI.

## Publishing our File to UDK

We've verified that our Actionscript works, but now we have to import our Flash file into UDK and make sure that it works *there*, too, otherwise this is all for naught.

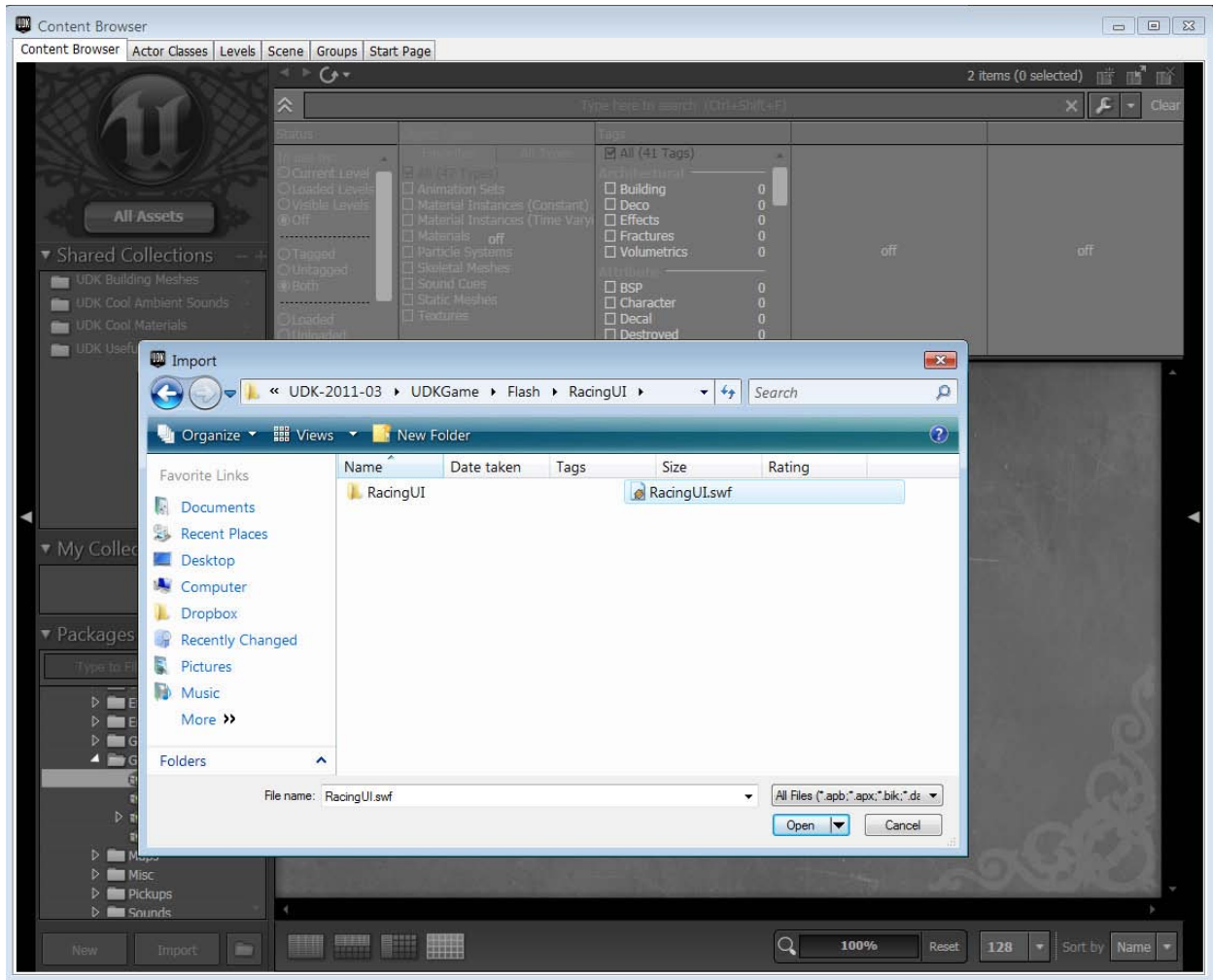
## Exporting to SWF



Make sure in your "Publish Settings" that you've got your publish directory set to "UDKGame/Flash/YourProjectName," as displayed here, as the UDKGame/Flash folder is the only place that Unreal will accept SWF files from. Once you've done this, go ahead and publish your project.

## Importing to UDK

Open up the UDK Editor, open the Content Browser, click "Import," then navigate to where you published your SWF. Give your UI its own package name instead of saving it to the package for any of your maps; that way you'll be able to efficiently load up this UI for any map of the proper gametype.

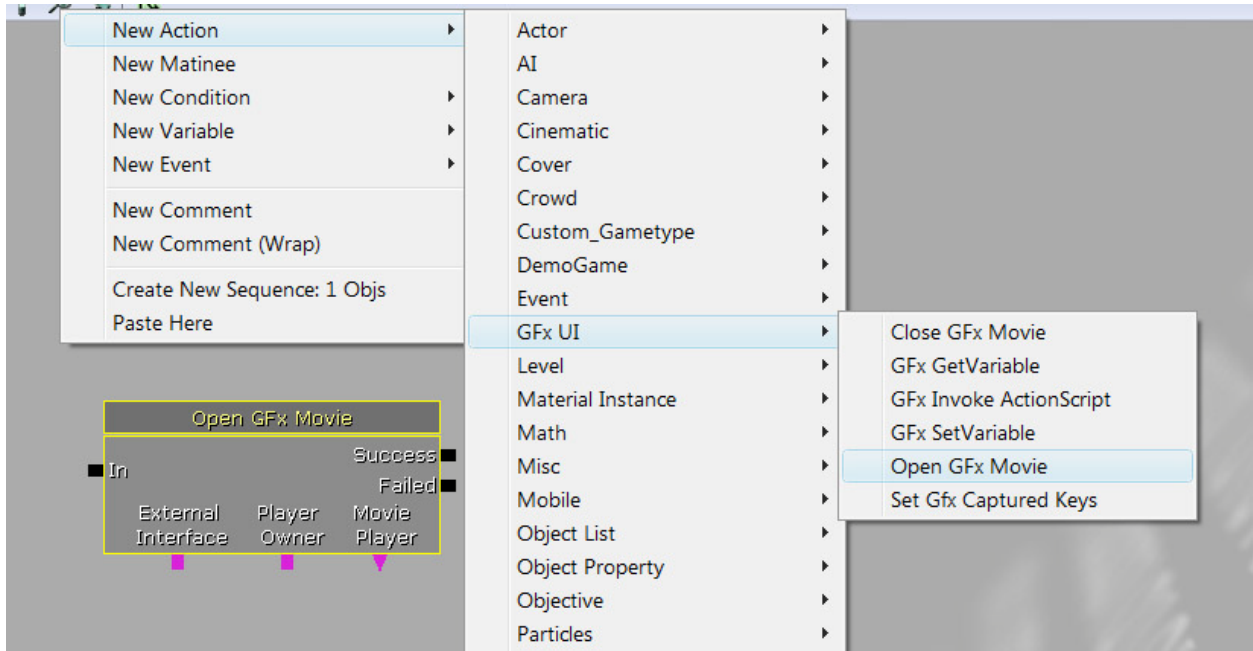


## Of Code and Content Browsers

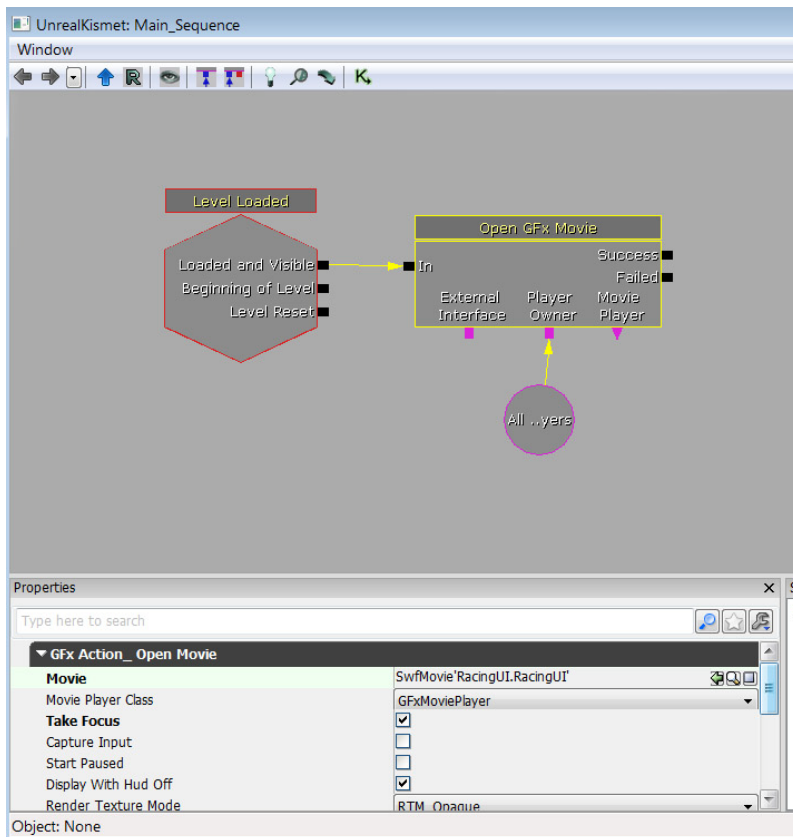
Believe it or not, this is an important step for importing *any* object that we explicitly reference in Unrealscript. Yes, it gets paths to files based on packages that you navigate in the content browser, *not* Windows folders. Remember, any time the UDK editor *itself* references anything it has to be imported and packaged, and Unrealscript is a part of that.

## Displaying our HUD with Kismet

Although we're eventually going to get into setting up our Scaleform HUD to communicate with Unrealscript, we first want to verify it quickly, so we're going to assign it to the player using Kismet. Open up the Kismet editor and create an "Open Gfx Movie" action, as displayed below:



Set it up to open on level start for all players, and in the "Open Gfx Movie" node set the "Movie" field to the SWF we just imported.



Now when you start up your map, your Scaleform HUD should appear overlaid on gameplay in place of the original Unreal HUD, counting up just like it did in the Gfx UI player before. Adding more functionality to the Flash file is simple enough from here on out and you should be able to easily edit your SWF to suit your needs with whatever text boxes and UI elements you need.

Although not an advisable way of publishing our HUD and limited what it can access, Kismet can still pass data in and out of our SWF file and even activate functions inside it with a few simple nodes. For single-player games and UI that's specific to objects and kismet actions in our map that's adequate, but when we have multiple players who have to share interfaces and *some* information between their HUDs--but not *all* of it--this can get very cumbersome, especially when we take things like respawning and resetting the HUD into account. Ideally we want this to be assigned and associated with a player directly, not to have to instance it in the map, manually associate it, and manually set up complicated spider web-like loops in kismet to do what we know a simple "Tick" function can already do. Thus, that will be the subject of the next chapter in this series.