Bonus Recipes

In this chapter, we will cover topics such as:

- Building a Dialog Box using CLIK for use in UDK
- Setting up Dialog Box functionality in Kismet
- Adding ActionScript for keyboard control to a Dialog Box
- Triggering unique dialog by toggling FsCommands
- Adding menu functionality to the Dialog Box in Flash
- Setting up the menu in Kismet with Invoke ActionScript
- Scripting for CLIK components and the importance of Focus
- Creating an animated day-to-night transition

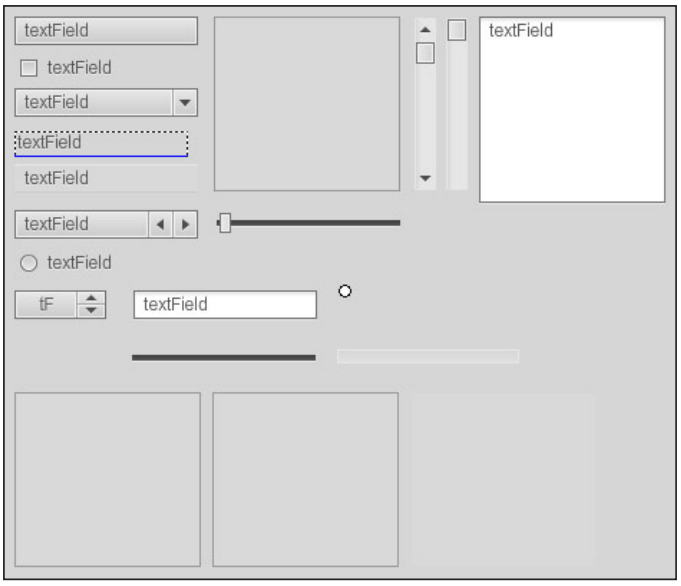
Building a Dialog Box using CLIK for use in UDK

CLIK components are pre-built Flash objects that Scaleform has included for use with UDK. You can adapt them in your custom Scaleform HUDs and menus. They are all the common components that virtually any interactive menu needs, from buttons and check boxes to dynamic text areas, sliders, and loading bars. Savvy Flash users will be familiar with the concept; Flash already has a number of these components available in its own libraries, but CLIK is custom-fitted for Scaleform integration in Unreal and other platforms. In the next image on the left is a CLIK Button, and on the right is the Flash equivalent. The Flash version won't function in Scaleform's media player.

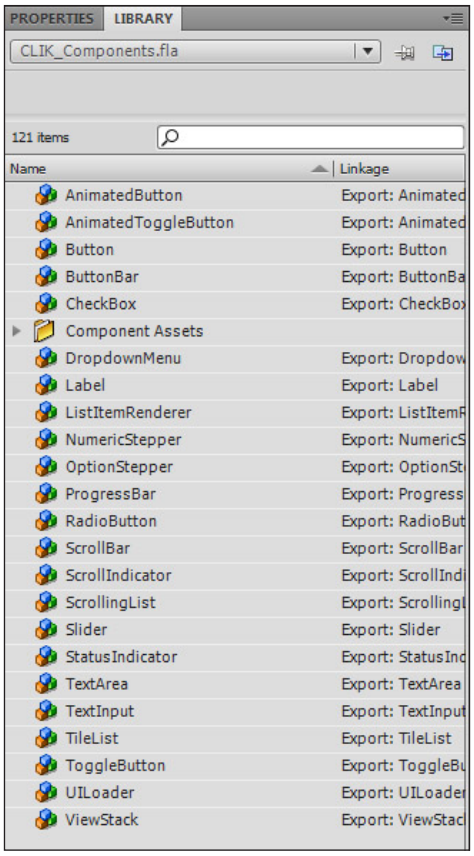
Getting ready

If you haven't set up Scaleform, please review the first recipe in this chapter, *Setting yourself up to work with Scaleform*. Likewise, if you need to, go to **Edit | Preferences** and choose **ActionScript**, then **ActionScript 2.0 Settings**. Click the **+** button to add a new entry, and set it to the CLIK directory, which will be in your `C:\UDK\~\Development\Flash` folder. Make sure it is placed second from the top in this list.

You will then need to add the CLIK components to your library. The easiest way to add CLIK objects is simply to load their original scene, then copy and paste them to another that you are working in. Scaleform has included a sample FLA that includes every functioning CLIK object. You can find this file here: `C:\UDK\~\Development\Flash\CLIK\components\CLIK_Components.FLA`.



As you can see in the image above, which shows the content of `CLIK_Components.FLA`, the CLIK objects are very plain, but they cover the Flash functionality of most of what you'd need to build any kind of menu. All you need to do is copy and paste them into your own FLA file from the Library, which should appear as follows:

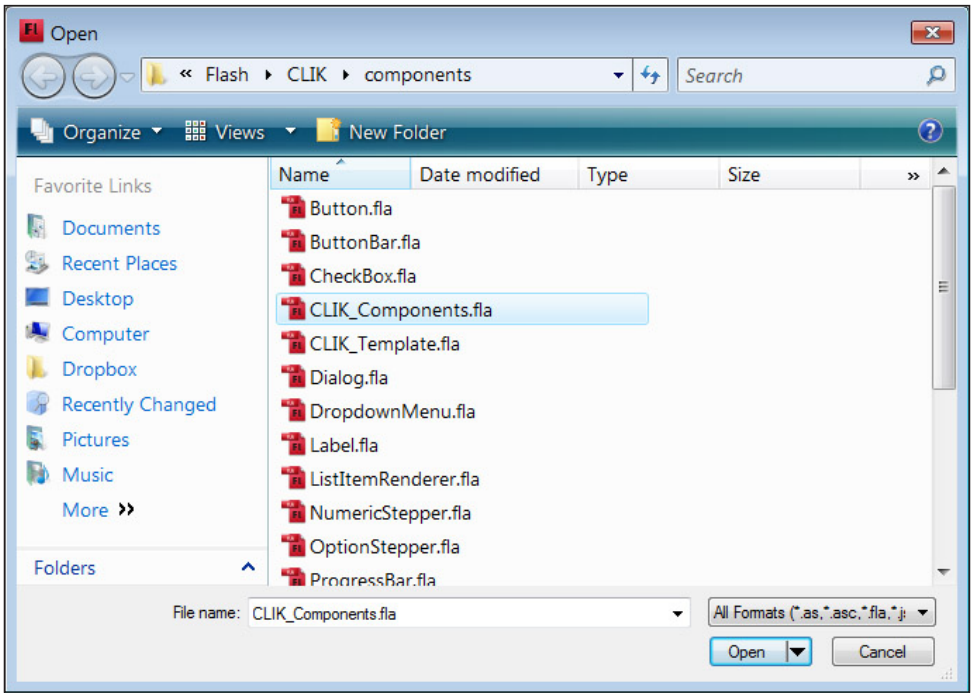


Again, these are very similar to Flash's native set of components. The difference is that the components we'll use have Scaleform's GFx class, so they do a lot of the heavy lifting for you. All you need to do once you've got them working is skin them with new graphics and animations as you please. Unlike with Flash's default components, the visual element and the functionality are separate, and the CLIK object is comprehensively broken down to allow you to easily replace parts.

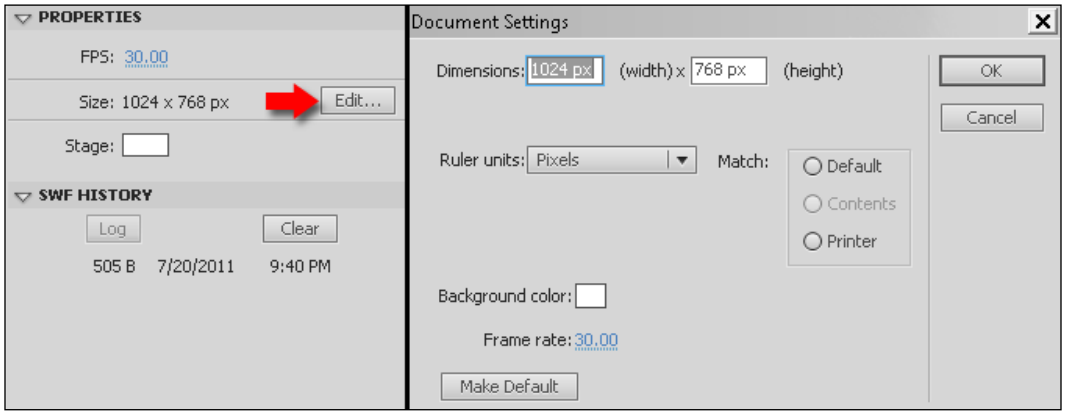
It's easy to build any kind of menu or UI with these components, and it's not difficult to go a step further and make customized, more complex components either; you can build your own **movieclips** just as you would with any other Flash application. Now, though, we want to move on to making these components do some work for us. In this section we're going to develop a simple Flash Dialog UI, establish communication between it and Kismet, and learn how we can use Kismet to access ActionScript.

How to do it...

1. We're going to use dynamic text boxes. **Dynamic text** is text content that can be changed when called by a script. The **Dialog box**, which we're going to control through Kismet, will help us cover a few topics at once: we'll learn how to open Flash SWFs in Kismet, and how to pass information from Kismet to the SWF, as well as the ins and outs of importing SWF movies.
2. Create a new Flash file, being sure to set it to **ActionScript 2.0**. You could also just restart Flash and choose the **ActionScript 2.0** blank stage preset from the prompt. Once started, also make sure the **File | Publish Settings | Flash** tab option for the player is **Flash Player 8**. It is useful actually to make a UDK **Profile** (using the **+** icon along from **Current Profile** at the top of **Publish Settings**).
3. Save the FLA in the `C:\UDK\~\UDKGame\Flash` directory, under its own folder; call it *DialogBox*. It's here that you'll keep all source files pertaining to this Flash project. Be sure to do the same with all Flash files you make.



4. Inside your FLA, go to the **Properties** panel. Click on the stage and you'll see a drop-down that says **Properties**. Here you can set the resolution. Make sure to set the size of the stage to 1280x720 in the **Document Settings** window, as that's the resolution that UDK sets itself to by default.

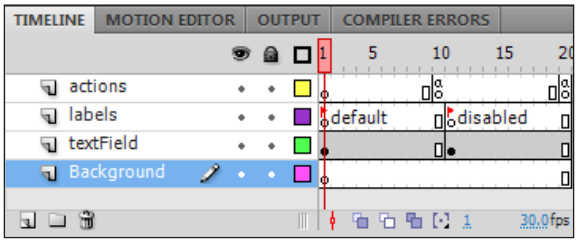


5. Once you've got the base document set up, copy the **Label** component from the *CLIK_Components.FLA* library and paste it into your FLA. Make a duplicate in your library and call this *Dialog Box*. In the **Properties** relating to the object, give the **Label** the instance name *DialogBox*.

For details on duplicating CLIK components, see the last section of the chapter's first recipe 'Setting Yourself Up to Work with Scaleform', which covers CLIK component properties, naming and identifiers.

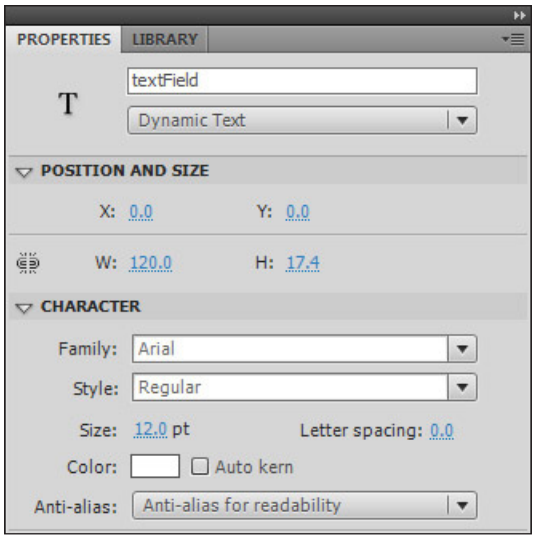
6. Drag it onto the stage, then highlight the symbol and press **Edit**. Select the *TextField* layer. Hit *Ctrl + T* to bring up the transformation dialog. You can click-drag on the elements below to scale it to take up a good portion of the bottom of the stage. If you need to, you can unlink the vertical and horizontal scales by clicking the little chain link icon on the right. After scaling the *TextField* layer, right-click on frame 1, which is all you've changed, and select the key on the same layer under 'disabled' and choose **Paste Frames**. This makes sure both states are the same.
7. While the object may look stretched, when the time comes to run a preview of it you'll see that it corrects itself very nicely. This is thanks to **Scale9grid**, a feature built into Scaleform's CLIK objects that allows them to maintain their visual integrity no matter how you stretch them. For more information on **Scale9grid** look at `http://www.scaleform.com/udkusers` for the PDF on this topic.

8. To make the text easier to read, as the **Label** is completely see-through, double-click on the text field object and within it add a new layer called *Background*, then on that layer draw a simple, dark box shape at frame 1 to show behind the text. You'll need to move this layer down to the bottom of the layers list on the Timeline, as shown below. Fit the box to the text's border using a color that contrasts with the text.

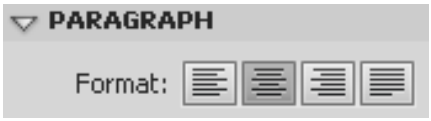


9. Select the text field in the *textField* layer. In the **Properties** Editor there are options for color, font family, and more. Change the font to Arial Regular and make it white to contrast better against the background, as shown here:

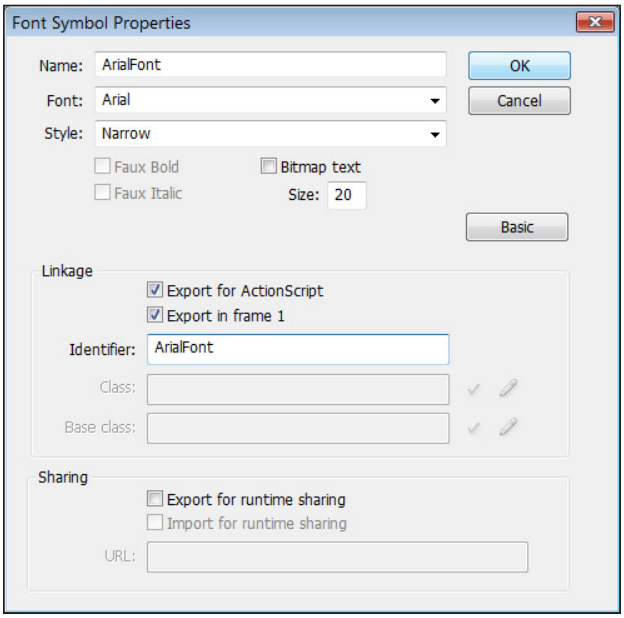
10. The next screenshot shows the dialog for changing the properties of the text field. You can change your font to anything you want, though you'll need to make sure you own the font so as not to run into problems like the *Slate Mobile* error mentioned before. In this instance we're using Arial. The **Color** and **Size** are set in this dialog too.



11. Under **Properties | Character** (where you can see font settings) you'll see section called **Paragraph**. Expand this. Here you can align the text to the center of the dialog box limits by clicking on the **Format** icon highlighted here:

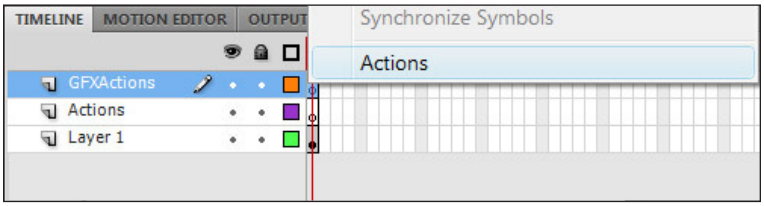


12. To ensure the new font shows up properly outside of our preview, you're going to need to **embed** it in this Flash document. In CS5, when you press **Properties | Character | Embed...** you will notice there are two tabs, **Options** and **Actionscript**. In **Options**, set the **Font** and type in a **Name**, and also tick on **Basic Latin** in the **Character Ranges** list. In the **Actionscript** tab click **Export for Actionscript**. You can also use the main menu option **Text | Font Embedding...** to handle things. The CS4 dialog is shown here:



13. Back at the Stage for our Flash file, click on the **CLIK** component we've just finished modifying and in the **Window | Properties** panel (*Ctrl + F3*) give it an **Instance Name**: *DialogBox*. This will be important later as we'll need a name to reference in order to access it.

14. Go back to the top level of the stage, *Scene 1*. Add two new layers: one called *GFXActions* and one called *Actions*. The following screenshot shows the process for accessing the script window to input script:



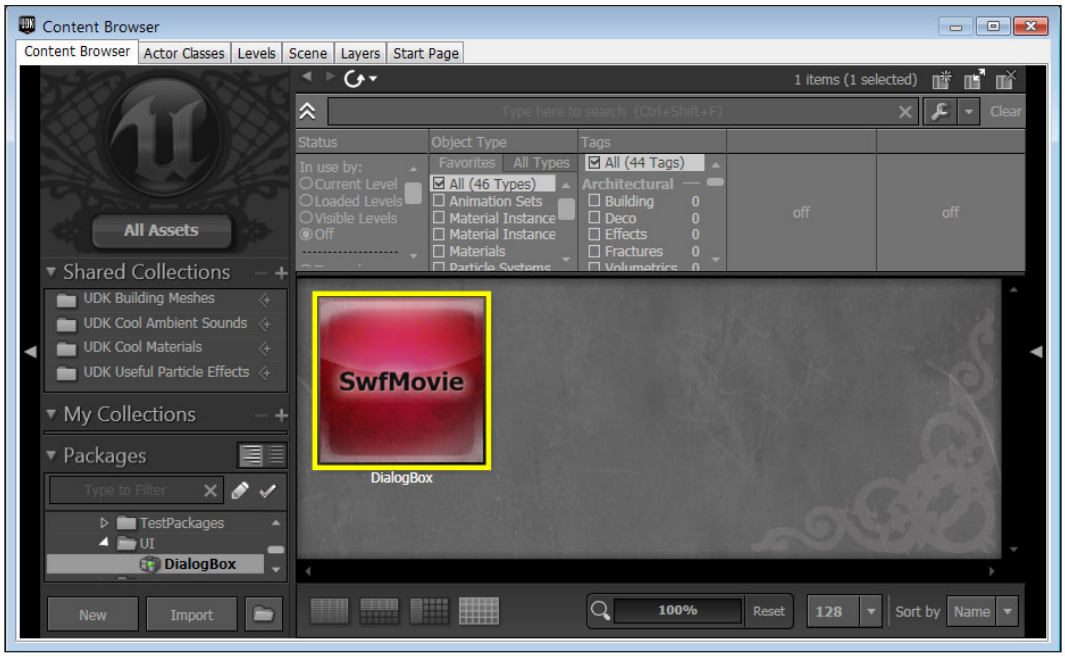
15. Right-click on *GFXActions* at frame 1 and click on **Actions**, and add the following ActionScript:

```
_global.gfxExtensions = true
_perspfov=25
stop();
```

16. Apart from this, keep the layer *GFXActions* clean. Any time you want to add script for the rest, you're going to add it to the *Actions* layer. Fortunately, everything we need is already set up for us. All we have to do is **Publish** (*Alt* + *Shift* + *F12*) our SWF and get it imported into UDK.

17. Save the .FLA, then publish the .SWF, which must be published to the proper folder in UDK's folder structure, otherwise it won't import. Specifically, UDK looks for SWF files in the `C:\UDK\~\UDKGame\Flash\` folder. It is advisable, as always, to keep everything for the current example in its own subfolder apart from other UIs you may develop later.

18. Open UDK. Press *Ctrl* + *Shift* + *F* to expose the content browser. Within your package, right-click and choose **Import**, then locate the SWF in its folder. Accept the default settings for the import. Save the package with the SWF you imported to whatever directory you want, though `C:\UDK\~\UDKGame\Flash\YourfolderUI` would make sense.



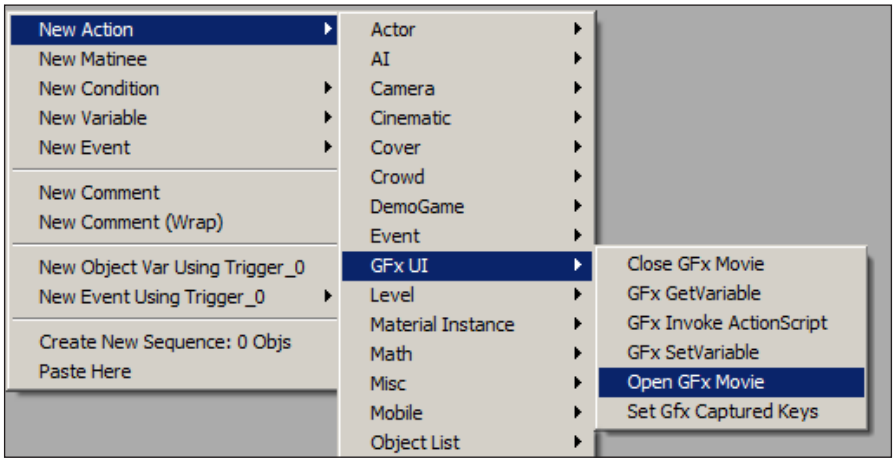
Setting up Dialog Box functionality in Kismet

This recipe continues on from the previous one, in which a customized *Dialog Box* component was based on the default CLIK component **Label**. You can continue from here if you wish using the provided content: *PacktUI.DialogBox*.

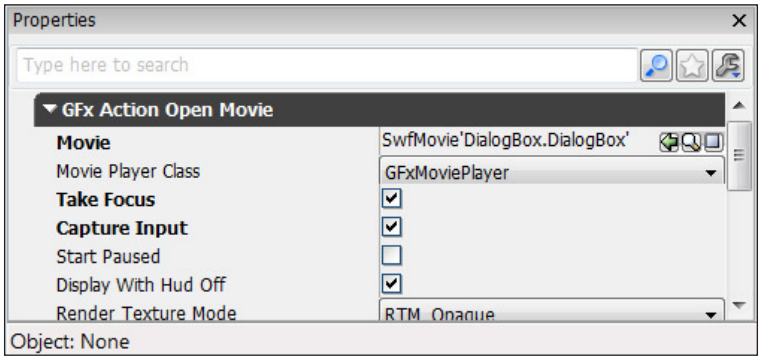
How to do it...

1. Having imported the SWF asset, we'll set up in Kismet an event to open and control the dialog box we made in Flash. Open *Packt_10_PacktUI_Start.UDK* and select *Trigger_0*, as we want to have the dialog box open up when the player touches it.

2. Open up Kismet with the Trigger selected and choose **New Event using Trigger_0 | Touch**. Right-click and choose **New Actions | GFx UI | Open Gfx Movie**. The sub-menu **Gfx UI** contains all the actions that Kismet can perform on a SWF movie, including setting and obtaining variables within ActionScript as well as opening and closing movies.

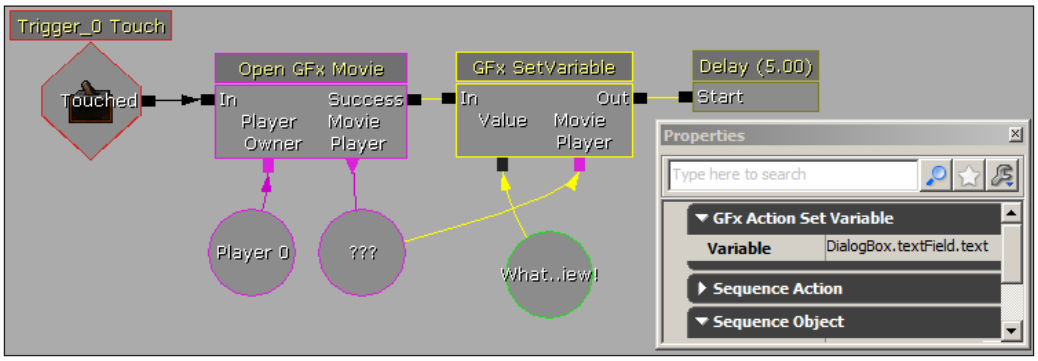


3. In the properties box for the ***Open GFx Movie*** action, find the **Movie** property and load in the *DialogBox* SWF asset from the content browser.



4. In the action's properties, make sure **Take Focus** and **Capture Input** are both checked. These properties tell UDK and Flash this SWF, when fired up, should be what's currently enabled for user input.

5. In Kismet, hold *P* and click to add a ***Player variable*** and in its properties turn off **Use All Players**. Hook this up to the action's **Player Owner** nub. Do not use the ***Player Variable*** as the **Instigator** input for the ***Trigger_0 | Used*** event. Normally we would (outputting comments to the player's log, changing variables on whoever uses the Trigger, and so on) but, with CLIK components, using the ***Trigger_0 Touch*** event's **Instigator** will cause UDK to crash when it tries to load an SWF. The usage is shown as follows:



6. Right-click and choose **New Action | GFxUI | GFxSetVariable**. We can use this action to set any variable inside the SWF, and in this case we're going to set the text that our label is showing. Add a ***String variable***, then attach it to the ***GFx SetVariable*** action's **Value** nub. In the ***String variable***'s properties, set the **Str Value** field to anything; in this example, type *What a wonderful view!*
7. Right-click on the ***GFx SetVariable*** action's **Movie Player** nub and choose **New Object Variable**. Also hook up the **Movie Player** nub from the ***Open GFx Movie*** action to this ***Object Variable***.
8. In the properties for the ***GFx SetVariable*** action, take a look at the **Variable** field. This is where we name the variable that this node is supposed to modify. In this case, we want to modify the **DialogBox** component we created. Specifically, we want to modify the **textField** inside that component. Even more specifically, we want to change the **text** within it. Therefore, type in: *DialogBox.textField.text*.
9. Right-click in Kismet and choose **New Action | GFx UI | Close GFx Movie**. Hold *D* and click to add a ***Delay*** and hook it up between this and the ***GFx SetVariable*** action, and set its duration to 5.0 seconds. The SWF has no means of closing itself, so this allows for that. Now, if you close Kismet and PIE, your dialog box should show prominently, displaying the dynamic text you told it to display.

10. Save your scene, and prepare for the next recipe where we will start using keyboard shortcuts to control the dialog box. An example is shown in *Packt_10_PacktUI_DEMO.UDK*.

Adding ActionScript for keyboard control to a Dialog Box

At this point we've developed the most basic of Flash applications and the most basic communication that Kismet can establish with it. Our dialog box still leaves a lot to be desired though, so we're going to add some controls that players can use to manually advance text and close out the box.

Getting ready

This recipe extends on the previous one on Dialog Box creation. You can use the map *Packt_10_DialogBoxKeys_Start.UDK* to begin here if you prefer.

How to do it...

1. Going back to Flash, open up `\files\Provided Content\DialogBoxKeysSTART.FLA`, which should be set to publish to `C:\UDK\~\UDKGame\Flash|YourfolderUI\DialogBoxKeys.SWF`.
2. Right-click on the first keyframe of the actions layer, then click on **Actions** to bring up its ActionScript. We need to allow our UI to understand keyboard and mouse input, so enter the following to declare a function to handle our mouse input:

```
var mouseListener:Object = new Object;
mouseListener.onMouseDown = function(button, target) {
// LMB
if ( button == 1  ){
fscommand("Next");fscommand("PlaySound");
```

```
}
// RMB
else if ( button == 2  ){
fscommand("Last");fscommand("PlaySound");
}
}
Mouse.addListener(mouseListener);
```

3. This allows left and right mouse clicks to do different things. The FSCommand **Next** means LMB will fire off an event called **Next** in Kismet, and RMB will fire the event **Last**.

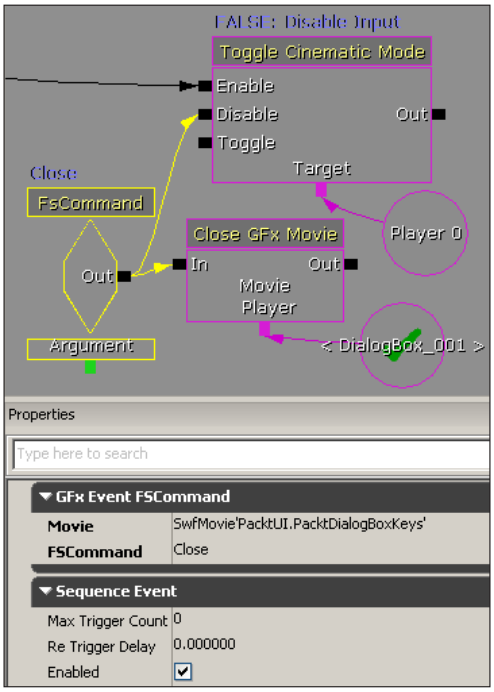
4. `PlaySound` here is just a descriptive name for an ***FsCommand*** event that will be added in Kismet. ***FsCommands*** are just strings that get sent to UDK from your UI. You can either make Kismet respond to these commands (which is what we'll be doing), or you can program responses in UnrealScript. We're going to make `Next` advance text for us while `PlaySound` will play a SoundCue when the player releases the mouse.

5. Before we handle that, we're going to add some extra functionality. **Keyboard listening** works very similarly to the mouse-down function, but it works off what's called **Key Code**. Every key on the keyboard has a specific number assigned to it that Flash and other programming languages can recognize. For example, 40 is the key code for the *Down Arrow* key (you can find some more at `http://people.uncw.edu/tompkinsj/112/flashactionscript/keycodes.htm`), and again, we'll associate it with the ***fscommand*** *Next* to tell Kismet to advance the text. In this way both *Down Arrow* and *MouseDown* handle the same task. Add the following to the ActionScript to use the key codes for *Space*, *Down Arrow*, *Up Arrow*, *W*, and *S*:

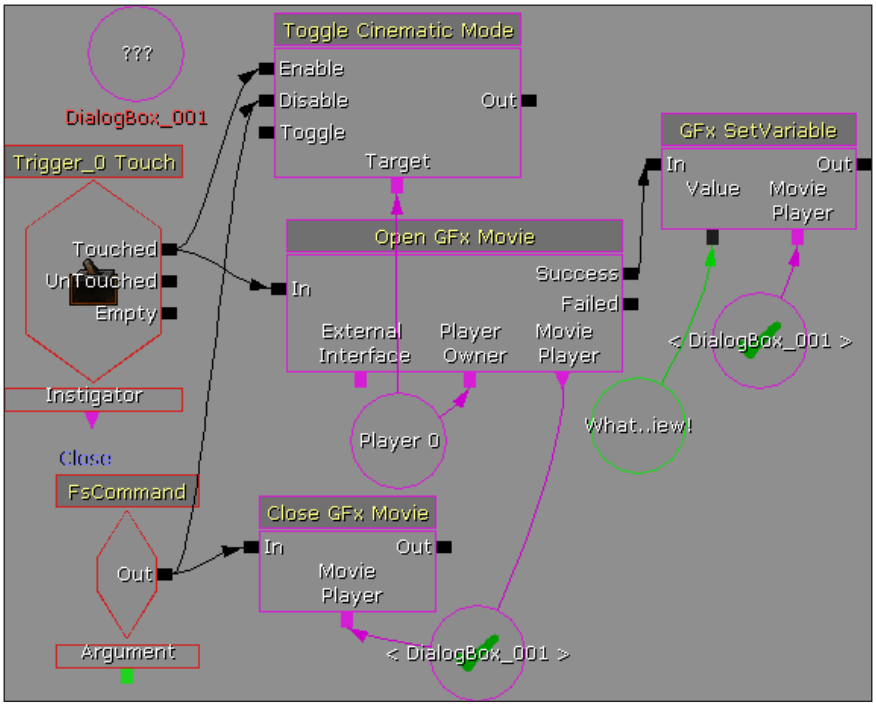
```
//32=Space, 40=down arrow, 83=s key, 38=up arrow, 87=w key

keyboardInput.onKeyDown = function ():Void {
  if (Key.getCode() == 32 ) {
      fscommand("Close");
  }
  if( Key.getCode() == 40 ) {
      fscommand("Next");
  }
  if (Key.getCode() == 83 ) {
    fscommand("Next");
  }
  if(Key.getCode() == 38 ) {
      fscommand("Last");
  }
  if (Key.getCode()==87){
    fscommand("Last");
  }
}
```

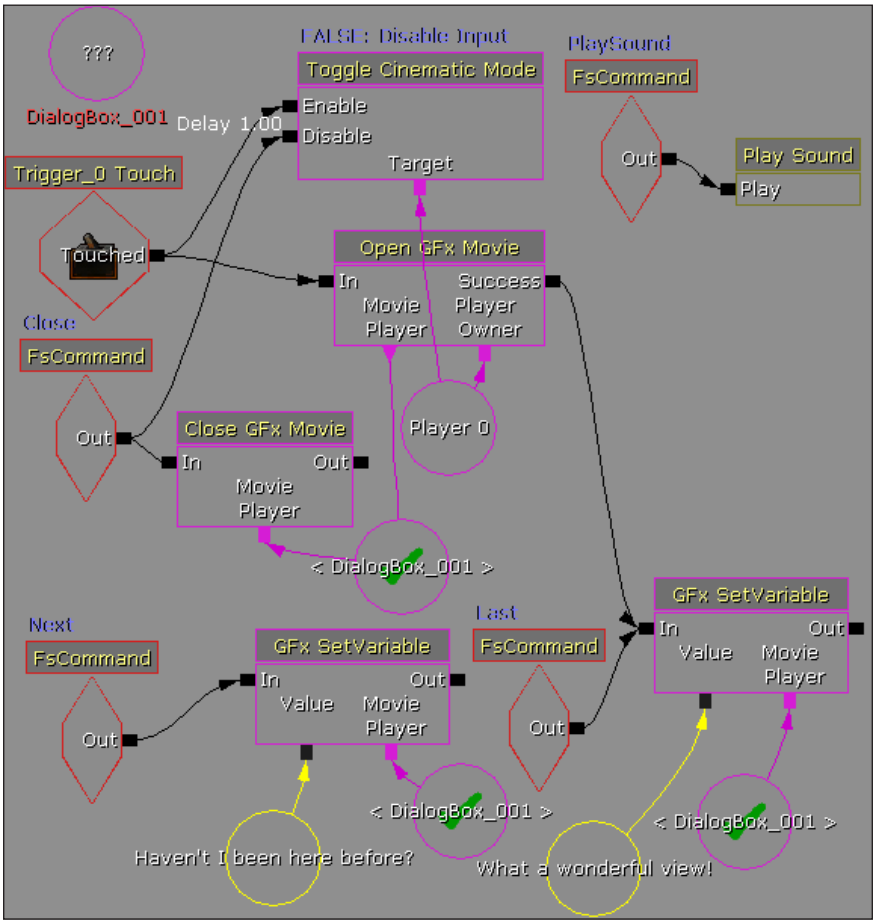
6. What we have allows players to move back and forward through the text we define as well as close the movie. We could use the *Escape* key to close the movie, but since *Escape* is set to exit the UDK PIE session by default, it won't help us very much here, so we need an alternative and *Space* should do fine.
7. Save the Flash file, publish it, and import the SWF into UDK.
8. In Kismet, to prevent messy layout, hold *O* and click to add an ***Object Variable***. In the variable's property **Var Name**, type *DialogBox_001*. Now hold *N* and click to add a ***Named Variable***, and in its **Find Var Name** property also type *DialogBox_001*. Hook up copies of this ***Named Variable*** respectively to the **Movie Player** nub of the ***Open GFx Movie*** action and to the **Movie Player** nub of the ***GFx Set Variable*** action.
9. Add a ***Toggle Cinematic Mode*** toggle to keep the player from moving when the ***Gfx Movie*** has been activated. This is optional, but probably makes for a better user experience. Connect it's **Enable** nub to the ***Trigger_0 Touch*** event's **Touched** nub. Set its **Target** as the ***Player Variable***.
10. Right-click and choose **Events | GFx UI | FsCommand**. Add it in and take a look at the properties. In the movie channel, designate the *DialogBoxKeys* SWF from the content browser so that it knows what SWF to get commands from. Type *Close* into the **FSCommand** field. This matches one of the commands that was defined in the ActionScript for the *DialogBoxKeys* in Flash. Hook up the event to the ***Close Gfx Movie*** action as shown in the next screenshot. It should also hook up also to the **Disable** nub of the ***Toggle Cinematic Mode*** action.



12. Your network should now look something like this:



13. Right-click and choose **New Event | GFx UI | FsCommand** and associate it with the *Next* command in its **FSCommand** property, which gives a new string to the dialog box. Set what you like for the second string text. Finally, add another ***FsCommand*** event and associate it with the *Last* command in its **FSCommand** property. This will just hook up to the first ***GFx SetVariable*** we defined, along with its first input. Now you should be able to control your UI with keyboard and mouse input. This makes for a much more useable and intuitive solution.



Triggering unique dialog by toggling FsCommands

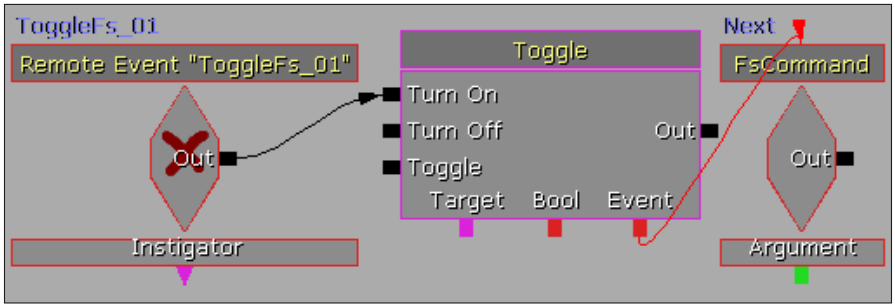
As you can probably tell looking at the last image, the dialog box we worked on in the previous recipe has limited functionality, having only two strings of text. This is mainly because the ***FsCommand*** events will always set the same variables. If we have more than one NPC giving multiple lines of dialog in the same scene, we're in trouble. Fortunately, there's more than one solution to that.

The first solution is to use the ***Toggle*** action, which can turn different Kismet events on and off. Normally it's used for enabling or disabling triggers, but it can also be used to enable or disable specific instances of the ***FsCommand*** event as well. Each instance of an ***FsCommand*** can be activated and deactivated at will and made to fire off different sets of actions.

Therefore, with a little careful linking we can create longer sets of text output, branching dialog options, and, of course, conversations that are specific to individual triggers in our level.

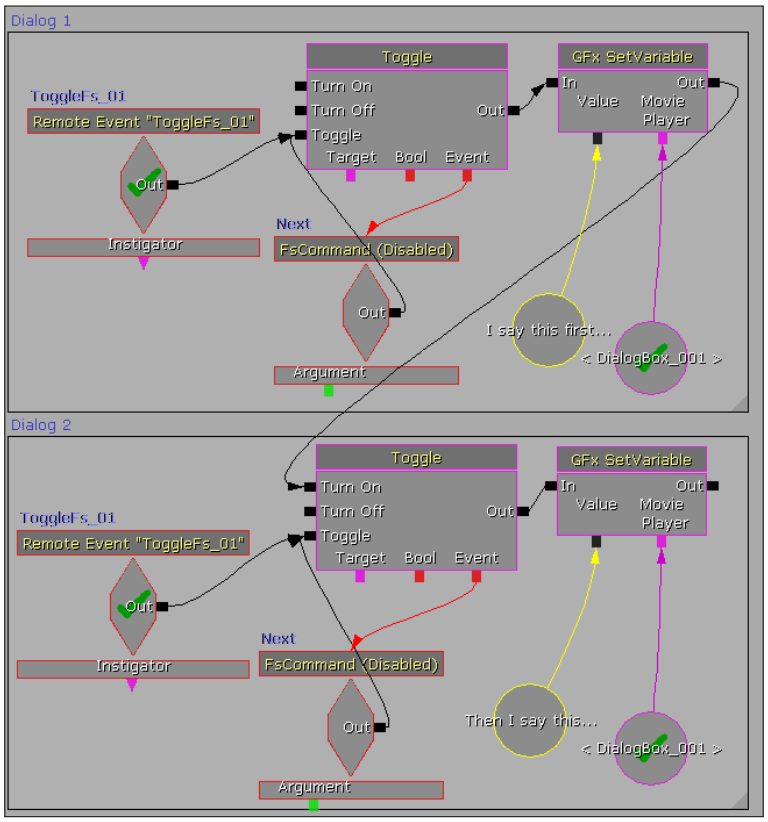
How to do it...

1. In the properties for the ***FsCommand*** actions you created for this conversation, look for their **Enabled checkboxes** and uncheck them. Any time you create a new conversation or text output, ensure that all the ***FsCommand*** nodes are disabled like this.
2. In Kismet, right-click and choose **New Event | Remote Event**. In its **Event Name** property type *ToggleFs_01*. This event will be used to turn on the ***FsCommand*** nodes specific to the conversation we built.
3. Right-click and choose **New Action | Events | Activate Remote Event**. In its properties set the **Event Name** to *ToggleFs_01*. Any time this action occurs in Kismet, it will activate all instances of the *ToggleFs_01* remote event.
4. Hold *T* and click to add a ***Toggle*** action. Note that it has an **Event** nub. Connect it to the ***FsCommand*** *Next*, then connect the input to the ***Remote Event*** *ToggleFs_01***.**



5. Select and copy the ***Remote Event*** *ToggleFs_01* and the ***Toggle*** command you just made, then link them to every other ***FsCommand*** event for this conversation in the same pattern.
6. Connect the ***Activate Remote Event*** *ToggleFs_01* to the trigger event that activates this conversation. Now whenever the player starts this conversation it will enable all of the ***FsCommands*** we created.
7. Duplicate the ***Activate Remote Event*** action, then connect it to the end of the ***Close GFx Movie*** action. Whenever the conversation ends, the ***FsCommands*** for it will all turn off.

8. You can also route one string of text into the upcoming ***Toggle*** action's input to create longer sets of dialogue to display, as shown in the next screenshot. Connect the *Next* ***FsCommand*** event back to the ***Toggle*** action which enabled it, and it will switch itself off. Then connect the ***GFx SetVariable*** action to the ***Toggle*** for another instance of the *Next* ***FsCommand*** event to change the text that it will advance to. All this is shown in the next screenshot, too:



How it works...

The basis of this is that we're re-programming our conversation and text output every time we make a new string of text to display. Turning ***FsCommands*** on and off all the time requires a lot of attention to the overall pattern, but offers us versatility with our Flash interface. Hypothetically, we can now create another, unique conversation simply by setting it up this same way, except that instead of using *ToggleFs_01* as the name for your ***Remote Events***, you would use *ToggleFs_02*. You can name the ***Remote Events*** however you like; after characters, objects in the game, or whatever else is intuitive to you. As long as the remote events that activate and de-activate them have different names, the ***FsCommands*** will advance through different sets of text for each Trigger.

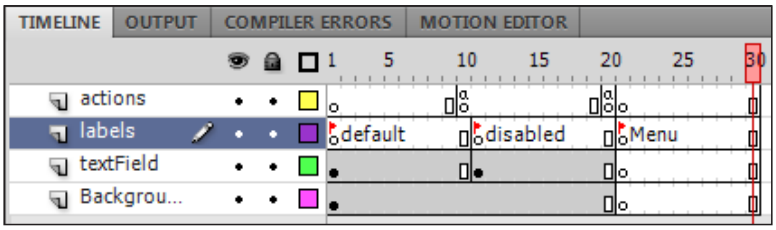
The alternative would be to program a custom Kismet node specifically for controlling dialog, condensing a lot of repetitive node-linking into something that simply displays one string of dialog per node, and looks for our input commands to advance from one node to the next. Although a better approach, that would require some heavy lifting in UnrealScript, and developing this kind of tool is more of a programmer's job than an artist's or a level designer's. For prototyping purposes, though, the solution we have here handles fairly well.

Adding menu functionality to the Dialog Box in Flash

Continuing from the previous recipe, the Dialog Box we've extended through Kismet can now be used to communicate almost any kind of information and initiate dialog with multiple triggers, but our system leaves something to be desired in terms of the player's ability to interact with it. So the next thing we'll add is some simple menu functionality.

How to do it...

1. Open up *\files\Provided Content\DialogBoxMenuSTART.FLA* in Flash, then select and right-click on the *DialogBox* **movieclip** on the stage and choose **Edit**. For all the layers, right-click on frame 21 and choose **Insert Blank Keyframe**. Then go to frame 30 and do so again, then right-click and choose **Clear Keyframe**. This leaves you with a range of new space at the end of the timeline.
2. Right-click at frame 21 in the labels layer and choose **Insert Blank Keyframe**. Then in the **Properties Inspector** on the right-hand side panel of Flash, in the **Label | Name** channel, enter `Menu` to give this segment of the timeline a name. We'll call this the *menu* state from now on.



In the actions layer, copy the keyframe at frame 10 and paste it at frame 30. This is a Stop command.

3. Select the keyframe at the start of the menu state. Since it's immediately following the disabled state, it will inherit the same look; its text will be grayed out and the text box won't have the edits that you did to the *active* state on frame 1, such as the increased font. You can fix this by clearing out the menu state and copy/pasting all the elements from the active state, or just by taking the time to alter it by hand. Either way, take a moment to make the menu state identical to the *active* state.
4. Add in a new set of four **CLIK labels**. Name them *option0*, *option1*, *option2*, and *option3*. Arrange them on the right-hand side of the dialog box and shrink the **label** for the main text display to give them some room. Leave a little extra between the main text box and the newly added option labels for a cursor.
5. Add a new movieclip to your library and use it to make a quick cursor or paste one in from another FLA scene, such as `PacktSlider.FLA`. Ensure that the origin point is in the upper-left corner of the movieclip.
6. Create a new layer in the *Scene 1* timeline called *cursor*.
7. We want keyboard and mouse input to drive this menu, but we don't want to accidentally shoot off the ***FsCommands*** we defined for the default state. Select the *actions* layer inside the *DialogBox* movieclip, right-click on the timeline at frame 21, and choose **Actions** to open up ActionScript Editor. Add the following at the top of the script:

```
stop();
Key.removeListener(keyboardInput);
Mouse.removeListener(mouseInput);
```

8. This will stop the movie on that frame and remove the **event listeners** for keyboard and mouse input that we defined for the default dialog box, allowing us to set up new ones specially for menu interaction.
9. Return to the keyframe at frame21 for the *menu* state. Add the following code by right clicking the keyframe and choosing **Actions**:

```
var currentOption = 0;
var maxOptions = 4;

var keyboardInput:Object = new Object();
var mouseInput:Object = new Object();

Key.addListener(keyboardInput);
Mouse.addListener(mouseInput);
```

10. The first two lines are variables we need to control our menu; the first is the number of the current option we're highlighting, and the second is the maximum number of options that we're fitting into our menu. We're including the `maxOptions` variable in case we want to leave one or more options blank.

11. Now we're going to start taking in the player's input. Define the following for key presses:

```
keyboardInput.onKeyDown = function ():Void {

 if( Key.getCode() == 40 ) { // down arrow
 currentOption++;
 }
 if (Key.getCode() == 83 ) { //s key
 currentOption++;
 }
 if(Key.getCode () == 38 ) { // up arrow
 currentOption--;
 }
 if (Key.getCode() == 87){ // w key
 currentOption--;
 }
 if (Key.getCode() == 13) {// Enter key
 fscommand("menuop_"+currentOption);
 }
 if (Key.getCode == 32) {// space bar key
 fscommand("menuop_"+currentOption);
 }
}

mouseInput.onMouseDown = function():Void{
 fscommand("menuop_"+currentOption);
}
```

12. Now whenever the user presses either the up or down arrow, it will increment `currentOption` up or down by exactly 1. The RMB throws an ***FsCommand*** that combines the string `menuop_` with the number set in `currentOption`. Thus, in Kismet, we'd have the Kismet ***FsCommand*** events *menuop_0* and *menuop_1* each do different things. Take note that here the *down arrow* increments by positive numbers because the options are listed in ascending order, with option0 at the top and option3 at the bottom.

13. Next, we need to make that cursor respond to the user's input. To do this we're going to use an `onEnterFrame` event and a `switch` statement:

```
this.onEnterFrame = function(){
switch(currentOption){
  case maxOptions:
        currentOption = 0;
        break;
    case -1:
        currentOption = maxOptions-1;
        break;
    case 0:
        MenuPointer._y = option0._y;
        break;
    case 1:
        MenuPointer._y = option1._y;
        break;
    case 2:
        MenuPointer._y = option2._y;
        break;
    case 3:
        MenuPointer._y = option3._y;
        break;
  }
}
```

14. A `switch` statement is a simple alternative to an if/else statement, reading a variable (in this case `currentOption`) and acting on multiple cases for what its value is. Take note of the `break;` commands in between each case. This tells the switch to stop what it's doing after executing the command given. If we hadn't done so, it would proceed down through all the cases in the order listed until hitting the end of the switch statement. We have it inside an `onEnterFrame` event so that Flash reads it as long as we're on the menu. Even though the timeline is paused, it will always be updating based on the `currentOption` variable as long as we're in this frame. What we're specifically doing is checking to see what option the player has selected, then putting the cursor at the *Y*, or *up and down* position of the string of text it corresponds to. If it should ever meet the value for `maxOptions` (which we make sure to check **before** reading anything else) it will immediately set us back to the first option. Likewise, if we ever go below option 0, it will set it back to the last option. Now, every time we press *up arrow* and *down arrow* on the keyboard, we'll see the mouse cursor change positions in relation to the option we're picking, giving us clear visual feedback.

15. Clear all of the default text for each of the options, so that they display blank by default.

16. Finally, we're going to start automating this interface so that it automatically knows how many options we've filled in. Find the line we already have: `this.onEnterFrame=function(){`. Add the following code just above it:

```
if (option3.text=="")
{
  maxOptions=3;
}
if (option2.text=="")
{
  maxOptions=2;
}
```

17. If the text for the fourth or third options are left blank, the max options will be set such that neither of them are taken into account. The menu will behave as if they don't exist. This will save us the trouble of setting the maximum number of options manually in Kismet.

18. Save your file as *DialogBoxMenuDONE.FLA*, click on **File | Publish Settings** and set the output path to `C:\UDK\~\UDKGame\Flash\YourfolderUI\DialogBoxMenu.SWF`.

Setting up the menu in Kismet with Invoke ActionScript

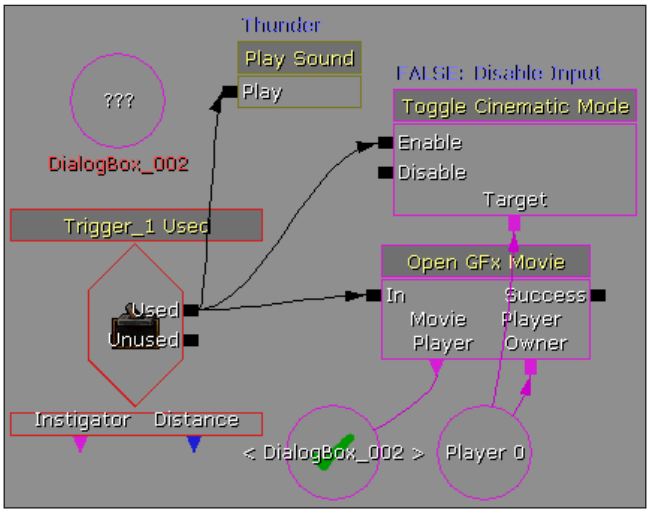
This recipe is intended to add a final layer of functionality to the stages we've been through in the previous recipes. We have a process by which we can enter strings of text into a succession of dialog boxes, but a menu to allow the player to influence the conversation is in order.

Getting ready

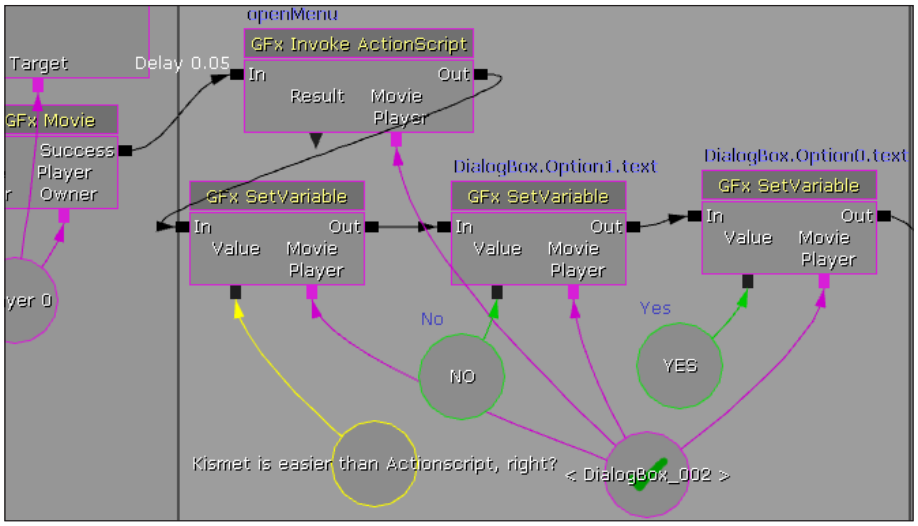
Open the provided map *Packt_10_DialogBoxMenu_Start.UDK*.

How to do it...

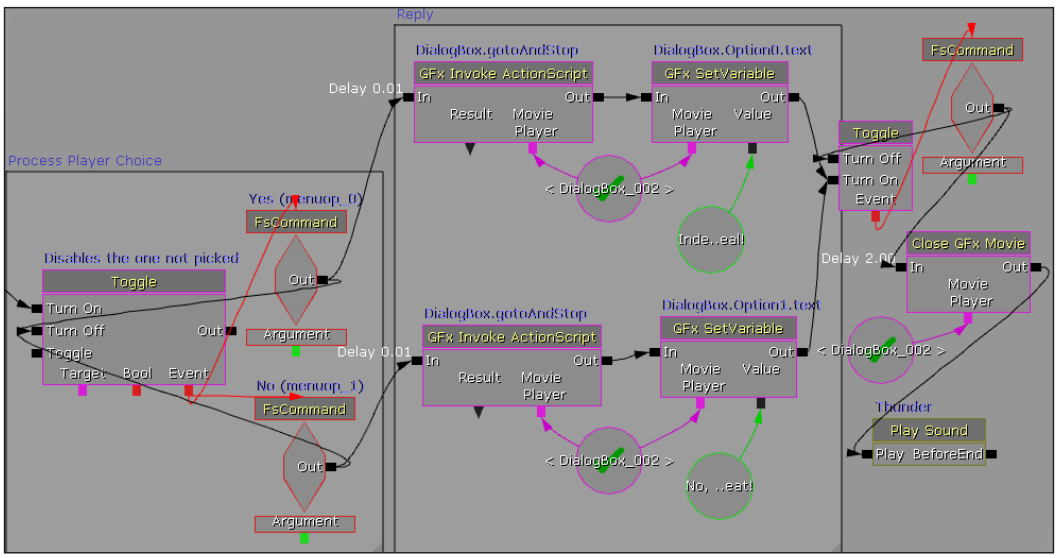
1. In your scene, set up a new Trigger in the middle of the scene, and in Kismet right-click and choose **New Event using Trigger_1 | Used**. Fire up an ***Open GFx UI*** action using your *DialogBoxMenu.SWF* in its **Movie** property, and associate the movie with the ***Named Variable*** *DialogBox_002*.



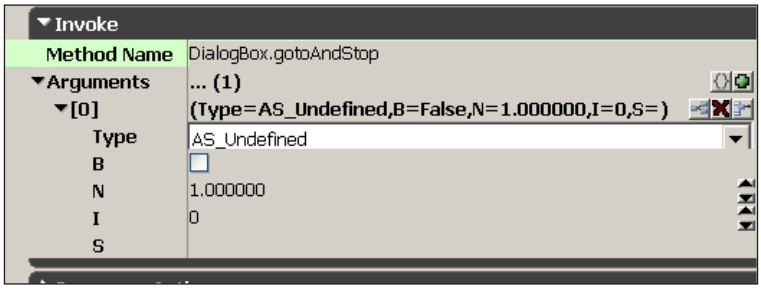
2. Right-click and choose **New Action | GFx UI | GFx Invoke ActionScript**. Connect this action to the previous one. This will allow you to run ActionScript functions from Kismet. Under the **Method Name** property in the properties box, enter *OpenMenu*. This command calls our SWF ActionScript to jump to frame 21 of the DialogBox symbol, the start frame of our menu state.
3. Right-click on the input for the ***GFx Invoke ActionScript*** action and give it an **activate delay** of 0.05. We wouldn't need to do this normally, but we need to give the SWF a split second to load before we can reliably invoke ActionScript in it.
4. Add two **New Action | GFx UI | GFx SetVariable** actions and use them to set `DialogBox.option0.text` to **Yes** and `DialogBox.option1.text` to **No**.
5. Add another ***Gfx SetVariable*** node to change **DialogBox.textField.text** to *Kismet is easier than ActionScript, right?*



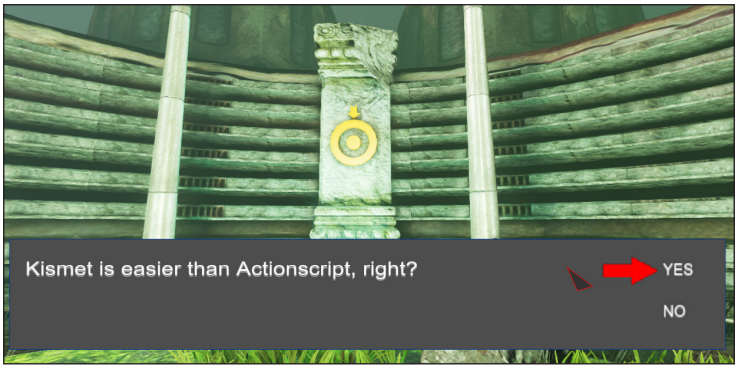
6. Create a ***Toggle*** node and connect the last ***Gfx SetVariable*** action into its **Turn On** nub. Add two ***FsCommand*** events for *menuop_0* and *menuop_1*, and disable them using the ***Toggle*** as shown in the next image within the section: *Process Player Choice*. In the **Obj Comment** field, name *menuop_0* as **Yes** and *menuop_1* as **No** to help differentiate them, and also don't forget to assign the SWF **DialogBoxMenu** to both the **FsCommand** events.



7. Add two new ***GFx Invoke Actionscript*** nodes and connect one to **Yes** and one to **No**. Have both of them execute the command `DialogBox.gotoAndStop` and set the argument to 1. This is done in the properties by pressing [ ] to add an entry, then putting 1 in the field **N** = 0.0, as shown here.



8. Add two new ***GFx SetVariable*** nodes to change the **textField** again. Connect one to the string for **Yes**, and one to the string for **No**. Under the one for **Yes**, add the **String Value**: *Yes, it's Unreal!* Under the one for **No**, add the **String Value**: *No, they are both great!*
9. Add a ***Toggle*** for the ***FsCommand*** *Next* and use it to close the dialog box, as shown in the previous Kismet sequence on the right-hand side.
10. Make sure each ***FsCommand*** event you use toggles itself off from its output. Now, run your game and test your dialog box. You now have a working conversation tree, and more importantly, a dialog box that can switch between displaying plain dialog and offering an interactive menu. You can use menu setups like this to turn on a variety of other events in your levels as well, or even as part of cinematic events. You can find the working example in *Packt_10_DilaogBoxMenu_DEMO.UDK*.



Scripting for CLIK components and the importance of Focus

CLIK components work differently from traditional AS2 components and movieclips when it comes to programming. While most basic script functions, like `gotoAndStop`, will work on them, other things such as registering button clicks will not work unless you use Scaleform's special functions. To understand this we're going to build another menu, namely, the main menu for the start of a game.

How it works

In our last example we noted that the dialog box was using some typical from-scratch ActionScript where it could have used CLIK. Specifically, we were using ordinary dynamic text boxes instead of CLIK label components, and CLIK components offer some shortcuts we didn't take advantage of.

If we were to re-build the menu we created with CLIK components, instead of adding text boxes we would add more **Label** components from our library. Their default text is set to blank, so we wouldn't need to edit that. Then, in addition to having our arrow take position alongside the player's chosen option, we can use the **Component Inspector** to set each of the inactive labels to the *disabled* state by default, graying out their text.

Finally, with a little extra code, we set `disabled` to false whenever an option is highlighted, like so:

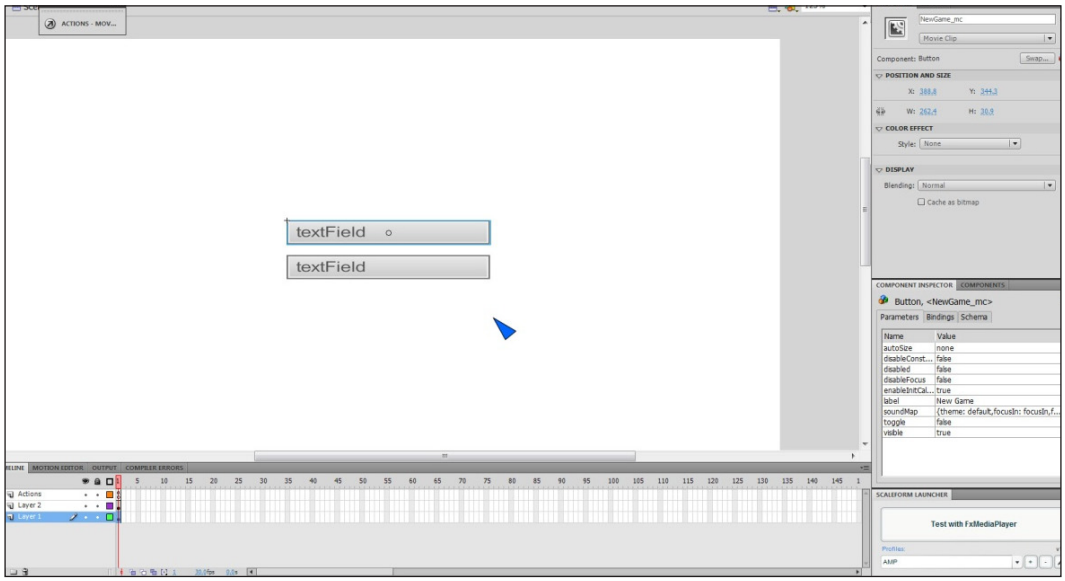
```
case 0:
  MenuPointer._y = option0._y;
  option0.disabled=false;
  break;
```

With this simple modification designed to take advantage of CLIK, the menu can now deliver even stronger feedback. As can be seen in the **Component Inspector**, a window that Flash uses for displaying and editing the default properties of pre-built components, CLIK components have a variety of built-in properties that can be edited through code like this, including the initial text and visibility. The Component Inspector is displayed on the lower right-hand side in the next image.

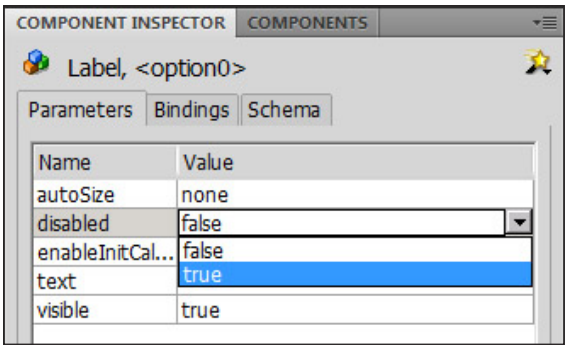
While this doesn't make much of a difference for the simple dialog menu we built in the last example, this knowledge makes all the difference in the world when dealing with things like buttons and check boxes, and we're going to use it to build a more advanced menu; specifically, a **main menu** for a game, also called a **Frontend GUI**.

How to do it...

1. Create a new Flash document and save it as *FrontendGUI.FLA*; copy the **button** CLIK component into the library.
2. Add two buttons to the stage. Give one the instance name *NewGame_mc*, and the other *Exit_mc*.



3. Press *Shift* + *F7* to bring up the **Component Inspector**. Dock it somewhere to keep it handy; under the properties and library viewer is a helpful place for it.



4. In the **Component Inspector**, change *NewGame_mc*'s **label** field to *New Game*, and *Exit_mc*'s label field to *Exit*.

5. Add an *Actions* layer to the timeline and at frame 1 open up the ActionScript Editor. Add the following code:

```
stop();
Mouse.hide();

NewGame_mc.addEventListener("click",this,"NewGame");
```

6. In the previous code snippet, `addEventListener` is a piece of syntax that CLIK borrows from ActionScript 3. Simply put, it's stating that we are adding an event listener to the *NewGame_mc* button; that it responds to the "click" action, and that when we click the button, we're going to use a function called *NewGame*. The parameter `this` is an ActionScript 2 formality telling the button to refer to itself when executing the click command.
7. It is vitally important that you use this syntax whenever trying to develop an event listener for CLIK buttons or any related components, as the traditional ActionScript 2 we used for our menu interface earlier will not work with them. If we try to use it, all the buttons' functions will fire no matter where in our Flash interface we click.
8. Add another `EventListener` to *Exit_mc*.

```
Exit_mc.addEventListener("click",this,"ExitGame");
```

9. Now, add two functions corresponding to the ones that we have *Exit_mc* and *NewGame_mc* pointing to:

```
function NewGame(){
fscommand("NewGame");
}
function ExitGame(){
fscommand("ExitGame");
}
```

10. Copy and paste all the elements from the **mouse cursor** at the beginning of this chapter into this file; otherwise we will have no mouse cursor when we try to use it in UDK. There's one last line of code we need to add. Add this after all the declarations for the button event listeners:

```
NewGame.focused = true;
```

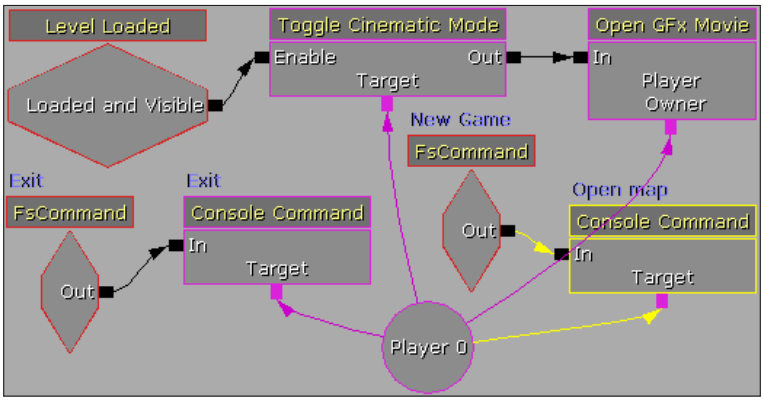
There's more...

A note on focus

Before we continue, it's important that at least one element in any Scaleform GUI be given focus so that the player can interact with it. If you preview your GUI after adding this line of code, you'll notice that the **New Game** button is highlighted, as if you were rolling over it with your mouse cursor. This is what "focus" means in Flash terms; it refers specifically to elements that the user has highlighted. When we set "focused" to true in this case, we're defining the default menu item that's highlighted when the player first starts up the menu.

Note that if you press *Enter*, it has the same effect on a highlighted menu item as if you were clicking it. Additionally, pressing the other arrow keys will shift focus from one button to another as if you were highlighting them with the mouse. CLIK automatically has this feature incorporated and requires no additional coding for keyboard controls. This is just one of the many advantages of using CLIK components.

1. Save your Flash file and export the SWF. Create a new map in UDK and import the freshly created SWF from the last lesson.
2. In the **Level** tab of the Content Browser, add one of your existing levels to the list. In this case we're simply using the same file as the *DialogBox*.
3. Add a ***Level Loaded*** event, a ***Toggle Cinematic Mode***, and an ***Open GFx Movie*** action and connect them in a chain. Assign the *FrontendGUI.SWF* file to the ***Open GFx Movie action***, in its **Movie** property.
4. Add two new ***FsCommand*** events for the *NewGame* and *ExitGame* commands we wrote in our ActionScript earlier.
5. Add a ***Console Command*** node and give it the *EXIT* console command. Connect its input to the ***FsCommand*** *ExitGame*.
6. Add a second ***Console Command***. Put in *Open [mapname]* as its command. Connect the ***FsCommand*** *NewGame* to this final ***Console Command***, as shown here:



7. Please note that this ***Console Command*** will not work when you're using play-in-editor; you have to either use **Play on PC** [ ] or else use the **UnrealFrontEnd []** program [ ] to publish a working build for the *Open [mapname]* command to work. However, if you preview your map with **Play on PC**, you should find that your menu will allow players to both start a new game and exit successfully.

See also

For general use of FrontEnd see *Chapter 1*, *Heads Up*. For more information on Kismet usage see *Chapter 4*, *Got Your Wires Crossed?*

Creating an animated day-to-night transition

Most of the recipes in *Chapter 8*, *Then There Was Light!* are brief. They revolve around how lights function, whereas this recipe's topic encompasses many aspects of development. With the passage of time, what changes might there be in a game? Sounds may change; swirling leaf particle effects may give way to hovering fireflies; the Skybox texture may ramp from a cloudy sunset to twinkling starlight; lights in windows could toggle on; NPCs might go to sleep and monsters would likely wake up. For a dry to rainy weather change the same kind of switches might occur. This bonus recipe is going to discuss the lighting concerns of a day-to-night transition in terms of animation of light and Material parameters in Matinee.

This recipe will cover the following topics:

- The use of ***Parameter*** nodes in Materials
- Material Instance Constants
- Material Instance Actor control through Matinee
- The creation of Skybox assets to reflect different times of day

Getting ready

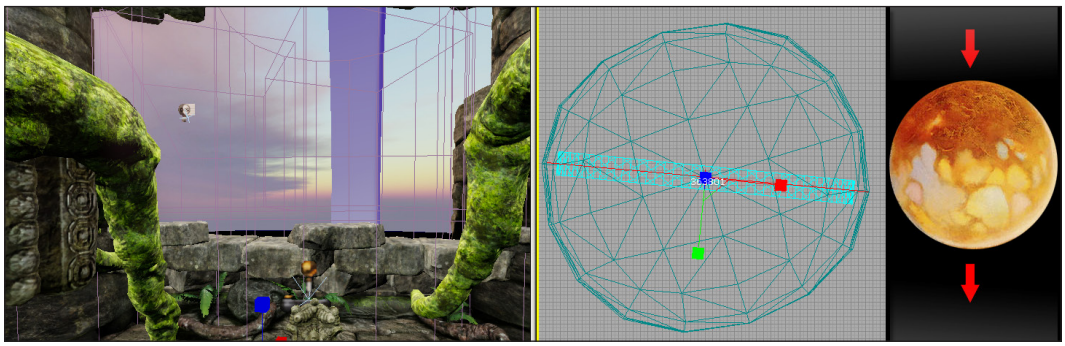
First let's consider the realism of a day-to-night cycle in a game.

Its presentation plays a part in the way it will influence user experience. There is hardly any realism in a sunset that lasts but a few seconds. So why would we want to show some kind of accelerated time progression? Maybe during a healing meditation? Maybe during a magic spell? Maybe during the player's death, or while the character takes a nap? Or maybe at the end of an AFK period to bring the player up to speed with what they've missed? The shift in each situation may require different under-the-hood mechanics or art direction. It is important to point out that this example is a functional one only, a suggestion of how to approach such a problem, and by no means a complete or artistically deep solution. The main thing is that the shift be pleasant and not creepy or jump too abruptly. We're not going to include any peripheral extras (though you could) like sound transitions or particle or weather changes.

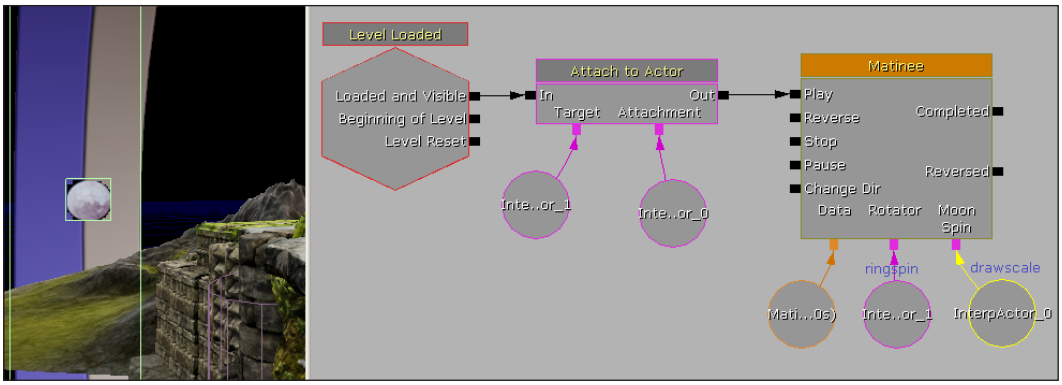
How to do it...

Controlling the heavens

1. Open up *Packt_08_DayNightCycle_Start.UDK*. We'll use a texture on a plane as the moon. That'll attach to a giant rolling tube. We'll use a sphere attached to the same tube as the sun. Why would we do that? Well, the sun mesh can emit light, plus its surface provides an area that always looks round to the camera. This matters, given the limited scale of our scene. The moon is less important for lighting and can be reduced to an image on a flat surface.
2. The Skydome in the scene is scaled up to around 36k units across at the base. The bigger the sky, the better the sense of parallax that you'll get when viewing foreground and background objects while the camera is moving. In the Content Browser, import the provided model *MoonTube.FBX* as a StaticMesh and set it in the scene so it falls just outside the Skydome. In its properties, turn off **cast shadows** and **receive lighting**. The model is just a circular strip of polygons obtained from the inner wall of a tube primitive. In the scene you will need to scale it up by a factor of about 500.
3. We're going to attach a quad plane carrying the moon texture to this strip on the right-hand side, as seen in the next screenshot (which is from a detail in Frank Frazetta's illustrated book cover for A Princess of Mars). Add to the scene the small mesh *Packt.Mesh.MoonQuads* and move it so it rests near the inward facing hoop with the sky surface in between. Make sure that you place it on the horizon, level with the base of the Skybox mesh, not up in the sky as we'll be scaling it later in Matinee based on this starting position.

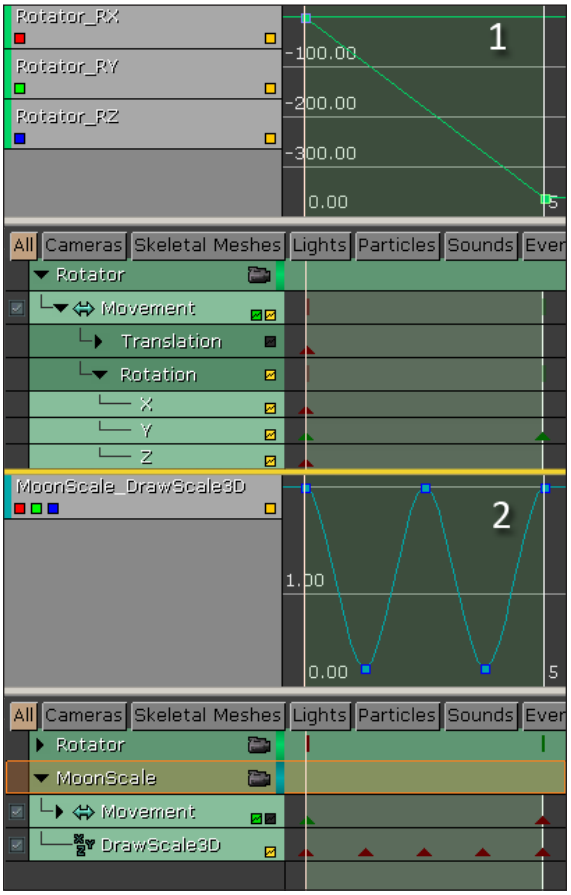


4. Press *K* (to show Kismet elements in the scene), then open Kismet with both the *MoonQuads* and *MoonTube* actors selected. Right-click on them and choose **Convert | Convert StaticMeshActor to Mover**. In Kismet, right-click and choose **New Object Vars Using ...** and then right-click again to add a **New Event | Level Loaded** first, and then a **New Action | Actor | Attach to Actor**. Add the *MoonTube* InterpActor's ***Object Variable*** to the **Target** nub of the ***Attach to Actor***, and the *MoonQuad* InterpActor's ***Object Variable*** to the **Attachment** nub, as shown in the following screenshot:



5. Now that the moon is attached to the tube, we need a ***Matinee*** action to rotate the tube, and besides that, we can scale the moon so it looks bigger at the horizon than the apex during its path across the sky. With the tube InterpActor selected in the scene, in Kismet hold *M* and click to create a ***Matinee***. Select the added ***Matinee*** and look in its properties. The **Looping** checkbox should be turned on. Double-click on the action, and in the Matinee Editor, right-click in the dark gray tracks panel and add a **New Empty Group** and name it *Rotator*. Highlight this, then right-click again and add a new **Movement** track.
6. Right-click on the added **Movement** track and choose **Split Translation and Rotation**. Highlight the **Rotate.Y** axis track (though the axis may vary according to some of your object placement choices) and key it at 5s on the timeline.
7. Press the show graph icon [ ], which turns yellow, so you can adjust the value for the rotation and right-click on the 5s keyframe in the Curve Editor (not the timeline) and choose **Set Value**. Enter `-360`.
8. Right-click on the equivalent key in the timeline and choose **Interp Mode | Linear**, so the tube will orbit at a constant rate. We've spun the tube and the attached moon follows.
9. That takes care of the moon's orbit. It is cool if the moon will scale up as it passes the horizon, and do so in a way that will be consistent even if we change the speed of the orbit. In the scene, select the *MoonQuad* InterpActor and in the Matinee Editor, right-click and add a **New Empty Group** named *MoonScale*. Highlight the added track, right-click and choose a **New Vector Property Track**. You will only be able to choose **DrawScale3D** as the method for this, but that's what we want.

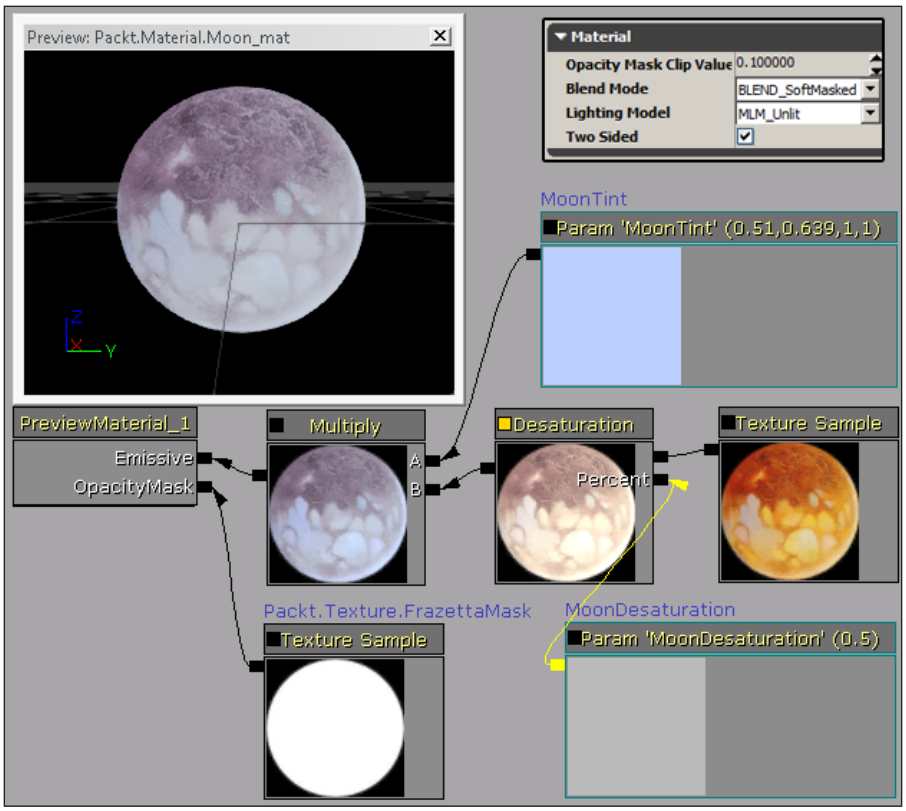
10. There are two graphs shown next that you'll need to set up by keying the tracks. The upper one is the linear rotation of the **moon tube**. The lower one is the scaling of the **moon plane** which you can see has two high and low periods (because it passes the horizon twice in its orbit, on either side of the scene). The values used are flexible, but a low of 1.0 and a high of 2.0 should be enough. The key times will be 0s, 1.25s, 2.5s, 3.75s, and 5s. It is easiest to place the keys loosely with the time slider, pressing *Enter*, then adjusting the created keys by right-clicking and choosing **Set Time**.



The Moon and the Sun

1. The texture for the moon has already been included in the provided content as *Packt.Texture.FrazettaMoon*. Add this as the texture in a **Texture Sample** feeding the Emissive channel of a new Material. Call the Material *YourFolder.Material.Moon_mat*.

2. The ***Texture Sample*** using *Packt.Texture.FrazettaMoon* needs to hook its RGB into a new ***Color | Desaturation*** node's input, as shown in the next screenshot. The ***Desaturation*** needs a percentage that we can animate.

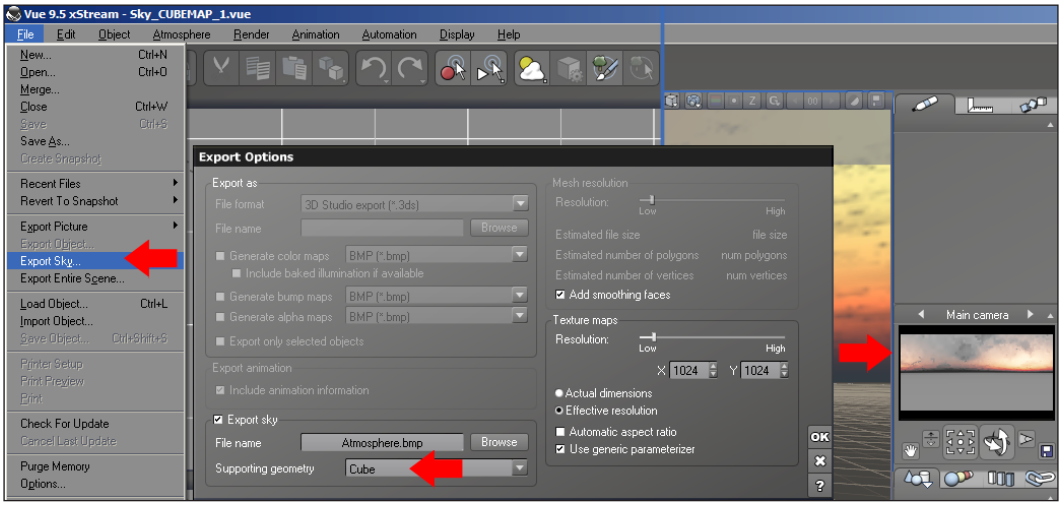


3. Hold *1* and click to add a ***Constant*** and right-click and choose **Convert to Parameter** too. In the node's properties, set the **Parameter Name** to *MoonDesat*.
4. Hook the **Out** of the ***Desaturation*** node into the **B** input of a ***Multiply*** node (hold *M* and click), and hook the *MoonDesat* parameter's output into the **Percent** input of the ***Desaturation*** node.
5. We just need a color tint and an opacity mask to complete the Material. In the Material Editor, press *3* to add a ***Constant 3 Vector***. You can set whatever RGB value that you prefer as this will be animated later in Matinee. Press *M* to add a ***Multiply*** node, and hook the **A** channel to the vector's Diffuse output. Then right click on the node and choose **Convert to Parameter**. Set the **Parameter Name** to *MoonTint*.

6. Also, we need to set the opacity mask for the texture's background. A texture *Packt.Textures.MoonMask* has already been prepared, so assign this from the Content Browser to a new **Texture Sample** in the Material Editor, and hook this up to the Opacity Mask channel of the **PreviewMaterial**. As yet, the Opacity Mask channel is not active, so we should deal with this next.
7. In the **PreviewMaterial** properties, set the **Blend_Type** to *BLEND_SoftMasked* and set the **Opacity Mask Clip Value** to 0.1 and check on **Two Sided**. The **Lighting Model** can be *MLM_Unlit* because we are using an Emissive channel. Compile the Material and save your package.
8. In the Content Browser, right-click on the finished Material and choose **Create New Material Instance (Constant)**. The default name that it sets will be fine. Open the new asset in its viewer, and check on the checkboxes beside the scalars *MoonTint* and *MoonDesat* to enable them. The parameters in the texture should be accessed from a **MaterialInstanceActor** [MI] placed in the scene from the **Actor Classes** list, using a **Matinee** in Kismet (one is a Material vector parameter and the other is a float Material parameter) to animate their values.
9. Assign the **MaterialInstanceConstant** to the *Moonquads* actor in the scene, and set its **Lighting** properties (*F4*) so the object does not cast shadows or receive lighting, in order to speed up builds.
10. Making the sun is easier. We could create it the same way we made the moon, but instead we'll use a sphere mesh. The emission of light off its surface may be helpful for the scene. In the Content Browser, locate and highlight the mesh *Packt.Mesh.Sphere*. Add this to scene as a Static Mesh then right-click on it and choose **Convert | Convert StaticMeshActor to Mover**. Place it opposite the moon's start position, inside of the big hoop, at horizon level.
11. In the Content Browser, locate the Material *Packt.Material.Hotspot* and with the **InterpActor** selected right-click in the Material Editor and choose **Material | Assign from Content Browser**. It should glow hot white.
12. In Kismet, copy and paste the existing **Attach to Actor** node and both its connected variables. To swap the moon to the sun, right-click on the **Attachment** nub of the pasted one, and choose **Assign Using InterpActor...** with the sun model selected in the scene. Make sure the **Attach to Actor** action remains hooked up to the **Level Loaded** event. It doesn't need an output. So we've attached the sun to the rotating tube around the sun just as we did with the moon.
13. To ensure the sun model actually lights its surroundings, select the sphere, press *F4*, and expand the properties section: **DynamicSMActor | StaticMeshComponent | Lightmass | LightmassSettings** then enable **UseEmissiveForStaticLighting**. Make sure its **Emissive Boost** property is 1.0, since we will be affecting the **Emissive** value in the Material through a parameter later on.

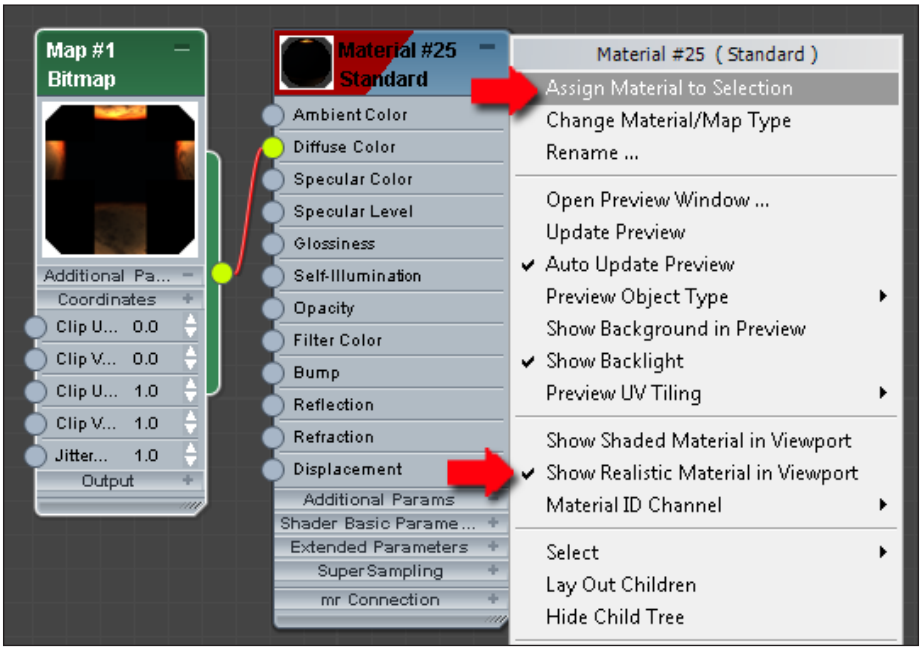
Enhanced assets

1. We have to provision a better sky which will look good with a changing time of day. The current one has a visible sun in the texture. To generate a sky texture, you may want to obtain the excellent landscape generation application **Vue 9.5 PLE** (`http://www.vue9.com/ple`), and use its **File | Export Sky** option which can give you preset cubemaps and UV Sphere projections. Exporting can be a long process if you are using large frame output sizes. As usual we provide some readymade skies, and there is one for each time of day we'll blend through. The arrows shown next mark the few steps needed to export the sky once you have set up its **Atmosphere**.



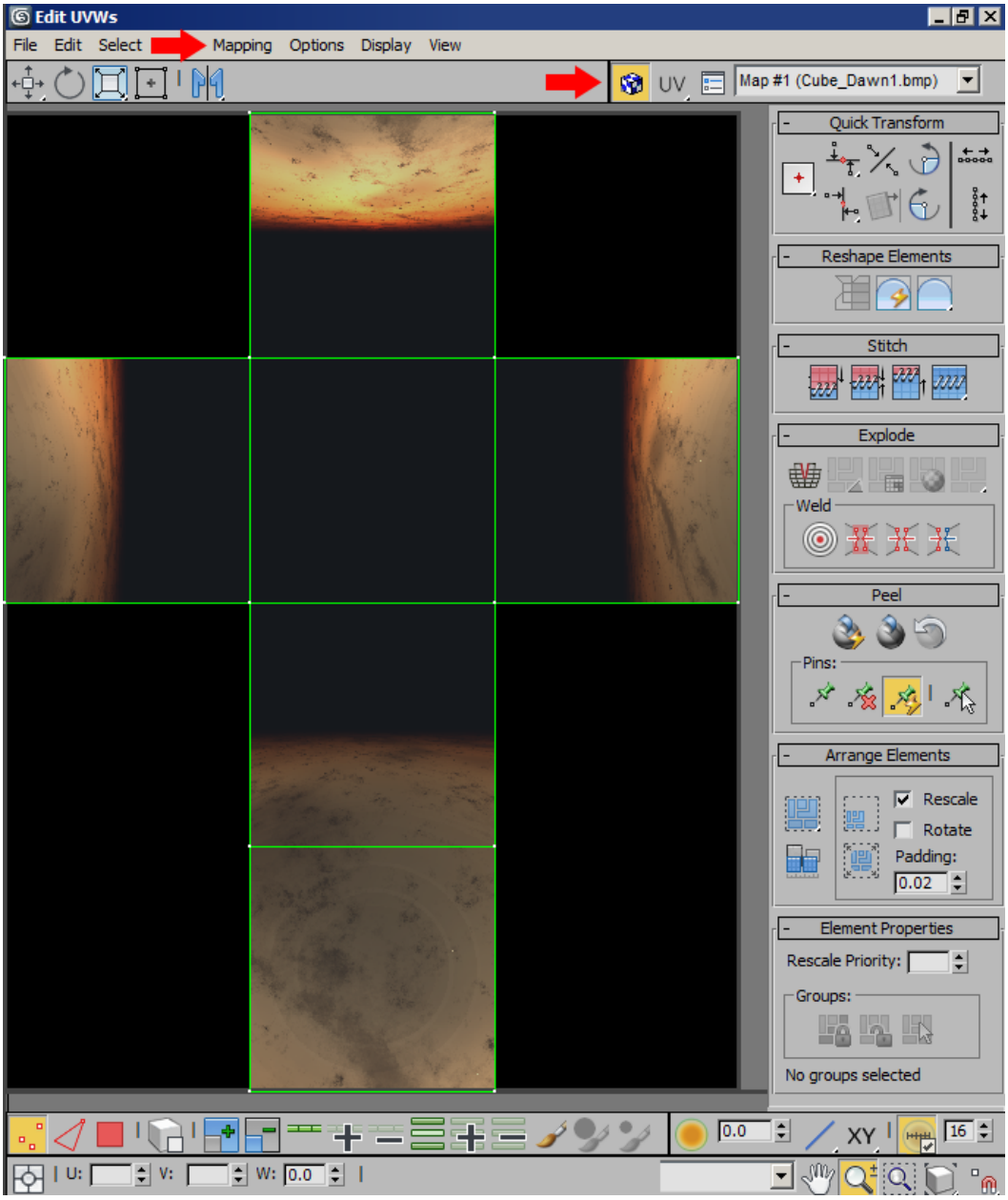
2. Once that sky is rendered to an image, what to do with it? First, remember to adjust the look of the Vue sky through the **Atmosphere** menu, and export out a few more (dawn, day, night at least). In Photoshop resize the renders to 1024x1536, and name them something like *CubeDawn.PNG*, *CubeDay.PNG*, and *CubeNight.PNG*. Then we need to make a sky mesh which has the right UV mapping.
3. So, over to 3ds Max we go. Luckily, this part is reasonably easy though some basic knowledge of 3ds Max 2012 is assumed. If you don't have this, you can use the provided assets.
4. Create a Box primitive with the default 1x1x1 segments, and use **Edit | Transform Toolbox | Center** to set the model at 0,0,0. You can set the **cube** option at creation time or just adjust its length, width and height manually in either the **Modify** panel or the **Transform Toolbox** after creation.

5. In the **Slate** Material Editor, right-click to create a **Standard Material** and apply the image generated in Vue to its **Diffuse** map channel. Right-click the Material and choose **Assign Material to selection** and again right-click to choose **Show Realistic Material in viewport**.

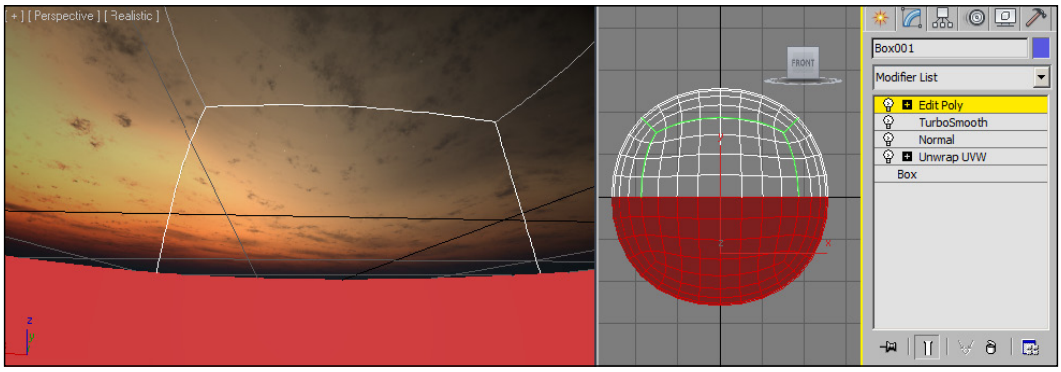


6. Now add an **UnWrap UVW** modifier to the cube object and click on the **Open UV Editor...** button.

7. In **Face** mode [ ], press *Ctrl* + *A* to select all faces and choose **Mapping | Unfold**. With a little luck this will generate a cross shape. If not, you'll have to right-click to **Break** and move the faces so they line up as seen in the following screenshot:


Edit UVWs
File
Edit
Select
Mapping
Options
Display
View
UV
Map #1 (Cube_Dawn1.bmp)
Quick Transform
Reshape Elements
Stitch
Explode
Weld
Peel
Pins:
Arrange Elements
Rescale
Rotate
Padding:
0.02
Element Properties
Rescale Priority:
Groups:
No groups selected
0.0
XY
16
U:
V:
W:
0.0

8. Match the sides of the box with the image. Preview the image in the UV editor to make this easier (the icon [ ] is highlighted at the top of the preceding screenshot). You may have to nudge the UV edges to avoid little seams. Since you are working with a box, add a **Normal** modifier to flip the faces inwards. Then add a **Turbosmooth** modifier with an increment of 2 or 3 to round it off. Lastly, add an **Edit Poly** modifier and select the base polygons and delete them. In the next screenshot, the **mapping seams** are shown on the rounded off box. It is advisable to delete the unnecessary polygons which are shaded.



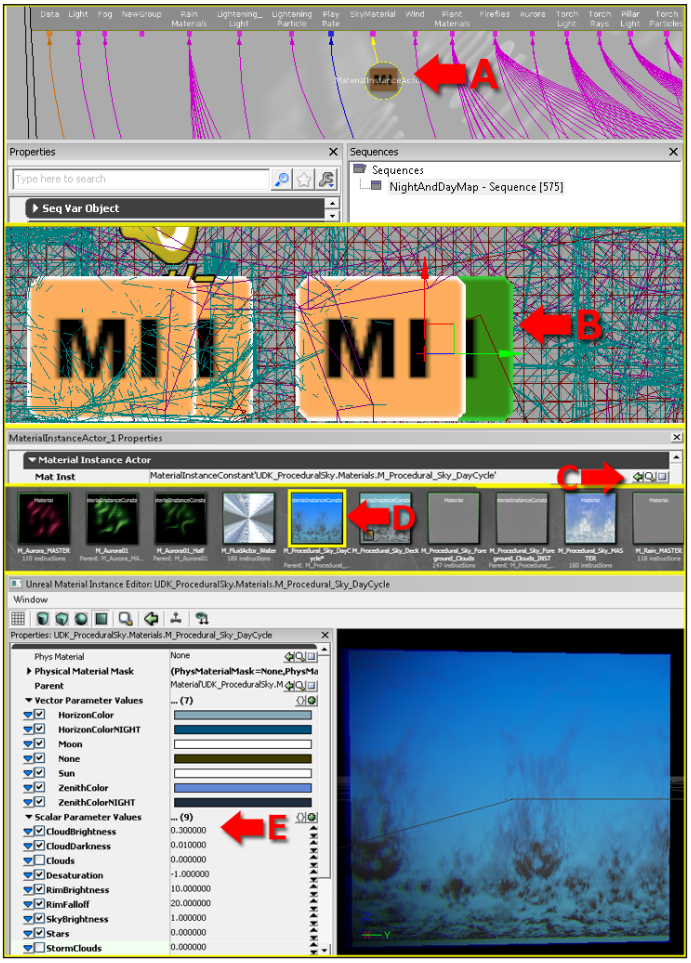
9. Export the Skydome as .FBX and import it in UDK as a StaticMesh in your package.
10. In UDK, add the Material derived from the imported Vue sky texture to the model through the Static Mesh Editor, in its **LODInfo | [0] | Elements** | **[0] | Material** channel. Alternatively, you can use the provided asset: *Packt.Mesh.Packt_SkyDomeDayNight*.
11. A Skydome lights itself through the **Unlit** setting in its Material, it speeds up calculation of the build if the actor does not need to receive any light.
12. Add the Skydome to the scene, replacing the existing one, which was just a stand in. Make sure the new sky dome is scaled so it fits snugly within the tube, so you have moon and sun clearly showing, then the sky, then the tube on the outside. That ensures the tube will never be visible. Here, you'll want to save. In the actor's properties (*F4*) go to the **Lighting channels** and disable cast and received shadows.

Lighting Control

1. We're now ready for the lighting aspect. In UDK's shipped content there is a reference map called *NightAndDayMap.UDK*. Open this and examine its Kismet to get ideas on transitioning textures and light properties.

2. The image sequence given next highlights that within the ***Matinee*** *Time of Day* there is a *SkyMaterial* track which uses the **Object Variable**, **A**, which uses a **MaterialInstanceActor** (**MI**) scene actor, **B**, which itself uses *UDK_ProceduralSky. Materials.M_Procedural_Sky_DayCycle*, **C**. If you open that from the Content Browser then you will see there are many parameters available from its **Parent** Material, including **D**— *HorizonColor*, *Moon*, *Sun*, *ZenithColor*, *CloudBrightness*, *SkyBrightness*, and even *Stars*.

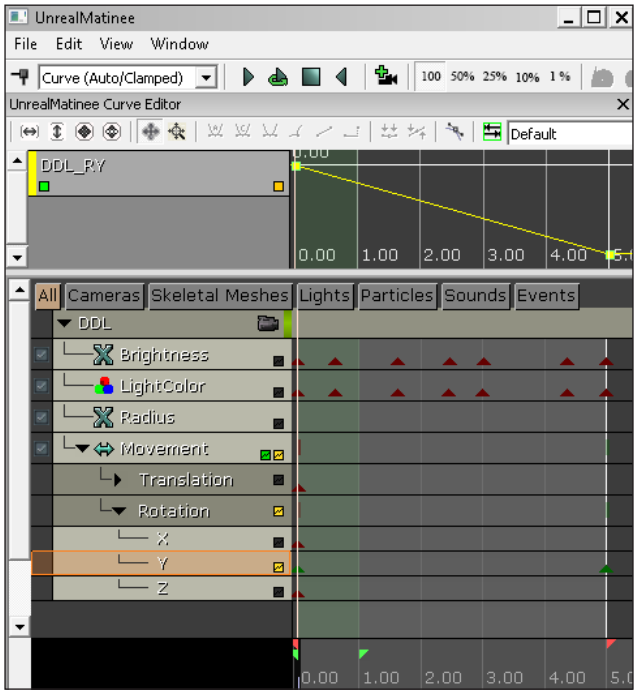
3. If you wiggle the MIC view preview, you'll observe there is a clever **Bump Offset** in the clouds, creating the illusion of cloud depth and layers. Bump Offset is discussed in *Chapter 9*, *The Devil is in the Details!* This is an elaborate example; the procedure that we'll follow is a simplification that can be built on. We won't have any moving cloud cover, but you could render sky textures without clouds then add extra rotating dome meshes which have **BLEND_Translucent** cloud textures on them, but it'd just involve similar steps built on top of the cycling we are about to set up.



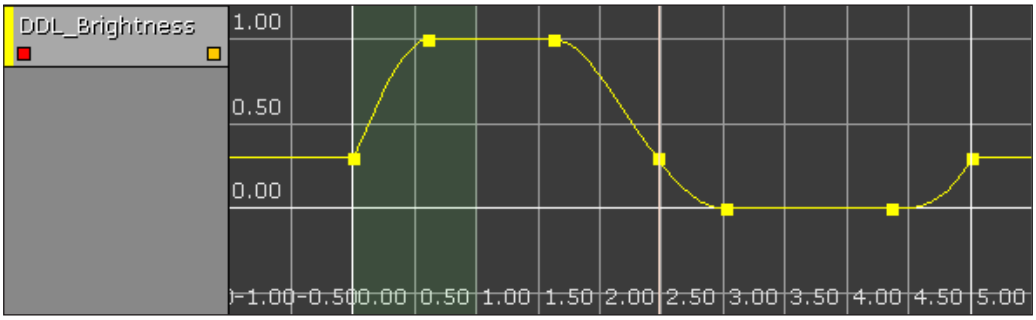
4. Our scene includes lights already. These will need to be controlled, along with changes in sky Materials and the sun's emissions. It could be done through one huge ***Matinee*** action, or several.

5. In the scene, choose **Edit | Find Actors** and type *Dom* in the search field, this should highlight the DominantDirectionalLight actor which is the major source of scene illumination. Select this, and move it so that it's in the middle of the large *MoonTube* actor, above the buildings. Make sure that its blue arrow faces away from the Sun model you placed earlier. Press *F4* and set the **Movement | Physics** property to *PHYS_Interpolating*. This avoids a warning the editor will otherwise throw up since keys on the timeline are interpolated, and any physics object that's animated there isn't directly under physics.

6. In Kismet, hold *M* and click to create a new ***Matinee*** action and label it *DDL*. In the ***Matinee*** *DDL*, add a **New Lighting Group** while the **DominantDirectionalLight** actor is selected in the scene, and add a **Movement** track to it. Create a 360 rotation over 5s in the same way the *MoonTube* rotates, so the light turning matches the sun mesh's orbit. For the **Movement** track, right-click and choose **Split Translation and Rotation**. Highlight the Y axis and set keys to provide the -360 degree rotation. Note the minus! Right-click on the keys and choose **Interp Mode | Linear**. Right-click on the **Movement** track and make sure **World Frame** is ticked, not **Relative to Initial**.

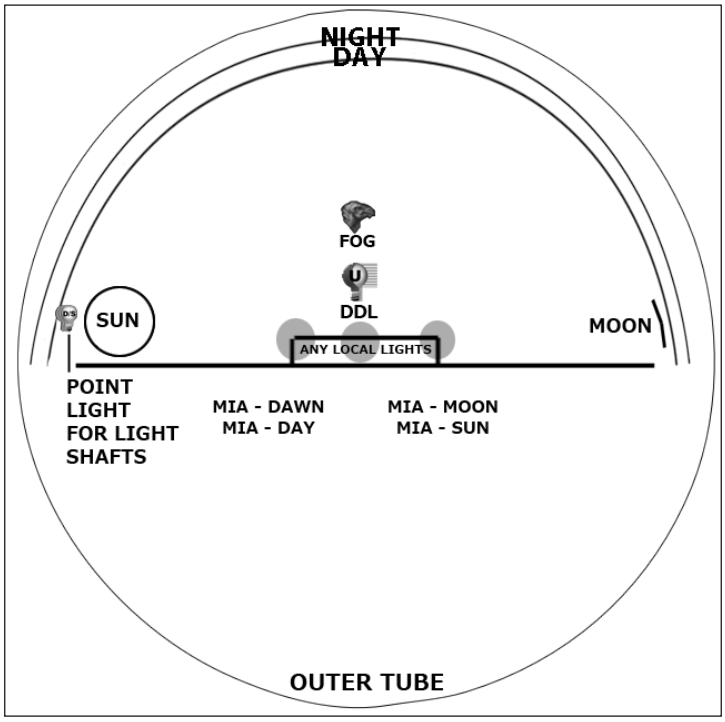


7. During night time we don't want the lights already in the scene to be bright. For the Dominant light we can adjust the track **Brightness** in the **Lighting Group,**. Given the sun is rising first and traveling under the world last, we want the brightness to start on 0.25 and build up to 1 quickly, then reduce back to 0 around 2.5s, then stay that way until the end, where it will lift again to 0.25., as shown here:

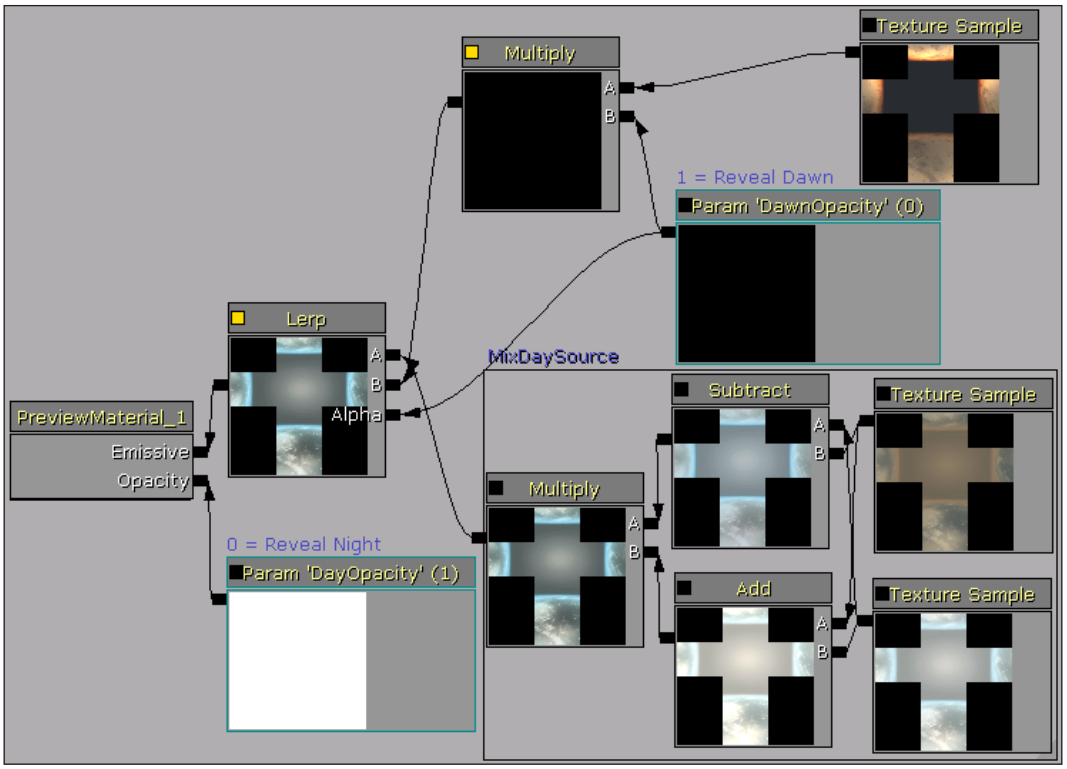


8. In the same vein, we'll have to edit the color of the DominantDirectionalLight so it is yellowish white at its zenith, orange red at the horizon, and blue below the horizon. The blue color probably won't contribute much due to the overall brightness being already reduced. To make these changes is easy. Select the **LightColor** track in the *DDL* group in the ***Matinee*** *DDL* and add keys. Remember that these are Red Green Blue values. RGB=XYZ. Place all the keys, then one by one right-click on them, choose **Change Color**, and set colors from the floating **Select a Color** dialog. Notice the curves for the **LightColor** track are tinted to reflect your choices.
9. If you were to build now and PIE you might notice the player weapon getting brighter and dimmer in accordance with the scene light, but the scene would be static (probably dark except the sky). Select the DominantDirectionalLight and press *F4*. Expand the **Light | Light Component | Lightmass | Indirect Lighting Scale** property and set it to a larger value, like 1024 or 2048. PIE and check if the light is changing. The sky will no doubt look the same no matter what, as it's self-lit. To avoid weirdness, do not enable **Render Light Shafts** for this light.
10. The sun is using an Emissive Material. If its Material's ***Constant 3*** is converted to a parameter and you use an **MIC** derived from the Material, the intensity of the Emissive channel can then be accessed through a **Matinee**.
11. The last thing to change then is the sky texture itself. If we set a **Translucent** type for the sky Material using a ***Constant*** so that it becomes opaque or transparent through a single value, we can use a ***Parameter*** to ramp the opacity in ***Matinee*** to fade it out and reveal a second Skydome added to the scene above the first.

12. We have a *Dawn*, *Day*, and *Night* set of textures for the sky, so we can blend between these. Add one more Skydome mesh to the scene (by selecting the first and choosing *Alt + Move* upwards or just by placing in a new instance from the Content Browser and scaling it so they form onion layers) and apply the Materials to them respectively. Put *Day* innermost, then *Night* on the outside, as shown next. *Night* doesn't need to animate and just needs an **Unlit** Material with the *Cube_Night1* texture feeding its Emissive channel. It will sit on the outside, as seen in the following diagram, waiting to be revealed:



13. Create a new *Day* Material and set its **Blend Type** to *BLEND_Translucent*. Add a ***Constant*** node, right-click on it and choose **Convert to Parameter**. Set the **Parameter Name** as *DayOpacity*. Hook this up to the Opacity channel. You can also use *Packt.Material.Cube_Day1_mat* which already has this done.



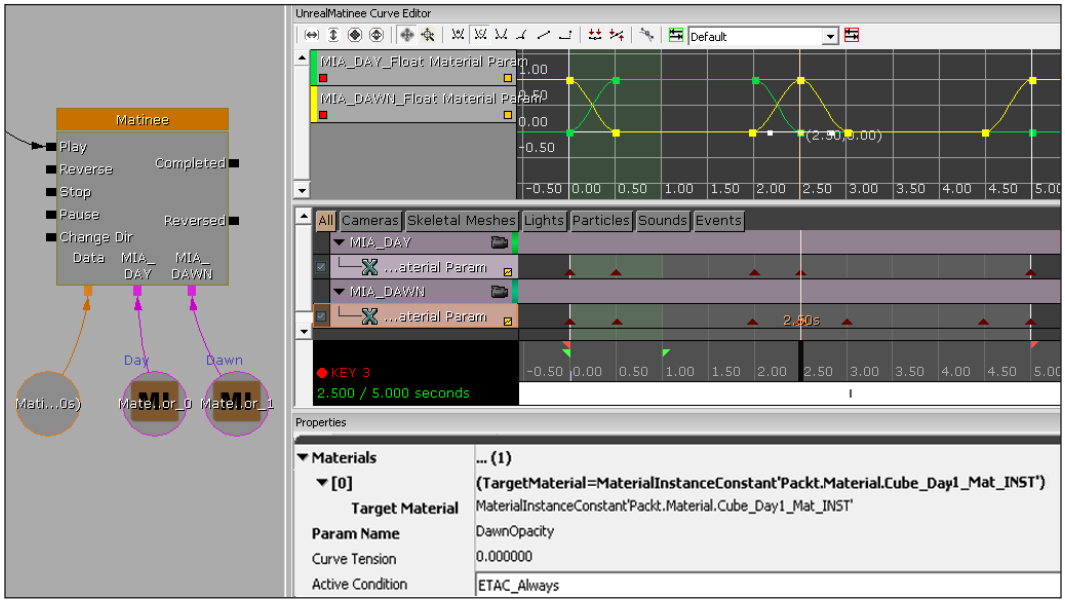
14. Right-click on the *Day* Material in the Content Browser and choose **Create New Material Instance (Constant)**. Go with the default name which just adds `_INST` on the end. The new asset will open in the MIC Editor, and you can expand the **Scalar Parameter Values** section to check the *DawnOpacity* and *DayOpacity* are there as intended. An essential, but easy to forget, step now is to tick the checkbox next to the *DawnOpacity* and *DayOpacity* entries. If you don't they won't be exposed to for adjustment in the Matinee editor.

15. Earlier we noted that UDK's *DayAndNightMap.UDK* affects Materials in ***Matinee*** using **MaterialInstanceConstant** assets which are staged using **MI actors**. We need to do the same. To locate these, open the Content Browser and switch to the **Actor Classes** tab. Expand the **Uncategorized** category and highlight **MaterialInstanceActor**. In the scene, anywhere you like, place two of these.
They have an icon like this: [MI].

16. Select one and press *F4*. In its property **Material Instance Actor | Mat Inst** add the asset *Packt.Material.Cube_Day1_Mat_INST* (or your own) by highlighting it in the Content Browser then clicking the assign icon [ ] in the property channel, and likewise for the second **MI actor**, add to the same property the same asset: *Packt.Material.Cube_Day1_Mat_INST* (or your own).

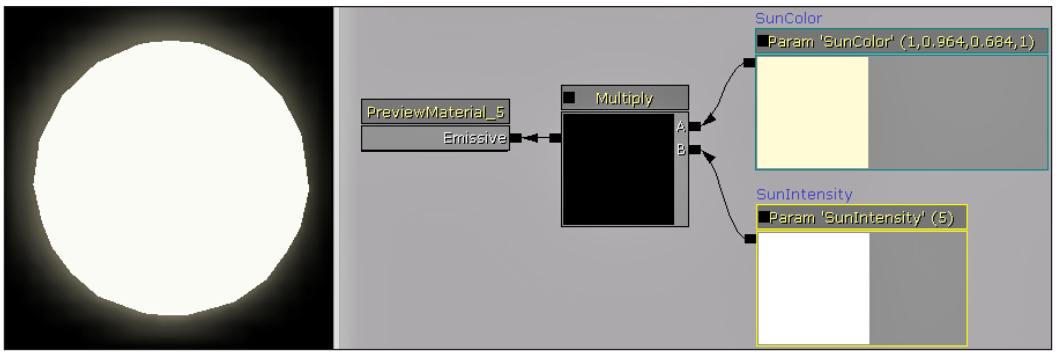
17. In Kismet add another ***Matinee*** action and comment it as *MIC Controls*. With the **MI actor** *Dawn* selected in the scene, right-click in the Matinee Editor and choose **New Empty Group** and name it *MIA_DAWN*. Right-click on the created track and choose **New Float Material Parameter Track**. Because the selected **MI** already has the MaterialInstanceConstant assigned to it, this should also show up in the properties of the track in the Matinee editor. If not, just add it. Then set the **Param Name** property to *DawnOpacity* editor. With the added track highlighted, press *Enter* to add a keyframe, and set more keys at 0.5s, 2.0s, 2.5s, 3.0s, 4.5s, and 5.0s. The value of 1.0 should be set for the keys 0.0s, 2.5s and 5.0s. For the rest, a value of 0.0 should be set.

18. Now select in the scene the **MI actor** *Day*. In the Matinee Editor, right-click below the *MIA_Dawn* group and choose **Add New Empty Group**, and name it `MIA_DAY`. Highlight this and right-click to choose **New Float Material Parameter Track**. As before, make sure the properties include an entry for the MaterialInstanceConstant **Materials | [0] | Target Material**: *Packt.Material.Cube_Day1_Mat_INST* (or yours). The **Param Name** should be *DayOpacity*. Keyframe the track so that there are keys at 0.0s, 0.5s, 2.0s, 2.5s. Set a value of 1.0 for the keys at 0.5s and 2.0s. The default, which is 0.0 is good for the other two keys. You should get a graph similar to what's shown in the following screenshot:



19. To ensure that everything works well, go to **View | World Properties**, and expand the **Lightmass** properties. Turn on **Force No Precomputed Lighting**. Save, build and PIE to check how it looks. An alternative way to do all this is indicated in the image below, and it uses just one Skydome mesh and one Material. The result via Kismet in the end would be the same, since it involves animating two parameters through Matinee.

20. Finishing up, we need to make the sun sphere redden and enlarge as it passes the horizon. Given what we've already done, there's nothing really new here in terms of process. Select the sphere, right-click and choose **Material | Find in Content Browser**. Create a copy of the Material and call it *Yourfolder.Material.SunSphere*. In this Material, adjust the emissive nodes from a ***Constant 3 Vector*** to a ***Parameter*** named *SunColor* and a ***Constant*** to a ***Parameter*** named *SunIntensity*, as follows:



21. From the Material create a **Material Instance Constant** (as before), and in the scene add another **MaterialInstanceActor** [MI], and apply the asset to it. Open the new ***Matinee***, and in the Matinee Editor create two tracks which will let you key the color and intensity. For color, choose a **New Vector Material Param** track, and for the intensity choose a **New Float Material Parameter** track. Don't forget to add the **Parameter names** in the track properties. This really repeats the process we used before.

22. Once you have keyed the tracks, you can build and PIE to check the sun for redness and scaling as it sinks and rises. The sun size would work from the sphere itself, not the **MI actor**, and a **Float Property Track | DrawScale** would drive it.

23. A final touch would be to add a PointLight with a huge scale to cover the scene just behind the sun sphere, then use Kismet to attach the PointLight to the sphere, and then tick the **Render Light Shafts** checkbox in its properties. You would need to right-click on the light and choose **Convert Light | PointLights | Point Light Moveable** for this light to work. It'd create nice glimmering sun beams.

24. Having completed the work, you can sit back and watch the show. Be sure to set a slower **Play Rate** for all the relevant ***Matinee*** actions, as a five second day is a bit too fast. It is clear enough that if it's a 24 hour day and we have a 5s orbit, then we'll want to reduce the **Play Rate** to around 5/Play Rate = 86400, where `86400` is a day in seconds. The real play rate should be 0.00005787 but UDK only allows 6 decimal points, so we'll use 0.000058. So a day in UDK will be about 3 minutes faster than a real one, which again doesn't matter on planet Frazetta. Only adjust this when everything else works fine or it will be hard to test your changes, having to wait a day to see the effect.
25. An example is included: *Packt_08_DayNightCycle_DEMO.UDK*. It's very dark during the night, so you may want to add some ambient blue light to the whole scene using fog, and obviously some local artificial lighting such as what's present in the scene already, in the cave mouth. If you want to, navigate in the editor to view the scene from up on the roof, select one of the rocks and choose **Play from here** to get a wider view of the sky.