Starling Tutorial

1. What is Starling? It is an Action Script 3 framework which is open source. It was developed to render games to the GPU on any device it runs on. It does this courtesy of the Stage 3D API built into Flash. This makes it very fast and efficient for mobile games, but also for Social networking games. It works well in conjunction with other frameworks like Box2D. In fact Rovio rewrote Angry Birds in Starling to port it to Facebook precisely because they could leverage the GPU prowess of Starling and the physics engine of Box2D.
2. Getting set-up:
   - Download the Starling frame work from http://gamua.com/starling/
   - After downloading, on the Mac, unzip and find
   - ***Downloads > PrimaryFeather-Starling-Framework-??????? > starling > src***
   - Copy this folder. Make a new folder on desktop, call it **code**, and paste the src folder inside it. Inside ***src*** are two folders, ***starling*** and ***com***. Drag them out of ***src*** so they are in the ***code*** folder. Throw ***src*** into the Trash.
   - Back on Starling website, click Help button, on Help page, click on the extensions link under the Wiki header. Select the Particle System extension.
   - Click the Download link to obtain the particle system. Unzip and find
   - ***Downloads > PrimaryFeather-Starling-Extension-Particle-System-v1.2-19-gbdfe72d > src > starling > extensions***
   - Copy this folder and paste it in this folder
   - **code > starling**
   - Next, connect to the Media Server and look for
   - ***Groups > Reid Perkins-Buzo Students > Public > MDIA 4905 > starling stuff > pooling.zip***
   - Copy this file, paste it here: ***code > starling > com***, and unzip it. You should have a folder path ***pooling > starling > StarlingPool.as***
   - This last is an object pooling class which will help manage a collection or “pool” of instances, so that we do not waste time creating and destroying them. This works best for apps that use a large, but known finite number of instances.
   - Delete the zip archives to help keep everything organized
3. We will be using Flash Professional for our Starling project. I had hoped to use Flash Builder, but it was not possible. Flash Builder has some advantages for doing Starling projects over Flash Pro: 1) Doesn’t make us keep stepping around a Timeline based interface when we just want to code; 2) Gives us another perspective on Action Script that we don’t normally have in Flash Pro.

- You can get a 30-day free trial Flash Professional; from the Adobe website: www.adobe.com/downloads > drop down menu > Flash Professional. If you subscribe to Adobe Creative Cloud (free accounts are available) you get it as part of the subscription.

4. We will also use Texture Packer, and will refer to a two other game creation helper apps:
   - Glyph Designer & Particle Designer: http://www.71squared.com/
5. Open Flash Pro and choose New > Action Script 3 Project. Save this in the ***code*** folder, and name the .fla file ***MyGame***. In the Properties panel, change the Target to AIR 3.4 for iOS. This makes it an iOS mobile project.

   Now click on the wrench icon next to the Target drop down to enter the correct iOS settings for a GPU-direct app. Set ***Aspect ratio*** to ***Portrait***, Select ***Full screen***, make the Render mode ***Direct***, and the Resolution ***High***. Click OK. Recall that a Starling application renders to the GPU, so if we don't have the render mode set to GPU or Direct, Starling won't run properly.
6. Click the pencil icon next to ***Class:*** in the Properties panel to open up an Action Script code window. By default, will define a class called ***MyGame***. To fix the parameters of the project we will enter a meta-tag which will convert to project properties. Just below the ***import*** statement, type

   [SWF(width=640, height=960, frameRate=60, backgroundColor=0x000000)]

   Since we are working in Flash Pro and not Flash Builder, we still need to set the stage size to 640 x 960 in the project Properties panel as well. Save this class file, but in the save dialog box, click the New Folder button, make a new folder inside your ***code*** folder, and call it ***core***. Save ***MyGame.as*** into this ***core*** folder. In the Properties panel for the Document, the ***Class:*** property should read ***core.MyGame.*** If it doesn't correct it so it does. Now go back into the ***MyGame.as*** editor and, on the first line of the class, add the word ***core*** after the word ***package***. Then save the file again.

   Why did we create the ***core*** folder? In order to organize our source files conceptually, we want to create separate folders for the conceptually different Action Script files we will be creating. These folders are also known as ***packages*** in other IDEs (like Eclipse and Flash Builder)
7. In the ***MyGame*** class, we want to create an instance of the Starling class which we will use to access all the advanced Starling objects. First, make sure to import the Starling framework by putting this just below the first import statement:

   import starling.core.Starling;

   This tells the Action Script compiler where to find the main Starling class, namely in the project folder > ***starling > core > Starling.as*** . Next, in the ***MyGame*** constructor

type:

```
var star:Starling = new Starling(Game, stage);
star.simulateMultitouch = true;
star.showStats = true;
star.start();
```

The first parameter, ***Game***, is the name of the main Starling class we will be creating for our game, and the second is the name of the main stage object we will be using.

Hint: using control-spacebar opens up the Flash tips windows for help with coding.

8. We need to create our basic Starling class, ***Game***. Choose File > New > Action Script 3 Class. Type ***Game*** into the Class name box, then click OK.

   The Game.as class will open in the editor. In the constructor, type the following to add an event listener to listen if the Game class has been completely added to the stage:

   ```
   addEventListener(Event.ADDED_TO_STAGE, init);
   ```

   Add ***import starling.events.Event;*** to the imports section of the class. And now, just below the constructor function, add the init function (which the listener calls when Game has been added to the stage entirely) and the update function:

   ```
   private function init(evnt:EVENT):void
       {
           addEventListener(Event.ENTER_FRAME, update);
       }
   private function update(evnt:EVENT):void
       {
           //update the Game as a whole
       }
   ```

   Save the ***Game.as*** code into the ***core*** package folder. Make sure that on the first line of the class, you add the word ***core*** after the word ***package,*** if it's not there. Then save the file again. Remember to do a File > Save All often as you work!

9. One of the ways to design games is to use the concept of a Finite State Machine. A FSM can be thought of as a device which has only a finite number of conditions or states it can be in. A toaster for example is either toasting something or not toasting something. A sewing machine is either sewing or not sewing (complex sewing machines have a great many states, however).

   For our game, will use a three state FSM. The game will either be in Menu state, Play state, or GameOver state. Each state will be implemented as an Action Script class. However, we want each state to share the same basic features. The easiest way to do this is to create an Action Script interface template, and make all the

states derive from it. That way they will share the same basic “interface” features.

Go to File > New, then choose Action Script Interface from the dialog box. Type ***IState*** (caps here are important) in the Interface name box, then click OK. The ***IState*** interface will open in the editor. Type the following just below the ***// interface methods*** comment:

```
function update():void;
function destroy():void;
```

These functions make sure that we end up with both an update and a destroy method for each state.

Save this interface file, but in the save dialog box, click the New Folder button, make a new folder inside your ***code*** folder, and call it ***interfaces***. Save ***IState.as*** into this ***interfaces*** folder. Again, make sure that on the first line of the class, you add the word ***interfaces*** after the word ***package,*** if it’s not there. Then save the file again.

10. Now we can set up our state classes. First let’s set up the Menu state. Select File > New, and choose Action Script Class. Type ***Menu*** (caps here are important) in the Name box. The ***Menu*** class will open in the editor. Make changes in the editor so that your code looks like this:

```
package states {
   import core.Game;
   import interfaces.IState;
   import starling.display. Sprite;
   import starling.events.Event;

   public class Menu extends Sprite implements IState
   {
      private var game:Game;

      public function Menu(game:Game)
      {
         this.game = game;
         addEventListener(Event.ADDED_TO_STAGE, init);
      }

      private function init (evnt:Event):void
      {
      }
      public function update():void
      {
      }
      public function destroy():void
      {
      }
   }
}
```

Save the ***Menu.as*** code, but in the save dialog box, click the New Folder button,

make a new folder inside your ***code*** folder, and call it ***states***. Save ***Menu.as*** into this ***states*** folder. Again, make sure that on the first line of the class, you add the word ***states*** after the word ***package,*** if it's not there. Then save the file again.

This is the basic code structure for the other two classes, so we can use it to create them as well. So select all the code in the ***Menu*** class and copy it. Select File > New, and choose Action Script Class 3. Type ***Play*** (caps here are important) in the Name box. The ***Play*** state opens in the editor. Now we simply select all the code, delete it, then paste the code we copied from the ***Menu*** class. We MUST change the Play state code in two places: in the actual class name, and also in the name of the constructor. Replace the name ***Menu*** in those places with the name ***Play***. We've just created the basic ***Play*** state. Now use the same process to make the ***GameOver*** state yourself. Remember to save them in the ***states*** package folder.

11. The Game class will implement the FSM and switch between the three game states as it needs to. So open the ***Game.as*** class and edit the code so that it looks like so:

```
package core {
    import interfaces.IState;
    import starling.display.Sprite;
    import starling.events.Event;
    import states.*;

    public class Game extends Sprite
    {
        public static const MENU_STATE:int = 0;
        public static const PLAY_STATE:int = 1;
        public static const GAME_OVER_STATE:int = 2;
        private var current_state:IState;
        public function Game()
        {
            Assets.init();
            addEventListener(Event.ADDED_TO_STAGE, init);
        }
        private function init(event:Event):void{
            changeState(MENU_STATE);
            addEventListener(Event.ENTER_FRAME, update);
        }
        public function changeState(state:int):void
        {
            if (current_state != null)
            {
                current_state.destroy();
                current_state = null;
            }
            switch(state)
            {
                case MENU_STATE:
                    current_state = new Menu(this);
                    break;

                case PLAY_STATE:
```

```
                current_state = new Play(this);
                break;

            case GAME_OVER_STATE:
                current_state = new GameOver(this);
                break;
        }
        addChild(Sprite(current_state));
    }
    private function update(event:Event):void
    {
        current_state.update();
    }
  }
}
```

I will explain this code in class.

12. Now we will turn our attention to the assets for our game. Switch to the Finder. Locate the file ***stuff.zip*** which came in the zip archive with these instructions. Unzip it, drag the ***stuff*** folder into the ***core*** package inside the ***code*** folder. Rename the ***stuff*** folder ***assets***.

    Now switch back to Flash Pro. We will create a class to manage these assets for us. Select File > New, then Action Script 3 Class. Type ***Assets*** (caps here are important) in the Name box, then click OK. This class should not be instantiated, it should be for reference only, such a class is called a ***static*** class. So we replace the constructor function with a static function that will always use the same instance of the class when it is called.

    The Assets class should be open in the editor, since we just created it. Change its code so that it looks like this:

```
package core
{
  public class Assets
  {
    public static function init():void
    {

    }
  }
}
```

    Save the ***Assets.as*** code into the ***core*** package folder. Make sure that on the first line of the class, you add the word ***core*** after the word ***package,*** if it's not there. Then save the file again.

13. Now to turn our assets into data objects that will be part of our final SWF file. Usually in Flash Pro, we just import them into the library, turn them into MovieClips, and then reference them in the code. Since we're doing this project in a "code-

alone" style (like Flash Builder does it) we will use SWF meta-tags, along with some class-creation, to do the same thing. Let's modify the Assets class so that it acts to embed the contents of the ***assets*** folder into our MyGame project.

We use the meta-tag **[Embed(source="...file path…")]** to embed an asset at a particular file location. Change the Assets class code so it includes the following:

```
...
import starling.textures.Texture;

public class Assets
{
    [Embed(source="assets/sky.png")]
    private static var sky:Class;
    public static var skyTexture:Texture;

    [Embed(source="assets/skyClouds.png")]
    private static var skyClouds:Class;
    public static var skyCloudsTexture:Texture;

    public static function init():void
    {
        skyTexture = Texture.fromBitmap(new sky());
        skyCloudsTexture = Texture.fromBitmap(new skyClouds());
    }
    ...
```

The Embed meta-tag, locates a texture in our Assets folder called ***sky.png*** and tells the Action Script Packager to include that in the final SWF file. We then use some standard Action Script to create a private static class to contain the image, calling it ***sky***. Then we create a public static Texture class called ***skyTexture*** that will allow us to manipulate this png image in our code. By using a ***public static*** class, we can access this Texture from anywhere in our code. This saves GPU memory since we don't need to instantiate multiple copies of it.

Finally, in the ***init*** function, we make ***skyTexture*** from the image contained in the ***sky*** class. Now this png file is embedded into our project and given the code handles we need to utilize it in our classes.

14. Now this method works well for one large png, used for a particular purpose, like a background, but for handling many small images, we will want to use Sprite sheets to pack them all together. This works better for two reasons: 1) minimizes latency since we only load one or two sprite sheets rather than dozens of small images; and 2) utilizes the GPU more efficiently since we load a smaller number of image files into the GPU's memory, then simply reference a Sprite sheet image as we need it.

    So unlike in our Unity example, we will be using a ***Texture Atlas*** approach which will guarantee accurate placement of the sprites with no kludge adjustments from image-frame to image-frame. But we need to produce our Sprite sheets in a way that works with such an approach.

If we had created all our Sprites in Flash Pro, we could use it to export the Sprite sheets in a nice format for Starling (recall it had a preset in its Sprite Sheet Generator for Starling). However, all our assets are already in our ***assets*** folder in the formats we need. I used the free version of Texture Packer (http://www.codeandweb.com/texturepacker) to create the ***atlas.png*** and the ***atlas.xml*** files for our game. You can read more about how to use this great game utility on its website.

The ***atlas.xml*** is an important file since it describes the location of the individual sprites inside the one large Sprite sheet. Open it up in a text editor (like Text Wrangler) to see the way that it encodes this data in XML. Starling has built-in parsers to read this sort of XML file and use it to make pixel-perfect mappings of the sprites onto the game stage.

15. We now need to set up the code in our MyGame project to utilize the Texture Atlas Sprite Sheet, ***atlas.png***, and its accompanying data file, ***atlas.xml***.

    Go back to the ***Assets*** class in our project. Add the following code just below the ***public static var skyCloudsTexture:Texture;*** statement:

    ```
    [Embed(source = "assets/atlas.png")]
    private static var atlas:Class;

    public static var txtreAtlas:TextureAtlas;

    [Embed(source = "assets/atlas.xml", mimeType="application/octet-stream")]
    private static var atlasXML:Class;
    …
    ```

    The first 3 lines embed the atlas.png Sprite sheet file, create a class to contain it, and give us a TextureAtlas class variable to instantiate it in. The second two lines embed the XML data file for the Sprite sheet give us a class variable to instantiate it in. Note that it is necessary to provide a mime-type for the XML file we embed. That's so it is NOT treated as a simple text file.

    Then in the ***public static function init():void*** function add the following code below the ***skyCloudsTexture*** line:

    ```
    txtreAtlas = new TextureAtlas(Texture.fromBitmap(new(atlas)), XML(new atlasXML()));
    ```

    This instantiates the class ***txtreAtlas*** as a TextureAtlas class which uses the ***atlas*** class for its texture, and the ***atlasXML*** class for its key to finding the individual sprites in that texture. Now we can use ***txtreAtlas*** to pinpoint each sprite we need when we need it using the built-in Starling methods for the TextureAtlas type of class.

16. We need to have access to at least one font for showing text to the player. We could use system fonts, but to give a game some style, we want to use a font that suits the game. Starling can use either system fonts or bitmap fonts that are embedded into the project (just like the bitmap images).

    We will use Gylph Designer to create a bitmap font. It does have a free version, but the free version won't export the bitmap. Because of this, I have made a bitmap font (***blippo.png*** and ***blippo.fnt***) in the ***assets*** folder.

    In Glyph Designer, I have a list of fonts on the left and I can choose any font on my system. On the right I have some panels which allow me to fine-tune my bitmap font, so it includes only the characters I will be using, and has the proper look. In the Included Glyphs section I can decide which characters I want in my bitmap. In the Glyph Fill I can determine the fill color or gradient. In the Texture Atlas panel I set the parameters for my bitmap font Sprite sheet. If we have the paid version, we would then click the Export button to produce a bitmap font and its data file (make sure that the file type is set to XML when you export).

    If we look at the ***blippo.png***, we see it contains all the glyphs that I selected in Glyph Designer when I created it. Opening the ***blippo.fnt*** file in a text editor, like TextWrangler, and we can see that it is an XML file giving the pixel-accurate position of each glyph of the font in the accompanying PNG file.

    We now need to embed the bitmap files in the ***Assets*** class. So back in Flash, open the ***Assets*** class, and add the following code just above the static ***init*** function:

```
[Embed(source = "assets/blippo.png")]
private static var blippo:Class;

[Embed(source = "assets/blippo.fnt", mimeType="application/octet-stream")]
private static var blippoXML:Class;
```

    Inside the static ***init*** function, we want to add the following code:

```
TextField.registerBitmapFont(new BitmapFont(Texture.fromBitmap(new blippo()), XML(new blippoXML())));
```

    Note that this is one line of code, but it is too long to fit in the page margins here. This one line registers the bitmap font so that it can be used for texts inside Starling textfields. With that in mind, we must remember to include the following import statements in the imports section of the ***Assets*** class:

```
import starling.text.BitmapFont;
import starling.text.TextField;
import starling.textures.Texture;
import starling.textures.TextureAtlas;
```

    If these aren't there, add them to the ***Assets*** class in the imports section. We want to be working with Starling classes, NOT standard Flash classes, so we must

always remember to include the Starling classes in the import sections of our classes. Save All.

17. Now we can begin building the action of the game. First we will create a background class which can handle the scrolling background. We have two parts to our background, ***sky.png*** and ***skyClouds.png***. Only ***skyClouds*** will actually scroll. Go File > New, then choose Action Script 3 class, and type ***Background*** in the name box. Add the following imports, in the import section:

```
import core.Assets;
import starling.display.Image;
import starling.display.Sprite;
```

Add these class variables, just above the constructor:

```
private var sky:Image;
private var skyClouds1:Image;
private var skyClouds2:Image;
```

In the constructor, type the following:

```
sky = new Image(Assets.skyTexture);
addChild(sky);

skyClouds1 = new Image(Assets.skyCloudsTexture);
addChild(skyClouds1);

skyClouds2 = new Image(Assets.skyCloudsTexture);
skyClouds2.y = - 960;
addChild(skyClouds2);
```

These image variables allow us to move the images on the stage. So we need a function to scroll the skyClouds on the stage. Here it is:

```
public function update():void
{
    skyClouds1.y += 4;
    if (skyClouds1.y == 960) skyClouds1.y = - 960;
    skyClouds2.y += 4;
    if (skyClouds2.y == 960) skyClouds2.y = - 960;
}
```

This update function keeps the skyClouds scrolling down by 4 pixels per call, but since there are two of them offset by 960 pixels vertically, there will always be clouds in the sky. When one of the goes off the stage (== 960), it get cycles back to the top of the stage by resetting vertically it to -960.

One further thing is to change the class declaration to

```
public class Background extends Sprite
```

We need to tell the Flash compiler which main class our class belongs to, in this case the Starling Sprite class, if we want to use the standard functions of that class.

Save the ***Background.as*** code, but in the save dialog box, click the New Folder button, make a new folder inside your ***code*** folder, and call it ***objects***. Save ***Background.as*** into this ***objects*** folder. Again, make sure that on the first line of the class, you add the word ***objects*** after the word ***package,*** if it's not there. Then save the file again.

Now we will create an instance of the ***Background*** class in the ***Menu*** class and get the background scrolling! So open the ***Menu*** class and add to the import section:

```
import objects.Background;
```

Add to the class variables:

```
private var background:Background;
```

And in the ***init*** function, add:

```
background = new Background();
addChild (background);
```

Finally, in the ***update*** function,

```
background.update();
```

Which calls the update function in the Background class to do the scrolling of the clouds. Save All.

18. We can now run our game for the first time. Use command-return to run the game. Check for any errors and correct them against the code in this handout (typos do occur). If Flash asks for the location of the Flex-SDK, point it here:

Applications/Adobe Flash CS6/Common/Configuration/Action Script 3.0/flex_sdk/4.6.0

You should see the sky bitmap with the clouds scrolling vertically over it.

19. We now want to put the opening title on the stage of the game during the Menu state. So in the ***Menu*** class we add an import to the import section to handle image variables:

```
import starling.display.Image;
```

Add an image variable to the class variables section:

```
private var logo:Image;
```

Then in the ***init*** function, after the addChild, type:

```
logo = new Image(Assets.txtreAtlas.getTexture("logoMyGame"));
logo.pivotX = logo.width * 0.5;
logo.scaleX = 0.31;
logo.scaleY = 0.31;
logo.x = 320;
logo.y = 250;
```

```
addChild(logo);
```

The pivotX property allows us to set the pivot for the image sprite wherever we like, in this case, at its horizontal center. The scaleX and scaleY properties allow us to adjust the size of the image sprite on the stage (scale is between 0 and 1).

Save All, and use command-return to run the game. You should see the logo over the scrolling clouds.

20. We will now put the play button on the stage, using the built-in Starling button class. First we want to put two more imports in the import section of the ***Menu*** class:

```
import starling.display.Button;
import starling.events.Event;
```

Put a Button variable called ***play*** in the class variables section:

```
private var play:Button;
```

Just under the addChild for the logo, type the following:

```
play = new Button(Assets.txtreAtlas.getTexture("playButtonMyGame"));
play.addEventListener(Event.TRIGGERED, onPlay);
play.pivotX = play.width * 0.5;
play.scaleX = 0.31;
play.scaleY = 0.31;
play.x = 400;
play.y = 450;
addChild(play);
```

The first line creates a new button using the image in the txtreAtlas which corresponds to our original playButtonMyGame.png. The next line adds an event listener which is listening for a special Starling type of event, TRIGGERED, which only applies to Starling classes. It calls a function named ***onPlay***. We need to create this function so the listener will work. Type the following after the end of the ***init*** function, but before the ***update*** function:

```
private function onPlay(evnt:Event):void
{
   game.changeState(Game.PLAY_STATE);
 }
```

With this ***onPlay*** function, we call the changeState function in the Game class, and tell it to switch to the Play state. So if the button is clicked, we switch to the Play state, and the user can begin to play.

Save All, and use command-return to run the game. You should see the logo and play button over the scrolling clouds. You can click on the button, but since we have not done anything to the Play state, everything just stops.

21. The last thing to do to finish the ***Menu*** class is to write the code for the ***destroy*** function. Starling has a unique method not found in ordinary Action Script 3 which is

very convenient for cleaning up a class. It is the ***removeFromParent()*** method. Put the following code inside the destroy function:

```
background.removeFromParent(true);
background = null;

logo.removeFromParent(true);
logo = null;

play.removeFromParent(true);
play = null;

removeFromParent(true);
```

The first six lines remove the three sprites we put on the stage in the ***Menu*** class, the ***background***, the ***logo*** and the ***play*** button. The final line removes the current ***Menu*** class from the stage and disposes of it. Memory management at work!

Save All, and use command-return to run the code. Now clicking on the ***play*** button empties the stage. It also loads a copy of the ***Play*** class, but since we have done nothing with that, there is nothing to see. This completes the ***Menu*** class.

22. We next move to the ***Play*** class. This is the most complex state class since it contains all of our game play.

    We need to put the scrolling clouds background on the stage in the Play state. We do this exactly the same way as we put it in in the Menu state. So open the ***Play*** class and add to the import section:

    ```
    import objects.Background;
    ```

    Change ***private var game:Game*** class variable to a ***public*** variable, then add:

    ```
    private var background:Background;
    ```

    And in the ***init*** function, add:

    ```
    background = new Background();
    addChild (background);
    ```

    Finally, in the ***update*** function,

    ```
    background.update();
    ```

    Which calls the update function in the Background class to do the scrolling of the clouds.

    Save All, and use command-return to run the code. Now clicking on the ***play*** button destroys the Menu class and loads a copy of the ***Play*** class. That's why you should see the scrolling clouds background.

23. The next thing we will create is our ***Bunny*** class. Select File > New, then Action Script 3 Class. Type ***Bunny*** (caps here are important) in the Name box, then click

OK. Save this in the ***objects*** folder inside the ***code*** folder, but before you save it, make sure you type ***objects*** just after the word ***package*** on the first line of the code.

In the import section, type:

```
import core.Assets;
import starling.core.Starling;
import starling.display.Image;
import starling.display.Sprite;
import starling.events.Event;
import starling.events.Touch;
import starling.events.TouchEvent;
import starling.events.TouchPhase;
import states.Play;
import flash.geom.Point;
```

Add the words ***extends Sprite*** (caps are significant here), after ***public class Bunny*** and before the left brace ***{*** . Then, add two class variables,

```
private var posNow:Point;
private var touch:Touch;
private var play:Play;
```

Inside, the ***Bunny*** class should look like so,

```
public function Bunny(play:Play)
{
   // constructor code
   this.play = play;
   var img:Image = new Image(Assets.txtreAtlas.getTexture("bunny"));
   img.pivotX = img.width * 0.5;
   img.pivotY = img.height * 0.5;
   img.scaleX = 0.31;
   img.scaleY = 0.31;
   addChild(img);
   addEventListener(Event.ADDED_TO_STAGE, onAdded);
}

private function onAdded(evnt:Event):void
{
   stage.addEventListener(TouchEvent.TOUCH, touchHandler);
}

private function touchHandler(evnt:TouchEvent):void
{
   touch = evnt.getTouch(stage);
   If (touch != null) posNow = touch.getLocation(stage);
}
```

The above code does the following: 1) the imports give us access to the Starling code we are using, especially the Touch-related classes; 2) the variable ***play*** allows us to access the functions and variables of the ***Play*** class; 3) the ***Bunny*** constructor places the bunny image on the starling display list, then it adds a listener to the stage to see when it has been added -- if we didn't have this listener we would get runtime errors; 4) the ***onAdded*** function adds an event listener to the stage for touch

events; and 5) the getStageCoords function finds the x and y position of the Touch event, and stores it in ***posNow***.

Below the ***Bunny*** constructor function, add an ***update*** function:

```
public function update():void

{
    x += (posNow.x - x) * 0.3;
    y += (posNow.y - y) * 0.3 - 20;
}
```

The ***update*** function makes the ***Bunny*** class follow the touch coordinates, but offset so we can see the bunny as we move it with our fingers.

Now we need to make an instance of this class in the ***Play*** class, put it on the stage and get it updating itself. Switch to the ***Play*** class, and add an import,

```
import objects.Bunny;
```

And a class variable,

```
public var bunny:Bunny;
```

Make sure that ***bunny*** is a ***public var***. We will need to access its location from other classes. Then, in the ***init*** function, just below the background lines,

```
bunny = new Bunny(this);
addChild(bunny);
```

Finally, in the ***update*** function, again just below the background lines,

```
bunny.update();
```

We've now added our Bunny class to the stage and it should follow our touches. Save All, and use command-return to run the code. Now click on the ***play*** button and check if the bunny is on the stage and following your mouse/finger.

24. Our bunny will be using carrots to bust-up the clouds that stream by. So we need to create a Carrot class for that purpose. Select File > New, then Action Script 3 Class. Type ***Carrot*** (caps here are important) in the Name box, then click OK. Save this in the ***objects*** folder inside the ***code*** folder, but before you save it, make sure you type ***objects*** just after the word ***package*** on the first line of the code.

    In the import section, type:

```
import core.Assets;
import starling.display.Image;
import starling.display.Sprite;
```

    Add the words ***extends Sprite*** (caps are significant here), after ***public class Carrot*** and before the left brace ***{*** . Then, the ***Carrot*** constructor function should look like so

```
public function Carrot()
```

```
{
    // constructor code
    var img:Image = new Image(Assets.txtreAtlas.getTexture("carrot"));
    img.scaleY = -1;
    addChild(img);
    pivotX = width * 0.5;
    pivotY = height * 0.5;
}
```

And that's it for the ***Carrot*** class. We will be implementing a ***CarrotManager*** class that will do all the work managing the carrots, so this class can remain very simple.

25. Now we will make that CarrotManager class. Select File > New, then Action Script 3. Enter ***CarrotManager*** in the name box, then click OK. Next save this file, but in the save dialog box, click the New Folder button, make a new folder inside your ***code*** folder, and call it ***managers***. Save ***CarrotManager.as*** into this ***managers*** folder. Again, make sure that on the first line of the class, you add the word ***managers*** after the word ***package,*** if it's not there. Then save the file again.

    In the imports section add the following:

```
import core.Assets;
import objects.Carrot;
import states.Play;
```

    Then add two class variables,

```
public var carrots:Array;
private var play:Play;
```

    Inside the ***CarrotManager*** class should look like so

```
public function CarrotManager(play:Play)
{
    this.play = play;
    carrots = new Array();
}

public function update():void
{
}

private function chuckCarrot():void
{
}

public function removeCarrot(crt:Carrot):void
{
}

public function destroy():void
{
}
```

    We will be filling in the code for the functions in the next few steps, but these are the basic five functions a good manager class needs to have for our game.

26. In order to handle the carrots being thrown, we don't want to continually create and destroy carrots, since this slows down game performance. Instead we will create a limited number of carrots and reuse them. This is called "pooling" and is a good technique to use in cases like this.

    In the Finder, look in the following file path inside your code folder: com > pooling > starling > ***StarlingPool.as*** . Move the ***StarlingPool.as*** file into your ***core*** folder. This will make it easier to link to (we won't need to use a long file path to get to it). Go back to Flash and open the ***StarlingPool.as*** file.

    On the first line, change the words after ***package*** to the one word ***core.*** That way the Flash compiler will look for the file in the ***core*** folder. The ***StarlingPool*** class is very simple. The constructor takes two parameters: the type of class object in the pool, and the number of such objects. It creates exactly the number in the parameter of them, putting them in an array. The ***getSprite*** function returns on of the items, and subtracts one from the usage counter. If all items are in use, the counter will be at zero, and an error will be thrown. The ***returnSprite*** function checks a sprite back into the pool for reuse. The ***destroy*** function sets the pool array to null; closing it down.

    Now go back to the ***CarrotManager*** code. In the imports section add the following:

```
import core.StarlingPool;
```

    Add two class variables,

```
private var pool:StarlingPool;
public var count:int = 0;
```

    And in the constructor add,

```
pool = new StarlingPool(Carrot, 100);
```

    This creates a pool of 100 carrot objects for our bunny to chuck. In the ***chuckCarrot*** function, type the following:

```
var crt:Carrot = pool.getSprite() as Carrot;
play.addChild(crt);
crt.x = play.bunny.x;
crt.y =  play.bunny.y + 15;
carrots.push(crt);
```

    This gets a carrot from the pool, adds it to the ***Play*** class, positions it initially next to the ***bunny*** and then pushes it onto the carrots array-stack so we can keep track of it.

    Now turn attention to the ***update*** function. In the ***update*** function, put the following:

```
var crt:Carrot;

var len:int = carrots.length;
for(var i:int=len-1; i>=0; i--)
{
   crt = carrots[i];
   crt.y -= 10;
```

```
            if (crt.y < 0) removeCarrot(crt);
        }

        if (play.chuck && count%30 == 0) chuckCarrot();

        count++;
```

Here the carrots that have been put on the stage are moved vertically by -25 pixels per frame. If the carrot has gone off the stage (***crt.y*** < 0), then it is removed. The if-statement checks to see if ***count*** is evenly divisible by 10; if so then it chucks a carrot. This is so it doesn't chuck carrots every frame (that would be 60 carrots per second!), but only every 30th frame. Finally ***count*** is increased by 1.

27. The last two functions of the ***CarrotManager*** are ***removeCarrot*** and ***destroy***. Here is the code for those two functions:

```
public function removeCarrot(crt:Carrot):void
{
    var len:int = carrots.length;
    for(var i:int=0; i<len; i++)
    {
        if (carrots[i] == crt)
        {
            carrots.splice(i,1);
            crt.removeFromParent(true);
            pool.returnSprite(crt);
        }
    }
}

public function destroy():void
{
    pool.destroy();
    pool = null;
    carrots = null;
}
```

The ***removeCarrot*** function goes through the ***carrots*** array, looking for the particular ***crt*** that it was passed. When it finds it, it splices it out of the array, removes it from its parent (the ***Play*** class), and returns it to the pool for reuse. The ***destroy*** function destroys the ***pool*** and sets both it and the ***carrots*** array to null.

This finishes the ***CarrotManager*** class, so Save All.

28. We want the user to control when carrots are chucked by placing a second finger on the touchscreen. The ***Play*** class has to be involved here. So open the ***Play*** class, and add two class variables,

```
public var carrotManager:CarrotManager;
 public var chuck:Boolean = false;
```

The first is an instance of our ***CarrotManager*** class, which the ***Play*** class will use to mange the carrots that get chucked. We set ***chuck*** to false initially since the player has to choose to chuck a carrot for it to be true.

Now in the ***Play*** class ***init*** function, add the following line:

```
carrotManager = new CarrotManager(this);
```

And in the ***update*** function add,

```
carrotManager.update();
If ( ! chuck) carrotManager.count = 0;
```

We also have to make some changes in the ***Bunny*** class to pick up the multitouch events. Right now it only pays attention to the first touch, ignoring later touches. Open ***Bunny*** class and add the following to the class variables:

```
private var touches:Vector.<Touch>;
```

In the ***touchHandler*** function, below the ***if (…***, add this code:

```
touches = evnt.touches;
If (touches.length > 1)
{
   play.chuck = true;
}  else
{
   play.chuck = false;
}
```

This code sets up a Starling Vector called ***touches*** to store all the touches associated with the current Touch event. We then check its length to see if it has 2 or more touches. If so we reach into the ***play*** class, and set ***chuck*** to true; if no, then we make sure that ***chuck*** is set to false. Then when the ***update*** function is called in the ***play*** class, it will call the ***carrotManager.update()*** which in turn will check ***play.chuck***, and chuck carrots if it is true. So we've now easily handled our multitouch input. Save All. Use command-return, to chuck a few carrots!

29. We move now to making the clouds classes. Make two new Action Script 3 classes, and call them ***Cloud01*** and ***Cloud02***.They are exactly the same, except that the one uses the Cloud01 sprite animation sequence, while the other uses the Cloud02 sprite animation sequence.

    These classes extend MovieClip instead of Sprite in order to take advantage of the built-in timeline that a MovieClip has and a Sprite doesn't. This saves us hours of drudgery having to create timelines for our sprites (as we were forced to do in Unity). By using a MovieClip, we can simply load our sprite animation sequences from the Texture Atlas using the ***getTextures()*** function, and they are ready to go! Here is the code for the two classes:

Cloud01

```
package objects
{
  import starling.display.MovieClip;
  import starling.textures.Texture;

  import core.Assets;

  public class Cloud01 extends MovieClip
  {
    public function Cloud01()
    {
super(Assets.txtreAtlas.getTextures("Cloud01_
"), 8);
      pivotX = width * 0.5;
      pivotY = height * 0.5;
    }
  }
}
```

Cloud02

```
package objects
{
  import starling.display.MovieClip;
  import starling.textures.Texture;

  import core.Assets;

  public class Cloud02 extends MovieClip
  {
    public function Cloud02()
    {
super(Assets.txtreAtlas.getTextures("Cloud02
_"), 8);
      pivotX = width * 0.5;
      pivotY = height * 0.5;
    }
  }
}
```

Place this code into your ***Cloud01*** and ***Cloud02*** Action Script classes, replacing all of the default code, and save them into the ***objects*** folder.

30. In order to manage our cloud classes on the stage, we will write a ***CloudManager*** class, similar to the ***CarrotManager*** class. Create a new Action Script 3 class, and name it ***CloudManager*** (caps are important here). The ***CloudManager*** will be based on the ***CarrotManager,*** so open up the ***CarrotManager*** class, and hit command-A to select the whole class' code. Copy it, then deselect, and close the ***CarrotManager*** class. Switch back to the ***CloudManager*** class, hit command-A to select its whole code, then paste the code you just copied from the ***CarrotManager***, replacing all of the default code originally in the ***CloudManager*** class.

    Now change the class name from ***CarrotManager*** to ***CloudManager***, and the constructor function name from ***CarrotManager*** to ***CloudManager*** as well. Save this in the ***managers*** folder.

    Use the Find & Replace (command-F) to make the following changes:

    Replace ***carrots*** with ***clouds*** as the name of the Array.

    Replace ***Carrot*** with ***Cloud01*** (we will only use one of the cloud classes at this point).

    Replace ***crt*** with ***cld*** as the local variable we use in several places.

    Next, in the constructor, change the number of items in the ***pool*** from 100 to 15. We may need 100 carrots for the ***CarrotManager***, but we probably only want 15 or so clouds on the stage at one time. That will be plenty for the ***CloudManager pool***.

    In the imports section of the class, add

    import starling.core.Starling;

Replace the ***chuckCarrot*** function with the following ***makeCloud*** function:

```
private function makeCloud():void
{
   var cld:Cloud01 = pool.getSprite() as Cloud01;
   Starling.juggler.add(cld);
   clouds.push(cld);
   cld.y = -50;
   cld.x = Math.random() * 700 + 50;
   play.addChild(cld);
}
```

Note the ***Starling.juggler.add(...)*** function in this last function. The ***Starling.juggler*** is a Starling class which updates the timelines of all MovieClips added to it. It assures us that our clouds will be animating as they appear on the stage.

The ***update*** function needs to be changed as well. Change it to the following:

```
public function update():void
{
   // this next line determines how often we make clouds (5% of the time)
   if (Math.random() < 0.05) makeCloud();
   var cld:Cloud01;
   var len:int = clouds.length;
   for(var i:int=len-1; i>=0; i--)
   {
      cld = clouds[i];
      cld.y += 8;
      if (cld.y > 960) removeCloud(cld);
   }

}
```

removeCloud01 needs changing also. Here is what it should be changed to:

```
public function removeCloud(cld:Cloud01):void
{
   var len:int = clouds.length;
   for (var i:int=0; i<len; i++)
   {
      if (clouds[i] == cld)
      {
         clouds.splice(i,1);
         Starling.juggler.remove(cld);
         cld.removeFromParent(true);
         pool.returnSprite(cld);
      }
   }
}
```

Save All.

Now go back to the ***Play*** class, so we can incorporate the ***CloudManager*** into our Play state. In the ***import*** section of the ***Play*** class, add the following:

```
import managers.CloudManager;
```

Then add the class variable,

```
public var cloudManager:CloudManager;
```

And in the ***init*** function of the ***Play*** class

```
cloudManager = new CloudManager(this);
```

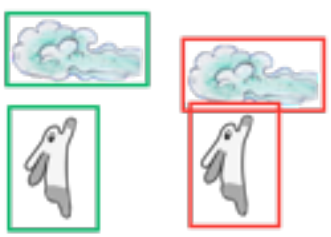
Lastly, in the ***update*** function of the ***Play*** class, add,

```
cloudManager.update();
```

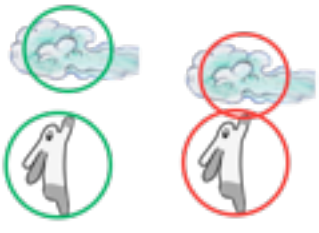
Save All. Use command-return, to chuck a few carrots and see the clouds roll by. Now we need to be able to chase the clouds away with our carrots!

31. Now for collision detection. When our carrots hit a cloud, we want the cloud to dissipate. So we need to be able to test for a collision between the carrots and the clouds. Also, if the bunny runs into a cloud we want to him to cough (10 coughs ground the bunny). So we also need to be able to detect collisions between the clouds and the bunny.

    In the spirit of our past efforts, we will create a ***CollisionManager*** class to manage the detection of these collisions. There are two basic methods for collision detection in Starling: sprite bounding box and distance-based. The first tests the outer box of two sprites to see if the have overlapped, if they have then there has been a collision. This may not be accurate depending on the bounding boxes of each sprite. The second distance-based is more accurate and is based on an circle around a sprite. The distance between two sprites is calculated, and then that distance is compared to the sum of two radii of the circles around the two sprites. If the distance is less than the sum of the radii, then a collision has occurred. The circles are not actually created in the code or in the sprites, they are simply imagined with a set radius that we decide. Here are two examples:



Bounding Box Collision



Distance-based Collision

32. If is faster to calculate distance-based collisions than bounding box ones, and much of the time it is at least as accurate. Given this, we will use distance-based collisions in out ***CollisionManager*** class. Creat a new Action Script 3 class and name it ***CollisionManager***. On the first line of the ***CollisionManager*** code, type ***package***, ***managers*** and after the word the ***managers*** folder, then save it in the folder.

In the import section put the following:

```
import flash.geom.Point;
import core.Assets;
import objects.Cloud01;
import objects.Carrot;
import core.Game;
import states.Play;
```

Then we will need class variables, so type the following inside the ***CollisionManager*** class:

```
private var play:Play;
private var p1:Point = new Point  ;
private var p2:Point = new Point  ;
private var count:int = 0;
private var coughCount:uint = 0;
```

In order to track collisions, we will need access to the ***carrots*** array in the ***CarrotManager***, and the ***clouds*** array in the ***CloudManager***. We can access these through the ***Play*** class, since that's were they get instantiated as objects. So in the constructor, we need to have the play object passed to us, then set our local play variable to it, like so:

```
public function CollisionManager(play:Play)
{
   this.play = play;
}
```

We will create a public ***update*** function that calls two different functions to check the two types of collisions, ***carrotsAndClouds*** or ***bunnyAndClouds***. Like this:

```
public function update():void
{
   carrotsAndClouds();
   bunnyAndClouds();
}
```

Both functions will go through the ***clouds*** array comparing it with the other objects (either the bunch of carrots or the one bunny) that are on the stage. The difference is that in the ***carrotsAndClouds*** function, we will need to compare all the clouds with each carrot to see if there's a collision (using the distance-based collision approach).

So we need to got through the entire ***cloud*** array for each item in the ***carrots*** array, comparing the distances between them. Here is the code to do that:

```
private function carrotsAndClouds():void
{
   var aCrt:Array = play.carrotManager.carrots;
   var aCld:Array = play.cloudManager.clouds;

   var crt:Carrot;
   var cld:Cloud01;

   for (var i:int=aCrt.length-1; i>=0; i--)
   {
      crt = aCrt[i];
      for (var j:int=aCld.length-1; j>=0; j--)
      {
         cld = aCld[j];
         p1.x = crt.x;
         p1.y = crt.y;
         p2.x = cld.x;
         p2.y = cld.y;
         if (Point.distance(p1,p2) < cld.pivotY + crt.pivotY)
         {
            play.cloudManager.removeCloud(cld);
            play.carrotManager.removeCarrot(crt);
         }
      }
   }
}
```

We have nested for loops here, the outer one counts through the ***carrots*** array, while the inner one counts through the ***clouds*** array. Inside the outer loop, we set a local variable ***crt*** to a particular item in the ***aCrt*** array (which is our local copy of the ***carrots*** array in the ***CarrotManager*** class). Then inside the inner loop we compare the distance between that particular carrot to all the clouds in the ***clouds*** array. We do this by first storing a particular item in the ***aCld*** array in a local variable, ***cld*** (***aCld*** is our local copy of the ***clouds*** array in the ***CloudManager*** class). Then we compare the distance between ***crt*** and ***cld***, using two point-class variables ***p1*** and ***p2***. The point-class variables have built-in very fast distance functions. If they have come closer than the sum of their ***pivotY*** values, then we count them as collided. The ***pivotY*** value for a sprite is half its vertical height. Since the carrots and clouds all travel only vertically, using the ***pivotY*** value is the right value to use for collision. If we find a collision, we simply remove both of the items from the game.

The ***bunnyAndClouds*** function needs only to compare all the clouds with the one bunny. We will only need on for loop to do that. Here is that code:

```
private function bunnyAndClouds():void
{
   var aCld:Array = play.cloudManager.clouds;
   var cld:Cloud01;
```

```
        for (var i:int=aCld.length-1; i>=0; i--)
        {
            cld = aCld[i];
            p1.x = play.bunny.x;
            p1.y = play.bunny.y;
            p2.x = cld.x;
            p2.y = cld.y;
            if (Point.distance(p1,p2) < cld.pivotY + play.bunny.pivotY)
            {
                // the bunny has hit a cloud and should
                //  play the cough sound here
                coughCount++;
                if (coughCount > 10)
                {
                    play.game.changeState(Game.GAME_OVER_STATE);
                }
            }
        }
    }
```

This is similar to the ***carrotsAndClouds*** function, but only compares the items in ***aCld*** (our local copy of the ***clouds*** array in the ***CloudManager*** class) to the one ***bunny***, checking the distance between them. If the distance is less than the sum of their ***pivotY*** values, then we count them as collided. However, we don't remove the bunny or the cloud, we just keep track of the number of clouds that the bunny has passed through. If that goes over 10, then we change the game state to GameOver.

Now we need to instantiate the ***CollisionManager*** class in the ***Play*** class so it can do what it was meant to do: detect collisons! So switch to the ***Play*** class and put the following in the ***imports*** section:

```
import managers.CollisionManager;
```

Then add the class variable,

```
public var collisionManager:CollisionManager;
```

And in the ***init*** function of the ***Play*** class

```
collisionManager = new CollisionManager(this);
```

Lastly, in the ***update*** function of the ***Play*** class, add,

```
collisionManager.update();
```

Save All. Use command-return, to chuck a few carrots at the clouds. Try to keep the bunny out of the clouds! If the bunny passes through 10 clouds, we go to the GameOver state … oh, wait, we haven't completed that state yet ...

33. Now to finish the FSM, by filling out the GameOver state. Open the ***GameOver*** class. Put the following in the ***imports*** section of the class:

```
import core.Assets;
import starling.display.Button;
```

```
import starling.text.TextField;
import objects.Background;
```

Add to the class variables:

```
private var background:Background;
```

And in the ***init*** function, add:

```
background = new Background();
addChild (background);
```

Finally, in the ***update*** function,

```
background.update();
```

Save All. If we use command-return, have the bunny pass through 10 clouds, we go to the GameOver state which now shows a scrolling background. Empty, of course.

So we want to provide an end-of-game message to the user, and put a play again button on the stage so the user can tap it to play again. In order to do this start by making two more class variables:

```
private var gameOvertext:TextField;
private var playAgain:Button;
```

Add the following code to the ***init*** function,

```
gameOverText = new TextField(620, 400, "Game\nOver", "BlippoBT-Black", 128);
gameOverText.hAlign = "center";
gameOverText.y = 200;
addChild(gameOverText);

playAgain = new Button(Assets.txtreAtlas.getTexture("playAgainButtonMyGame"));
playAgain.addEventListener(Event.TRIGGERED, playItAgain);
playAgain.pivotX = playAgain.width * 0.5;
playAgain.scaleX = 0.31;
playAgain.scaleY = 0.31;
playAgain.x = 320;
playAgain.y = 650;
addChild(playAgain);
```

The first line of this code makes a new Starling Textfield, and puts the text “Game Over” in it, but uses our bitmap font Blippo to do it. We use the name of the font that is in the ***blippo.fnt*** file. Recall that that file is really an XML file, and inside of it is a tag <info> which contains basic information about the font, including the all important reference name for the font. In this case it is “BlippoBT-Black” which you can find out by opening the ***blippo.fnt*** file in any text editor like TextWrangler or TextEdit. We then center it horizontally in the Textfield, position it, and finally add it to the display.

The second section of text is similar to the code we used in the ***Menu*** class to handle the play button. This time, however, it calls the ***playItAgain*** function when clicked. So we need to create that function as well. Here it is:

```
private function playItAgain(event:Event):void
```

```
{
    playAgain.removeEventListener(Event.TRIGGERED, playItAgain);
    game.changeState(Game.PLAY_STATE);
}
```

This should go right after the ***init*** function, but before the ***update*** function.

Finally, put the following in the destroy function

```
removeFromParent(true);
```

This acts to clean up the GameOver state when it is no longer needed.

Save All. Now use command-return, have the bunny pass through 10 clouds, and see the GameOver state which now is fully functional. Click the button to play again!

34. The last fundamental thing we need to do is to fill out the ***destroy*** functions we have left empty. The ***Menu*** class needs the following code in its ***destroy*** function:

```
background.removeFromParent(true);
background = null;

logo.removeFromParent(true);
logo = null;

play.removeFromParent(true);
play = null;

removeFromParent(true);
```

This removes all the objects we put on the stage in the ***Menu*** class, then removes it from its parent (namely, the ***Game*** class instance).

The ***Play*** class destroy function needs the following in it:

```
carrotManager.destroy();
cloudManager.destroy();
removeFromParent(true);
```

This destroys the two manger objects and then removes the ***Play*** class instance from its parent (the ***Game*** class instance).

Finally, the GameOver class needs this code in its ***destroy*** function:

```
removeFromParent(true);
```

35. Our ***Code:B*** game is now fully functional, but it needs some additional razzle-dazzle to make it better. Audio would be the number one thing to add, but also some particle effects, like puffing the clouds away instead of just removing them, also some effect for the carrots and bunny. So let's go to work!

36. We will use two sound FX files and a music file in our audio. First we'll embed the audio files. On the server in our usual folder find the ***Starling Stuff*** folder. In that folder is a zip archive called Audio. Copy it to your Mac and unzip it. There are three .mp3s in the Audio folder, ***audio_ambiMusic.mp3***, ***audio_cough.mp3***, ***audio_puff.mp3***. Drag these into the ***assets*** folder inside the ***core*** folder.

Switch to Flash and open the ***Assets*** class. To embed our three .mp3s, place the following code just under the earlier embeds:

```
[Embed(source = "assets/audio_RSakamotoPutYourHandsUpLoop.mp3")]
private static var musicSound:Class;
public static var music:Sound;

[Embed(source = "assets/audio_cough.mp3")]
private static var coughSound:Class;
public static var cough:Sound;

[Embed(source = "assets/audio_puff.mp3")]
private static var puffSound:Class;
public static var puff:Sound;
```

Then, in the ***init*** function, just below ***TextField.registerBitmapFont(...***, place:

```
music = new musicSound();
var musicChannel:SoundChannel = new SoundChannel();
musicChannel = music.play(0, 0, new SoundTransform(0));

cough = new coughSound();
cough.play(0, 0, new SoundTransform(0));

puff = new puffSound();
puff.play(0, 0, new SoundTransform(0));
```

Each of these pairs of statements does two things: 1) creates a new sound from the corresponding Class file, and 2) plays the sound, but with the volume set to zero. This second step is important, since in order to ensure that our sounds play without latency in the game, we must have them loaded into memory. By playing them as soon as we create them, we make sure they are loaded and ready to play during the game. We use a SoundChannel for the music since it is the only one that must loop.

One last thing in the ***Assets*** class, is to add the Flash sound libraries to the imports section of the class. We do this by adding these lines to the imports section:

```
import flash.media.Sound;
import flash.media.SoundTransform;
import flash.media.SoundChannel;
```

Now we need only add a little code in the spots in the game where we want to play the sounds. Primarily, we want the clouds to puff when they disappear, and the bunny to cough when it passes through a cloud. We detect both of these in the ***CollisionManager*** class. So open the ***CollisionManager***, and in the ***bunnyAndClouds*** function, just below the comment line, ***// the bunny …***, add the following:

```
Assets.cough.play();
```

This places a playing of the cough whenever the bunny collides with a cloud.

Next, in the ***carrotsAndClouds*** function, inside the if-statement, add,

```
Assets.puff.play();
```

This places a playing of the puff sound whenever a carrot dissipates a cloud.

Finally, we start the game music playing in the ***Game*** class. We don't want to do it in one of the game state classes since they get destroyed as the state changes. We also want the music to play as long as the game does, then stop. We don't want the music to continue to play AFTER the user has exited the game!

Open the ***Game*** class, and add these lines to the import section of the class:

```
import flash.media.SoundTransform;
import flash.media.SoundChannel;
import flash.utils.getTimer;
```

The first two are familiar sound libraries from Flash, while the last imports a Flash utility for getting the current time since the game started (in milliseconds). We will use this to loop the music.

Four new class variables need to be added as well:

```
private var musicTimer:int;
private var musicLength:int;
private var musicChannel:SoundChannel = new SoundChannel();
private var musicTransform:SoundTransform = new SoundTransform(0.2);
```

In the ***init*** function of the ***Game*** class, place the following:

```
musicTimer = getTimer();
musicLength = Assets.music.length;
musicChannel = Assets.music.play(0,0,musicTransform);
```

This sets the start time of the music in the ***musicTimer*** variable, the time length of the music in ***musicLength***, and plays the music through the ***SoundChannel*** called ***musicChannel*** using the ***SoundTransform*** called ***musicTransform***.

Next we turn to the ***update*** function, and we place some code in it to check if the music needs to be restarted to keep it looping. Here is that code:

```
if (getTimer() - musicTimer > musicLength)
{
   musicTimer = getTimer();
   musicChannel.stop();
   musicChannel = Assets.music.play(0,0,musicTransform);
}
```

This checks if the current time (***getTimer()*** gives us the current time) minus the time the music started (stored in ***musicTimer***) is greater than the length of time for the music. If that's true, then the music has run out and we need to start it again. We

reset the ***musicTimer*** variable to the new start time of the music, stop the ***musicChannel*** (just to be safe), and begin to play the music again.

Save All. Press command-return. All the audio should now be working!

One thing to mention: a great, free tool for making old-timey sounding game sounds is ***cfxr*** (http://thirdcog.eu/apps/cfxr). The sounds it makes are reminiscent of older style games (think SuperMario), but for bleeps, buzzes and arcade-style sounds it's great! Check it out!

37. Particle FX add visual razzle-dazzle to motion or action in a game. Starling easily incorporates particle effects through the particle effect extension, which we downloaded in step 2 and put in the ***starling*** folder. Check to see if you have this installed. It consists of the following Action Script classes: ***ColorArgb.as, Particle.as, ParticleDesignerPS.as, ParticleSystem.as, PDParticle.as, PDParticleSystem.as***. These should be in the ***starling > extensions*** folder in order for the next steps to work.

    Designing particles is always challenging and fun. We will use the inexpensive, but still payware, ***Particle Designer*** from 71squared. An online, free, particle design editor (written in Action Script, btw), can be found at http://onebyonedesign.com/flash/particleeditor . This does everything that ***Particle Designer*** does, but a little bit clumsier. But, hey, it's free.

    Open up ***Particle Designer***. We have two windows, an iPhone example window and a particle design/library window. Have a look through the library. Select a particle in the library that you want to explore, and click ***Emitter Config***. Here you can fiddle with the settings to customize your particle effect. Click the ***Load*** button and navigate to the ***assets*** folder inside the ***code*** folder. Open up the ***puffReddish.pex*** file. Click the ***Play*** button to see it. Will use this particle effect when the clouds are hit by a carrot and dissipate. Click the ***Load*** button and navigate to the ***assets*** folder inside the ***code*** folder. Open up the ***bunnyTrail.pex***, and click ***Play***. We will use this effect for the trail of the bunny as it flies on the stage. Quit ***Particle Designer.***

    The two ***.pex*** files we just created are just XML files. We need to embed them in the ***Assets*** class in order to use them. Since we are using them in different ways, we will handle each one separately here in the tutorial. We will embed the ***bunnyTrail.pex*** first. In the ***Assets*** class, after the last audio embed, put the following code:

```
[Embed(source = "assets/bunnyTrail.pex",mimeType = "application/octet-stream")]
public static var bunnyTrailXML:Class;
```

    Now switch to the ***Bunny*** class, since that's where the trail will be generated. Add the following import:

```
import starling.extensions.PDParticleSystem;
```

    And the following class variable:

```
private var bunnyTrail:PDParticleSystem;
```

Then in the constructor add,

```
bunnyTrail = new PDParticleSystem(XML(new Assets.bunnyTrailXML()),
Assets.txtreAtlas.getTexture("bunnyTrail"));
```

Note that this is really one line of code, it just does not fit in the margins of this page. This creates the particle system that we can now use to make our particle trail for the bunny. It takes two parameters, an XML file describing the particle effect (in this case ***bunnyTrailXML***) and a Texture file (which we get from our Texture atlas, ***txtreAtlas***). Next we add the following code just under that line:

```
Starling.juggler.add(bunnyTrail);
play.addChild(bunnyTrail);
bunnyTrail.start();
```

This adds the particle system to the animator manager (the ***Starling.juggler*** class), then puts it on the stage as part of the ***play*** class instance, and finally starts the ***bunnyTrail*** particle system rolling.

The last thing we need to do is add the following code to the ***update*** function of the ***Bunny*** class:

```
bunnyTrail.emitterX = x;
bunnyTrail.emitterY = y + 45;
```

This code places the bunnyTrail.emitter exactly at the hindquarters of the bunny. Save All. Use command-return to see your bunny emitting its ***bunnyTrail***!

38. The Puff when a cloud is hit by a carrot we will handle differently since we may need to display several particle emitters at one time. So we will create a subclass of the ***PDParticleSystem*** to allow us to instantiate as many as we need.

    But first we need to return to the ***Assets*** class and embed the ***puffReddish.pex*** XML file into the assets class. We do this exactly as we did the bunnyTrail.pex file:

```
[Embed(source = "assets/puffReddish.pex",mimeType = "application/octet-stream")]
public static var puffReddishXML:Class;
```

    Next we create a new Action Script 3 class (File > New, Action Script 3) and name it ***PuffReddish*** (caps count!). In the editor, replace the default code with the following:

```
package objects
{
   import core.Assets;
   import starling.extensions.PDParticleSystem;
   import starling.textures.Texture;

   public class PuffReddish extends PDParticleSystem
   {
```

```
        public function PuffReddish()
        {
            super(XML(new Assets. PuffReddishXML()),
Assets.txtreAtlas.getTexture("puffReddish"));
        }
    }
}
```

This imports the Starling classes we need as well as our Assets class. The class is defined as extending the ***PDParticleSystem***, and the constructor uses the super constructor to pass the ***PuffReddishXML*** description of the particle effect, and the Texture, ***puffReddish***, from the texture atlas, ***txtreAtlas*** to the parent class. The parent class is of course, just ***PDParticleSystem***. So this gives a class, ***PuffReddish,*** that we can instantiate whenever we need a reddish puff!

Next, we make a PuffManager class that will manage the ***PuffReddish*** classes as they are created and destroyed. File > New, Action Script 3 and name it ***PuffManager*** (caps count!). In the editor, replace the default code with the following:

```
package managers {

    import starling.core.Starling;
    import starling.events.Event;

    import core.StarlingPool;
    import objects.PuffReddish;
    import states.Play;

    public class PuffManager
    {
        private var play:Play;
        private var pool:StarlingPool;

        public function PuffManager(play:Play)
        {
            this.play = play;
            pool = new StarlingPool(PuffReddish, 15);
        }




        public function makePuff(x:int, y:int):void
        {
            var pr:PuffReddish = pool.getSprite() as PuffReddish;
            pr.maxNumParticles = 30;
            pr.endSize = 20;
            pr.emitterX = x;
            pr.emitterY = y;
            pr.start(0.1);
            play.addChild(pr);
            Starling.juggler.add(pr);
```

```
            pr.addEventListener(Event.COMPLETE, onComplete);

        }

        private function onComplete(evnt:Event):void
        {
            var pr:PuffReddish = evnt.currentTarget as PuffReddish;
            Starling.juggler.remove(pr);

            if (pool != null) pool.returnSprite(pr);
        }

        public function destroy():void
        {
            for(var i:int=0; i<pool.items.length; i++)
            {
                var pr:PuffReddish = pool.items[i];
                pr.dispose();
                pr = null;
            }
            pool.destroy();
            pool = null;
        }
    }
}
```

Again, we are using the ***StarlingPool*** class to give us a pool of objects to put on the stage, this time particle systems! In the constructor we assign a class variable ***play***, which gives us access to the ***Play*** state, and create the pool of 15 ***puffReddish*** particle systems. The function ***makePuff*** has two parameters, the x and y coordinates where the puff should made. In it, we first get an object from the pool and put it in the variable ***pr***. We then set the number of particles for our effect, and the end size of the particles. I did this here to show that, although we did set these in ***Particle Designer***, we can change them freely in the code. We then position the emitter, start it (with a duration of 0.1 second), add it to the stage and to the Starling ***juggler***. We then add a listener to handle when the puff has completed its time, which calls the ***onComplete*** function. In the ***onComplete*** function, we get the now defunct puff particle system from the event passed to the function, remove it from the ***juggler***, then return it to the pool of particle systems. We check to see if the pool still exists, however, since its possible we might destroy the pool before this executes.

Lastly, in the destroy function, we loop through the pool items disposing of them in order. This is important since merely destroying the pool WILL NOT dispose of the particle systems it had in it. We must dispose of all the particle systems first, then destroy the pool. This is different than the destroy function for the other managers we wrote. Save All.

Now, we need to instantiate this ***PuffManager*** in the ***Play*** class. So open the ***Play*** class and add another import

import managers.PuffManager;

And a class variable:

public var puffManager:PuffManager;

And then in the init function of the Play class add,

puffManager = new PuffManager(this);

Finally, in the ***CollisionManager,*** we want to play the puffs when the carrots collide with a cloud. We do that in the ***carrotsAndClouds*** function, so right under the ***Assets.puff.play();*** statement, add the this line of code:

play.puffManager.makePuff(cld.x, cld.y);

Save All. Use command-return to see the clouds go up in a puff when a carrot hits them!

39. Final things: We provide the Flash Native application code to exit the app gracefully. This code must go into the main class for the game, namely ***MyGame***. So open up ***MyGame.as*** to make the following changes.

    In the imports section, import the Flash desktop library to get access to the Native controls. Also import the SoundMixer library so we can stop all sounds on exit:

```
import flash.media.SoundMixer;
import flash.desktop.*;
```

    Then in the constructor place the following code after the code that's already there:

```
NativeApplication.nativeApplication.addEventListener(Event.DEACTIVATE, overAndOut, false, 0, true);
NativeApplication.nativeApplication.addEventListener(Event.EXITING, overAndOut, false, 0, true);
```

    As before when we used this code, it establishes listeners to call clean-up code when user exits the game. The listeners call a function ***overAndOut*** that does the actual clean-up. Here's the code for that function for this game:

```
public function overAndOut(evnt:Event)
{
   SoundMixer.stopAll();

   // on iOS devices there is no way for an application to quit by itself.
   // So the user must press the home button to complete the quit.
   // to make this work, the XML file for the AIR app must include
   // some additional material as well.  See
   // http://forums.adobe.com/thread/870733
```

```
        // reply 17 by gingerman for the correct code to add to the XML file

        // on all other mobile devices this line will terminate the application
        NativeApplication.nativeApplication.exit();
    }
```

Add this code after the constructor in the ***MyGame*** class. The game is now complete.

Save All. Use command-return to play ***Code B***!

40. Now use the standard ways we used in class of making the iOS ***ipa*** file for development and/or distribution!