Notes 6: Simple user keyboard interactivity

Now that we are starting to learn how to use Actionscript, we can make all sorts of interesting animations to watch: raindrops, snow, and teddy bear heads falling from the sky. But a game requires user interactivity. The most common forms of interaction are the mouse and the keyboard. Let’s start with the keyboard.

We want to control which way the ball moves on screen by using the keyboard arrows. If we push left, we want the ball to move left. Up, then up, down, then down, right, then right. Not surprisingly flash has built in keyboard support. Open and look at the following .fla file.

a_keyboard1.fla:

```Actionscript
var tempBall:MovieClip = attachMovie("ball", "b"+i, i);
var Increment:Number = 5 ;
tempBall._x = Stage.width / 2 ;
tempBall._y = Stage.height / 2 ;

onEnterFrame = function () {
    // move the ball if keys pressed
    if (Key.isDown(Key.LEFT)) tempBall._x -= Increment ;
    if (Key.isDown(Key.RIGHT)) tempBall._x += Increment ;
    if (Key.isDown(Key.UP)) tempBall._y -= Increment ;
    if (Key.isDown(Key.DOWN)) tempBall._y += Increment ;
}
```

Run the program, but do not mouse-click in the window first. Push the keyboard arrow keys. Nothing happens. This is because you need to identify flash, versus the browser, as the center of attention. This is called “focus”. To do this, move the mouse inside the stage area, cleverly delimited by the box (thanks Scott), and mouse-click. Now focus is on flash so if you hit an arrow key it works.

Every time a frame is entered a check is made to see if one of the four arrow keys is down, and if so the code to move the ball is executed.
Go to the Key class description in the actionscript dictionary part of the help menu. You will see that .isDown() is a method of the class, and Key.LEFT is a constant for the mapping of the left arrow key.

Now lets make it a big more interesting. Lets add a second ball which moves about at random motion. The player controls one ball, say the red one, and either try to evade or hit the black ball. See the following code and .fla file:

**a_keyboard2.fla:**

``` ACTIONSCRIPT
var tempBall1:MovieClip = attachMovie("redBall", "b1", 1) ;
var tempBall2:MovieClip = attachMovie("ball", "b2", 2) ;

var userIncrement:Number = 5 ; // speed of player ball

tempBall1._x = Stage.width - tempBall1._width - 20 ;
tempBall1._y = Stage.height - tempBall1._height - 20 ;
tempBall2._x = tempBall2._width + 20 ;
tempBall2._y = tempBall2._height + 20 ;

var xv2 = -8 ;
var yv2 = 4 ;

// This allows you to scale the size of the ball to your liking
var scaleFactor:Number = 60 ;
tempBall1._xscale = scaleFactor ;
tempBall1._yscale = scaleFactor ;
tempBall2._xscale = scaleFactor ;
tempBall2._yscale = scaleFactor ;

onEnterFrame = function () {
    // update new location of tempBall2
    tempBall2._x += xv2 ;
tempBall2._y += yv2 ;

    // move tempBall1 if arrow keys pressed
    if (Key.isDown(Key.LEFT)) tempBall1._x -= userIncrement ;
    if (Key.isDown(Key.RIGHT)) tempBall1._x += userIncrement ;
    if (Key.isDown(Key.UP)) tempBall1._y -= userIncrement ;
    if (Key.isDown(Key.DOWN)) tempBall1._y += userIncrement ;

    // Check boundary conditions, if tempBall1 is of stage move back (so
    // the user can not hide off stage), and if tempBall2 goes off
    // negate velocities so the ball reflects off the walls.
    if (tempBall1._x < 0) tempBall1._x = 0 ;
}
```
if (tempBall1._x > (Stage.width - tempBall1._width))
    tempBall1._x = (Stage.width - tempBall1._width);
if (tempBall1._y < 0) tempBall1._y = 0;
if (tempBall1._y > (Stage.height - tempBall1._height))
    tempBall1._y = (Stage.height - tempBall1._height);

if (tempBall2._x < 0) xv2 *= -1;
if (tempBall2._x > (Stage.width - tempBall2._width)) xv2 *= -1;
if (tempBall2._y < 0) yv2 *= -1;
if (tempBall2._y > (Stage.height - tempBall2._height)) yv2 *= -1;
}

To move towards making this a more interesting game we need to add collision code so that something happens when the player’s and computer’s balls collide. In a_keyboard3.fla we add the collision code and make the computer’s ball stop upon collision. You will see the same intersect() function that was used earlier, it simply returns true if the bounding boxes of two MovieClips intersect, and a few lines of code calling intersect and stopping the animation if they do intersect. I have highlighted in red the most important new lines of code.

a_keyboard3.fla:

    var tempBall1:MovieClip = attachMovie("redBall", "b1", 1);
    var tempBall2:MovieClip = attachMovie("ball", "b2", 2);

    var userIncrement:Number = 5;

    tempBall1._x = Stage.width - tempBall1._width - 20;
    tempBall1._y = Stage.height - tempBall1._height - 20;
    tempBall2._x = tempBall2._width + 20;
    tempBall2._y = tempBall2._height + 20;

    var xv2 = -8;
    var yv2 = 4;

    // This allows you to scale the size of the ball to your liking
    var scaleFactor:Number = 60;
    tempBall1._xscale = scaleFactor;
    tempBall1._yscale = scaleFactor;
    tempBall2._xscale = scaleFactor;
    tempBall2._yscale = scaleFactor;

    // intersect() returns true if the bounding boxes of
    // the two argument MovieClips intersect
intersect = function(mc1:MovieClip, mc2:MovieClip):Boolean {
  var lx1:Number = mc1._x ;
  var lx2:Number = mc2._x ;
  var hx1:Number = lx1 + mc1._width ;
  var hx2:Number = lx2 + mc2._width ;

  var ly1:Number = mc1._y ;
  var ly2:Number = mc2._y ;
  var hy1:Number = ly1 + mc1._height ;
  var hy2:Number = ly2 + mc2._height ;

  // if the X intervals do not intersect set xNotIntersect to true, else false
  var xNotIntersect:Boolean = ( (hx1 < lx2) || (lx1 > hx2) ) ;

  // if the Y intervals do not intersect set yNotIntersect to true,else false
  var yNotIntersect:Boolean = ( (hy1 < ly2) || (ly1 > hy2) ) ;

  // if either interval does not intersect, hit is false, else both do and
  // hence true
  if (xNotIntersect || yNotIntersect)
    return(false) ;
  else
    return(true) ;
}

onEnterFrame = function () {
  // update new location of tempBall2
  tempBall2._x += xv2 ;
  tempBall2._y += yv2 ;

  // move tempBall1 if arrow keys pressed
  if (Key.isDown(Key.LEFT)) tempBall1._x -= userIncrement ;
  if (Key.isDown(Key.RIGHT)) tempBall1._x += userIncrement ;
  if (Key.isDown(Key.UP)) tempBall1._y -= userIncrement ;
  if (Key.isDown(Key.DOWN)) tempBall1._y += userIncrement ;

  // Check boundary conditions, if tempBall1 is off stage move back (so
  // the user can not hide off stage), and if tempBall2 goes off negate
  // velocities so the ball reflects off the walls.
  if (tempBall1._x < 0) tempBall1._x = 0 ;
  if (tempBall1._x > (Stage.width - tempBall1._width) )
    tempBall1._x = (Stage.width - tempBall1._width) ;
  if (tempBall1._y < 0) tempBall1._y = 0 ;
  if (tempBall1._y > (Stage.height - tempBall1._height) )
    tempBall1._y = (Stage.height - tempBall1._height) ;
if (tempBall2._x < 0) xv2 *= -1;
if (tempBall2._x > (Stage.width - tempBall2._width)) xv2 *= -1;
if (tempBall2._y < 0) yv2 *= -1;
if (tempBall2._y > (Stage.height - tempBall2._height)) yv2 *= -1;

// now check to see if the intersect, if so stop the game
if (intersect(tempBall1,tempBall2)) {
    delete _root.onEnterFrame;
    stop();
}
}

Notes 7: Timeline manipulation from within actionscript

We can change the normal sequential timeline execution with a few actionscript commands: gotoAndPlay(), gotoAndStop(), and stop(). For games this is necessary to enable the flow from start screen, to game screen, to end screen.

First, load up the following .fla file and run it by using ctrl-Enter so you can see the trace() command outputs.

a_goto_1.fla

I set the frame rate to 1 fps so it just plays frame 1, 2, 3, 4. Then the animation ends, and of course starts up again as that is the default for flash. So, it just keeps counting 1 to 4 over and over.

By using actionscript you can change the order that frames are visited (or make it so they are never visited. Load of the following .fla file and again run by using ctrl-enter.

a_goto_2.fla

The file has four frames, they contain the following Actionscript code.

Frame 1:
trace("Inside frame 1")
gotoAndPlay(3);

Frame 2:

trace("inside frame 2")
gotoAndPlay(4);

Frame 3:

trace("inside frame 3")
gotoAndPlay(2);

Frame 4:

trace("inside frame 4")
// stop();

The command gotoAndPlay( ) causes the execution of flash to jump to the frame specified as an argument. The argument can be a frame number as used here, or a frame label in quotes. Frame labels are created in the property panel when a frame is selected. As a result, when frame 1 is played, it prints the trace command but then jumps, immediately, not waiting for the frame duration to expire, to frame 3. The code in frame 3 is immediately played and then execution jumps to frame 2, whereupon the trace command is executed and execution jumps to frame 4. All of this happens without waiting for the frame duration (determined by the frame rate) to expire. Then, we hit the end of the animation, wait for the frame duration to expire, and then the whole animation starts over again at frame 1. So, every second you see printed out the for lines showing the execution of frame 1, 3, 2, 4.

Consider what happens if you uncomment the stop( ) command on the last line of frame 4. This is what is done in:

a_goto_3.fla

The code is identical to a_goto_3.fla except frame 4 has a stop( ) command. The stop( ) command stops the execution of the MovieClip. Note, the main timeline is simply the _root MovieClip, thus the main animation is stopped. Try moving the stop to frame 3. Yep, it plays frame 1, 3, and then stops.

Now, lets add an onEnterFrame event handler to the code. Check out:

a_goto_4.fla

Run the code with ctrl-enter to see the trace command output. In a_goto_4.fla the first frame has the following code:
onEnterFrame = function() {
    trace("inside _root.onEnterFrame()")
}
trace("Inside frame 1")
gotoAndPlay(3);

The stop() has been removed from frame 4. Thus, on the first entry to frame 1, the event handler is defined, the trace command executed, and the gotoAndPlay command executed. Thus, next is frame 3, then 2, then 4. Then the frame duration expires in one second and the animation restarts with frame 1. But now the event handler is active so the message “inside _root.onEnterFrame()” is printed out, then 1, 3, 2, 4. And this cycle continues.

So, how do we stop it? Just add the stop() command back in on frame 4, right? Try it and see. Check out:

**a_goto_5.fla**

Okay, the visiting of frame 1, 3, 2, 4, only occurs once, but the onEnterFrame event handler keeps getting executed every second! This is because the stop() command stops the execution of the main MovieClip(), but, it does not cancels any event handlers. This is important since often you may put game code in an event handler, get to the end of the game, or so you think, and the event handling code still goes on! This can be fixed by deleting (i.e. canceling) the event handler when you stop the animation. Check out:

**a_goto_6.fla**

Here, frame 4 now contains:

```javascript
    trace("inside frame 4")
    delete _root.onEnterFrame
    stop()
```

If you load and run it, you see the onEnterFrame handler is never called, because it is canceled.

It may seem like we just took a long digression from games, but it was an important one. Most games have different types of screens. For example a start screen, a game screen, and then an end screen. Perhaps different screen for different levels. The way to do this in flash is with gotoAndPlay( ). Also, when a game is over, you need to be able to stop the game.
**Notes 8: Buttons**

Actionscript “buttons” provide another from of user interactivity. Mostly buttons are used for start/stop/modifying animation flow or to enable text input. Lets start with how buttons are used for starting animations.

Buttons are another built in Actionscript class, hence check out the class definition in the help manual actionscript dictionary. We will mostly be using the event handler Button.onPress().

Lets create our first button! Follow these steps:

- open a new flash document, save it as some name
- Go to: Window -> Other Panels -> Common Libraries -> Buttons
- Open the “arcade buttons”, and drag one of them onto the stage
- Go ahead and publish (F12), and click. It clicks! But it does not do anything else
- Back on the flash stage, select the butt on, then give the instance a name in the Properties panel. Say “btn_first”. You can now use “btn_first” as an identifier for this button inside of Actionscript code.
- Create a new layer for actionscript, and in the first frame add:

  ```javascript
  btn_first.onPress = function() {
    trace("ouch, you pressed me!");
  }
  ```

- Run the code by ctrl-enter so you can see the trace, move the output window to the side so you can keep pressing the button and see the output

If you want, you can download a file I created as above:

**a_button_1.fla**

Buttons have other useful event handlers such as onRelease and onRollOver. Check out the following code in:

**a_button_2.fla**

```javascript
btn_red1.onPress = function() {trace("pushed btn_red1");};
btn_red1.onRelease = function() {trace("released btn_red1");};
btn_red1.onRollOver = function() {trace("rolledOver btn_red1");};

btn_red2.onPress = function() {trace("pushed btn_red2");};
btn_red2.onRelease = function() {trace("released btn_red2");};
btn_red2.onRollOver = function() {trace("rolled over btn_red2");};
```
Here we have two buttons. Load the file and run it with ctrl-enter to see the trace command outputs. Again, you probably will need to move the output window to the side to keep seeing the output and the buttons at the same time.

Instead of just printing out messages, you can use the button event handlers to do interesting things. Check out and run:

**a_button_3.fla:**

```actionscript
function init() {

    // initializes call back functions for buttons and anything else
    if (this.initied != undefined) {
        trace("ERROR, calling init more than once, exiting....");
    }

    this.initied = true;  // set initied == true to indicate have run this

    btn_left.onPress = function() { mc_penguin._x -= 10 ;} ;
    btn_right.onPress = function() { mc_penguin._x += 10 ; } ;
    btn_DecSize.onPress = function() {
        mc_penguin._xscale -= 10 ;
        mc_penguin._yscale -= 10 ;
    } ;
    btn_IncSize.onPress = function() {
        mc_penguin._xscale += 10 ;
        mc_penguin._yscale += 10 ;
    } ;
}

init() ;
stop() ;
```

Before trying to understand the code, run the animation first. Now lets talk about the layers in the flash file. The top layer contains the penguin image saved as a MovieClip and names mc_penguin (check out the properties panel for the name). The next layer contains the text found on the buttons. The next layer contains the four buttons. If you select each one separately and look at the property panel you will see their names. Finally, the bottom layer contains the actionscript code above. The first code chunk defines a function called “init()”. Then, the function is called and the program stopped. The function sets up the onPress() event handlers for each of the four buttons, thus the stop() does not stop the handling of button presses. The only way to cancel would be to delete the handlers (or create new empty handlers, but that is just odd!). The use of this.initied is a normal way to make sure code is executed only once.
So, what else could we use buttons for? How about controlling the execution of frames within a program to give us a game start, play, and end screen? Check out the next section of the notes.

**Notes 9: BallChase_1: Keys, buttons, and game phases**

Now let's put together what we learned in sections 6 – 8. Games, as said before, often have phases: A stat phase, where rules are presented or some clever animation is running, a play phase, and then an end phase where the game outcome, such as a score, is displayed. Often the play phase is a series of phases corresponding to levels with screen between each level. By using buttons and gotoAndPlay( ) combined with different frames we can build these game phases.

Using buttons to start a game has the additional advantage of setting the focus to the game so the keyboard is activated.

Check out the following fla. Load it, play it (F12), and look at the code/structure:

*a_ballChase_1.fla*

A few notes about this code:

- There are three layers: AS (actionscript), background (below the other layer), and graphic/buttons/text.

- The keyframes in the AS layer are labeled: startPhase, playPhase, endPhase. The gotoAndPlay( ) calls use these names instead of frame numbers. This way if you add/move frames the code still works. This is the preferred way of referencing frames.

- AS code in “startPhase” frame: This code defines the onPress() handler for the start button and then has the command “stop( )”. It is essential you add this stop command, without it the game would just define the onPress( ) handler and then advance to the next frames. This way the game remains in this frame until the player presses the start button.

- Again, pressing the start button has the added benefit of setting focus so the keyboard keys work in the playPhase.

- “playPhase” code: Again see the inclusion of the “stop( )” command. This prevents the animation from leaving this frame until the player hits the other ball.
The `onEnterFrame()` function checks for this condition, and when true calls `gotoAndPlay("endPhase")`.

- “playPhase” code: The initial location and the velocities of the computer’s ball are set somewhat randomly to make the game more interesting
- In the “endPhase” screen the time to complete would be displayed. To do that we need to learn about how to time actions in AS and how to do dynamic text output.

All we need now to make this a more complete game are:

- A way to time how long it take the player to capture the computer’s ball
- A way to display that time
- Possibly add some sounds (background music, maybe a sound at capture time?)

We will deal with the first two of these next, sounds later.

**Notes 10: Text input/output**

Can you believe we have gone two weeks now and you don’t know how to do input and output yet? I guess Barney getting rained on is just so exciting no one even thought to mention that we don’t know how to do user I/O.

I/O is done in action script through text boxes created by the text tool (the button with the big letter “A” in the tool panel). With the text tool we can create three types of text boxes:

- Static: just used to put words on the screen, there is no way to modify them from AS.
- Input: used to get input from the user. The text box is associated with an AS variable via the properties panel.
- Dynamic: used to write text to the screen from action script. Again the box is associated with an AS variable via the properties panel.

We won’t cover how to use the static choice, it is pretty straight forward, just select it, draw a box on the screen, and type in the text you want. You modify the text properties, like font and size, via the Properties panel.

Let’s start by creating a program that prints out the frame number every frame:
• Create a new document

• Click on the text tool

• In the properties panel select “dynamic text” (top left of the panel)

• Modify the font (size and name) if you wish

• On the stage click and hold down and drag out a box for the text

• IMPORTANT: now associate an AS variable with the box. In the Var field of the properties panel type in “outbox”

• Create a new layer for the AS code

• In frame one add the following AS code:

```actionscript
var frameNum:Number = 0 ;
onEnterFrame = function() {
    frameNum++ ;
    outbox = "frame " + frameNum ;
}
```

• Publish and run the code

Note, if you had problems, this example can be found in:

**a_io_1.fla**

To do input we need to use the input text option for the text box, and again assign a variable to the box via the properties panel. Look at the following program:

**a_io_2.fla**

Inside of flash, click on each of the three boxes and look at the properties panel for each. The words are static text. The box to the right is input text, and the bottom box is dynamic text. Also notice that the variable field for both the input box and the dynamic box is “inString”. Now publish/run the program. Click on the input box (next to the word input) and type something in. As you type, each character it is echoed in the box below. This is because every change made in the input box automatically changes the content of the action script variable name “inString”. Note, you never even declared this variable in AS, that is okay, it is done automatically. The dynamic text box is also tied to the variable “inString”, thus, any change to variable “inString” is automatically sent to the dynamic text box associated with it. Granted, this example is sort of weird, but it
shows how the variable associated with input boxes is automatically updated as the user
types into the box, and how the dynamic text box associated with a variable is
automatically updated as the variable changes.

Now lets do something a bit more normal: ask the user for their name, and output a
welcome message. The following is contained in:

a_io_3.fla

    btn_enter.onPress = function() {
        outString = "Welcome " + inString ;
    }

Look at the .fla file. You will see there are three text boxes (one each of static, input,
dynamic) and a button. When the button is pressed the value of variable outString, which
is bound to the dynamic text box (you can see this by looking at the properties panel), is
assigned the string “welcome” concatenated with the string entered by the user. The idea
is the user fills in the input box, pushes the button, and gets the desired welcome
message.

Notes 11: Timing Events and Using the Date Class

Games often require timing of events. We may have an obstacle course and players try
to get their best time. One way to get times is by using the ActionScript Date class. As
usual, check it out in the help manual dictionary. If you get the current time at the
beginning and end of the time interval of the event you wish to time, you can then just
take the difference to get the elapsed time. The following code shows this. Load it and
run it with ctrl-enter so you can see the trace command output.

a_timer_1.fla

    var theDateObj:Date = new(Date) ;
    var startTime:Number = theDateObj.getTime() ;

    var temp:Number ;
    // do a bunch of computations to fill up some time
    for (var counter:Number = 0 ; counter < 1000000 ; counter++) {
        temp = 2 * 3 * 6 / 8 + 3 ;
    }

    var dateObj2:Date = new(Date) ;
    var endTime:Number = dateObj2.getTime() ;
    var difference:Number = endTime - startTime ;

    // convert difference to seconds
difference /= 1000 ;

trace("startTime = " + startTime) ;
trace("endTime = " + endTime) ;
trace("Elapsed time was: " + difference + " seconds") ;

The data class can be used for timing as above, but it is also useful to get the current day and time. Let's look at code to print out the current date and time. All that is used in this code is the Date class and two text boxes (dynamic and static). Note, you could modify this code to keep a running time going by writing an appropriate onEnterFrame event handler.

**a_timer_2.fla**

```javascript
var theDateObj:Date = new(Date) ;


outString = months[ theDateObj.getMonth() - 1 ] + " " ;
outString += theDateObj.getDate() + " at " ;
outString += theDateObj.getHours() + ":" ;
var min:Number = theDateObj.getMinutes() ;
if (min < 10)
  outString += "0" + min ;
else
  outString += min ;
```

The Date.getMonth( ) command returns a number 1 to 12, hence the use of the array to convert to a string. Likewise, Date.getMinutes( ) returns a number 0 to 60, and hence for 0 to 9 needs to be converted to a string and prepended with a “0”.

**Notes 12: Putting it all together: Ball Chase 2!**

Okay, we are ready to put together the keyboard, buttons, different frames for different game phases, dynamic text output, and timers. Check out the following .fla file. You should understand everything in the code, read through it and ask questions if you don’t.

**a_ballChase_2.fla**