Welcome to the second article on Actionscript – aka Flash Programming.

If you haven't already done so, I recommend you to read the first part before going further. The first part was intended for the absolute beginner in Actionscript and laid the foundation for what's to come.

Starting with this part, I'm going to discuss different aspects of Actionscripting. I don't intend to make this a tutorial, but rather a material that you can read, learn from and improve your skills. Here we'll be dealing with Events. We'll start with simple, button events and move up to more complex interactions, suitable for games.

Note that these pages alone won't make you an expert in Actionscript. You still need to work hard, read and experiment a lot. All I'm doing is to guide you.

I also need your feedback. I'm not a professional writer. English is not my native language. If you have comments, suggestions, need to ask further questions or want to correct something, email me at armandn@media-division.com

Flash programming:
Writing better Actionscript

2. Event-driven programming

First step – linear programming

To learn events and how to handle them, you first need to understand the "old" way of programming. Have you ever written a batch script or a small program in Pascal or basic?

Suppose you want to make a small program that adds two numbers. How would you do it? Well, let's see step by step:

1. Ask for first number;
2. Ask for second number;
3. Make the result equal to first number plus second number;
4. Present the result.

We can rewrite the lines above in *pseudocode* (or something close to it anyway)…

```
1. Start;
2. Input firstNumber;
3. Input secondNumber;
4. result = firstNumber + secondNumber;
5. Write result;
6. End.
```

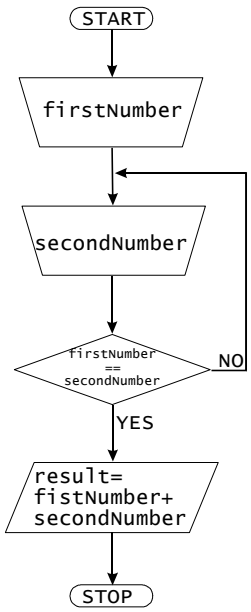
This is a linear program, it has a start, an end, and some clearly defined instructions in the middle, which are executed in a strict order. Of course, we can complicate matters a bit, like adding loops and conditions.

Now let's say we want a program to *divide* two numbers instead of adding them. Just take the previous code, replace "+" with "/"in line 4 and we're done. Right? Err… Wrong. This is the first and most common mistake in the world of programming.

Always, and I do mean always, check the data that's being entered. Never assume that the user or some other program is providing valid data. What happens if the second number that's entered is 0? We get a division by zero. The computer can't handle it. Therefore, what we must do is force the user to enter valid data.

```
1. Start;
2. Input firstNumber;
3. Input secondNumber;
4. if secondNumber == 0 then go to line 3;
5. result = firstNumber + secondNumber;
6. Write result;
7. End.
```

When writing a program, it's always a very good idea to make a sketch on a piece of paper about its flow. There are some standard elements for these diagrams, but I'd be happy if you'd just take the time to make a rough sketch, even if you "invent" your own symbols.



Have a look at the diagram on the left. It shows the flow of program above. Of course, as your programs will get increasingly complex, the diagrams will look a lot more complicated too... However, most of the time they're worth the trouble, as you'll have a better view of the overall structure and maybe you'll find ways to optimize the way it works.

OK, so far so good, but what does it have to do with Flash? We're not programming using diagrams; we don't even have such actions like "Input()"! True. Read on.

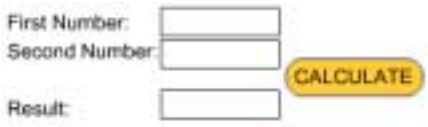
By looking at the previous examples, you can see the shortcomings of the method: it's very rigid. The program asks you for the first number, then the second, then it displays the result. You can't change the order. If you want to change a number, you have to start all over again.

For this to work, we need an entirely new approach.

Events

We can create several objects and let them interact with each other and the environment. In order to interact with the environment (e.g. the user) the objects may have event handlers attached to them.

In Flash, most of these events are related to the mouse pointer: clicks, drags and so on. If you have spent some time with Flash, you probably already know about how you can work with buttons. As a *warm-up*, we can rethink our division program from the previous section to work with flash. For this we need two text fields to enter our numbers, a thirst field to display the result, and a button to calculate.



Start with a blank movie. Create two text fields. In the Text Options panel, set both of them as Input Text and in the variable field enter `firstNumber` for the first field and `secondNumber` for the second field. Create a third text field, set it as Dynamic Text and set its variable name as `result`. For all three fields select the Border/Bg checkbox, so they have a visible boundary when you export the movie. Also, add some simple text in front of these fields, to explain what each one means. Now create a simple button and add some actions to it – the code from the left. Export the movie and play it by entering different numbers and clicking on Calculate button.

```
on (release) {
        if (secondNumber == 0) {
                result = "Division by zero";
        } else {
                result = firstNumber / secondNumber;
        }
}
```

How does it work? You can consider that the three text fields and the buttons are objects that can interact with each other. The button object is the one that performs some actions based on an event. These events have to be attached to the button, not placed in the frames of the button.

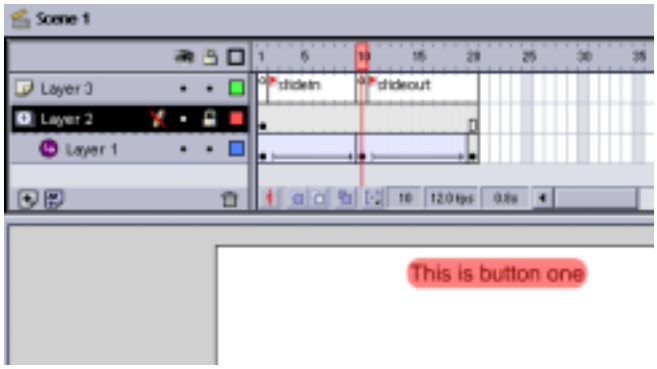
The `on(release)` event takes place whenever the user releases the previously pressed mouse pointer over a button. There's also an `on(press)` event that we could have used, but triggering the event on release is preferred from an usability perspective as it allows the user to change his or her mind after they pressed the mouse button, all

they have to do is release the mouse pointer outside the button... Try this in any GUI (Windows, MacOS, KDE...).

Simple button events

We move on to what's probably the most used effect on the web and especially in Flash: the mouseover.

Let's say you have an interface with three buttons and want that every time you mouseover (rollover) an element, an explanatory text to be displayed. Start by making three buttons and label them "1", "2" and "3".



Now, in the library, create three movie clips with an animated text, like the picture to the left. In each movie clip, make the text move from left to right and the again left, maybe also using a Ease In/Ease Out effect, so in the first frame the object is at the left, on frame 10 it is to the right and at the frame 20 it is also at the left. Add a second layer and draw a rectangle over the position of the text as it is in frame 10. Make this second layer a mask and play the animation. The text should slide in and out if the visible area. Add a third layer. In frames 1 and 10 place a single `stop();` action. In frames 2 and 11 add some labels, like "slidein" and "slideout". Make two other clips with different text and then bring all three to the stage. Place each clip under the corresponding button. Note that while you may drag each clip from library to the stage, most likely the clip will be. Now give an instance name to each clip, like "rollover1", "rollover2" and rollover3". Open the Actions window and attach the following code to the first button:

```
on (rollOver) {
    rollover1.gotoAndPlay("slidein");          /* will execute when mouse pointer rolls */
}                                              /* over button */

on (rollOut) {                                 /* will execute when mouse pointer rolls out */
        rollover1.gotoAndPlay("slideout");
}

on (release) {                                 /* executes when user clicks on the button */
    getURL("page1.htm");
}
```

Do the same for the other two buttons; just replace "rollover1" with "rollover2" and "rollover3" as necessary. Play the movie. As you roll over each button, the text slides in below it; when you roll out, the text slides out.

Let's see how it works. When you start the movie, each of the three movie clips encounters the `stop()` action on frame 1. They stop, and the text is invisible. When you roll over button 1, the `rollOver` event takes place. It tells the movie clip with the instance name of "rollover1" to go to the frame titled "slidein" and play from there. "Slidein" is the second frame, so we see the text gradually appearing. When the animation reaches frame 10, it encounters another `stop()` action – and it stops. When the user moves the mouse pointer away from the button, the `rollOut` event comes up

```
on (rollOver) {
    rollover1.gotoAndPlay("slidein");          /* the text for button 1 comes in… */
    rollover2.gotoAndPlay("slideout");         /* ... while the others go out */
    rollover3.gotoAndPlay("slideout");
}

on (release) {                                 /* executes when user clicks on the button */
    getURL("page1.htm");
}
```

and so the “rollover1” clip will go to frame 11 and play.

Next, let’s do a variation of the above. What if we want the animated text to stay visible until we rollover another button? Then obviously we don’t need the on(rollout) event anymore. Picture this: the user first rolls over Button 1. The text comes in. The user rolls over Button 2. The text below Button 1 slides out while the text for Button 2 slides in. Let’s write the event for Button 1:

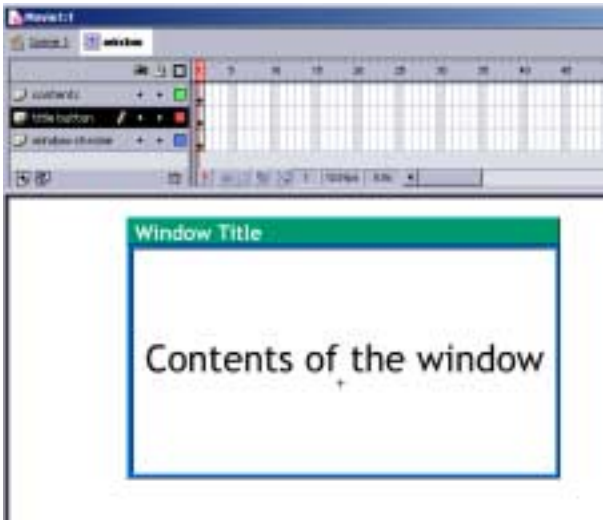
You should add the script to the other buttons too; just make sure that you replace the clip names (for button 2, swap “rollover1” with “rollover2” and so on).

What if the user rolls over and out the same button (let’s say button 1) very quickly? The slide-in animation for the clip rollover1 would go halfway and then start over, while the text for the other buttons, already hidden, would suddenly appear (straight at frame 11 from frame1) and slide out. This doesn’t look quite well. What we need to is this: make sure an animation is started only if it isn’t already taking place and that a text doesn’t slide out if it wasn’t visible in the first place.

```
on (rollOver) {

    if (rollover1._currentframe == 1) {
        rollover1.gotoAndPlay(“slidein”);
    }

    if (rollover2._currentframe == 10) {
        rollover2.gotoAndPlay(“slideout”);
    }

    if (rollover3._currentframe ==10) {
        rollover3.gotoAndPlay(“slideout”);
    }
}

on (release) {
    getURL(“page1.htm”);
}
```

`_currentframe` is a property of movie clips. As its name implies, it returns the current frame of the movie clip. With it, we can check where the animation is. `if (rollover1._currentframe == 1)` tests whether or not the “rollover1” clip is not at frame 1. Only if it is so the animation is started, otherwise we conclude that the animation is already taking place (or it has ended and the clip has stopped at frame 11). The same goes with the other two clips. They’ll slide out only if they are visible, stopped at frame 10.

Drag’n’drop



Another nice thing you can do in Flash is to make an interface with drag’n’drop capabilities. Think about a window that can be dragged around the screen, or a file that can be sent to Trash Can / Recycle Bin by dragging it.

Start by making a new movie clip, something like the screenshot on the left. Make the area at the top of the window (green on the screenshot with text Window Title) a button. We will use it to drag the window.

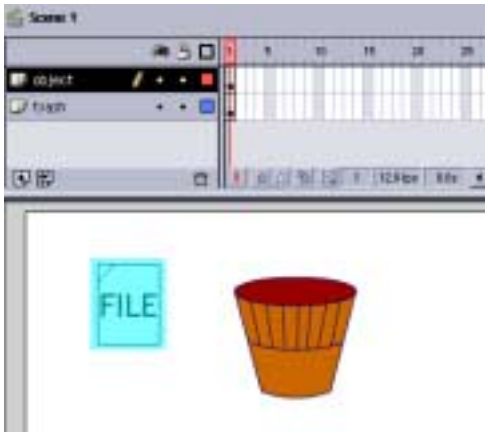
We must now add some actions to our button, so that when we press the mouse button over the title area, the window is dragged, and when the mouse button is released, the window is dropped.

As you’ve probably already guessed, we use `on(press)` and `on(release)` events. All we have to do is to attach a few lines of code to our window title bar button:

```
on(press) {                                    /* when mouse button is pressed, we start */
    this.startDrag(false,20,20,600,400);       /* dragging window between these boundaries*/
}

on(release) {
    this.stopDrag();                           /* stop the dragging action */
}
```

Piece of cake!

For more about the syntax of StartDrag, see the Actionscript Dictionary that comes with Flash or my Quick Flash Actionscript Reference.



Next, let's do something just a bit more complex. An object can be dragged and dropped on another one, leading to a certain action. This form of interactivity can be used in games, interfaces, simulations and so on – the possibilities are endless.

There is more than one way to achieve this. As you become more familiar with Actionscript, you'll realize that you can use different solutions to achieve the same result... The trick is in finding the optimal solution (fast, easy and reliable.)

Start with a new document and create two movie clips (objects). The graphics of each of them is unimportant; maybe make them look like in the screenshot above. Make sure you give an instance name to each one, such as "fileObject" and "trashCan". It's also a good idea to place them in different layers, so that fileObject is on top.

Inside the fileObject movie clip, place an invisible button (this is a button that only has the Hit frame defined, the others [Up/ Over/ Down] are empty). This button is shown by Flash as semi-transparent cyan, as in the screenshot. Once you export the movie, it will be invisible, but would still work when you roll over it.

```
on(press) {
    this.startDrag(false,20,20,600,400);
}

on(release) {
    this.stopDrag();
    if (this._droptarget == "/trashCan") {
        trace ("Help, I'm going to trash!");
    }
}
```

The code that you must attach to the invisible button is very similar to the previous one, but now we also check to see if the clip was dropped over the trashCan. For this, we have a simple property of movie clips called `_droptarget`. It returns the name of the clip that is directly underneath the mouse cursor currently dragging. This means that if you have several movie clips in different layers but at the same position on the screen, the topmost movie clip will be considered. It is also worth noting that the path for the movie clip is returned – not sure why – in the "/"-syntax, not "."-syntax (i.e. "/trashCan" not "_root.trashCan".

How do you find this entire Event handling so far? Not hard, I hope, since we're going to do even some more advanced stuff in the next part.

Clip Events

So far we've discussed only events linked to buttons, but guess what, movie clips can have events too and this means that we can do many neat things.

Do you remember the script at the end of the first chapter of these articles? At that time, we were discussing a very simple way to animate an object using Actionscript. Here's the code again:

```
[Frame 1]
        startX = 50;                    /* ball's start location */
        endX = 400;                     /* ball's end location */
        speed = 10;                     /* ball's speed */
        currentX = startX;              /* initialize the position variable */
        BallInstance._y = 200;          /* positions the ball on Y-axis */

[Frame 2]
        ballInstance._x = currentX;     /* positions the ball on X-axis */
        currentX = currentX + speed;    /* calculates the new postion */

[Frame 3]
        if (currentX < endX) {          /* is the position lower than the end? */
                gotoAndPlay(2);         /* yes, loop back */
        } else {                        /* no... */
                stop();                 /* ... stop */
        }
```

We were using a script in three frames that controlled a movie clip called BallInstance. What if we want the object to be "smart" enough to take care of itself? We could rewrite the code inside the clip and replace `BallInstance._x` with `this._x` and `BallInstance._y` with `this._y`, however there a more elegant approach.

Look at the code. The actions in the first frame are executed only once, when the clip is started and then it loops in frames 2 and 3 until it reaches a certain position on X.

In Flash, we have several clip events that can be handled with `onClipEvent()` attached to a movie clip, just as the `on()` events are attached to a button. Two of these events are `load` and `enterFrame`. The first event occurs when the clip is first loaded. The second event occurs whenever the clip advances to the next frame – actually, it is linked to the movie frame rate because the event occurs even if the movie clip is stopped.

Let's use these events to recreate the animation script. Start with a new movie and make a new movie clip with just one frame and some dummy graphics. Place it on stage and give it an instance name. Open the Actions window and add the following script to the clip:

```
onclipEvent(load) {
    startX = 50;                        /* ball's start location */
    startY = 200;
    endX = 400;                         /* ball's end location */
    speed = 10;                         /* ball's speed */
    _x = startX;                        /* initialize the position variable */
    _y = startY;                        /* positions the ball on Y-axis */
}

onClipEvent(enterFrame) {
        _x += speed;

        if (_x >= endX) {               /* is the position past the end? */
                stop();                 /* yes, stop the clip */
        }
}
```

I couldn't resist optimizing the clip some more... As you can see, what was written before in three frames, now takes only one frame and does not reside anymore in the main timeline, but on our object. This increases flexibility. A few additional notes: "`this`" is not required in this script so writing simply "_x" is just as good as "`this._x`". Secondly, there was the variable called currentX that isn't used in the new script. Sometimes you need to use an extra variable to perform some changes on and only in the end to update the movie clip – and sometimes you don't. It depends on how elegant and flexible you want your code to be. If you want something quick, you can reduce the size of the script even further by eliminating all the variables, like in the script below.

```
onclipEvent(load) {
    _x = 50;
    _y = 200;
}

onClipEvent(enterFrame) {
    _x += 10;

    if (_x >= 400) {
        stop();
    }
}
```

I don't recommend this style of programming, because it is hard to update. Sure, we have 7 lines of code now, but what if we have 700 and need to change a value that occurs a hundred times in many frames and movie clips? Always make your programs as flexible as possible and you'll avoid headaches and sleepless nights.

Let's move on.

Buttons without buttons

Using movie clips, we can emulate the functionality of a button but use a movie clip instead. Now, why would we want to do this? After all buttons are easy to create and work just fine. Well, sometimes you need to add some extra functionality that is hard to do with a button or simply impossible. Let's say you have a fast forward button used to advance a movie clip, or maybe a button that moves an object on the screen. With a regular button, you need to click each time you want the event to take place. However, what if you want a smooth, continuous action so that the event takes place as long as you keep your mouse left button down?

Make a new movie clip with three keyframes. In each frame, draw a rectangle (or some other shape) with a different color. Each frame will act as a state of the button, Up, Over and Down. The shape of the movie clip will determine the Hit area. Add a `stop()` action to each keyframe. Place the movie clip on stage, give it an instance name and attach the code below to it.

```
// ----- test if the mouse pointer is over the button -----
onClipEvent(mouseMove) {
        if (this.hitTest( _root._xmouse, _root._ymouse,true) == true) {
                isOver = true;
        } else {
                isOver = false;
        }
}

// ----- test if the mouse pointer is down -----
onClipEvent(mouseDown) {
        isDown = true;
}

// ----- test if the mouse pointer is up -----
onClipEvent(mouseUp) {
        isDown = false;
}

// ----- set the state of the movie clip (pseudo-button) -----
onClipEvent(enterFrame) {
        if (isOver == true) {
                if (isDown == true) {
                        gotoAndStop(3);
                        /* add any actions here */
                } else {
                        gotoAndStop(2);
                }
        } else {
                gotoAndStop(1);
        }
}
```

Here there are four events. The first one, `mouseMove`, takes place every time the mouse pointer changes its position. When this happens, we test for collision between the shape of the movie clip and the mouse pointer. If the test returns `true`, we set the `isOver` variable accordingly. We know now that the mouse pointer is over the movie clip.

Next, we need to know when the mouse button is pressed and when it is released. For this we use two events, `mouseDown` and `mouseUp` to set another variable, `isDown` to true or false.

The fourth event is `enterFrame`. Here we have two nested conditions, which can be tricky to follow and that's why I color-coded the matching start-end brackets. We test to see if the mouse pointer is over the movie clip. If so, we test if the mouse button is also down. If both conditions are satisfied, we know that the movie clip was clicked, so we set the frame (state) of the clip and we can proceed with whatever actions we want to do. If the mouse pointer is over the movie clip but the mouse button isn't down, we just set the state of the clip to over (frame 2). Finally, if the mouse pointer isn't over the button, we don't care about the rest and just set the frame of the clip to 1.

Custom cursors

If we can do custom buttons, why not make a custom cursor too? In Flash 5, you can hide the actual mouse pointer and replace it with a movie clip of your own. It's a very powerful feature but unfortunately, I haven't seen it used effectively.

```
onClipEvent (load) {
    Mouse.hide();
}

onClipEvent (unload) {
    Mouse.show();
}

onClipEvent (mouseMove) {
    _x = _root._xmouse;
    _y = _root._ymouse;
    updateAfterEvent();
}

onClipEvent (mouseDown) {
    gotoAndStop (2);
    updateAfterEvent();
}

onClipEvent (mouseUp) {
    gotoAndStop (1);
    updateAfterEvent();
}
```

The idea is very similar to the previous button technique. Make a new movie clip with two keyframes (say an arrow for first and a hand for the second), add a `stop()` action to each keyframe, place the clip on the stage and add the following code to it:

The `load` and `unload` events make the real mouse pointer disappear and reappear when the custom cursor is placed on/ removed from screen. The `mouseMove` event makes the cursor to follow the mouse pointer position around the screen. `mouseDown` and `mouseUp` events set the cursor to frame 2 when clicked, and then back to frame 1.

The `updateAfterEvent()` action forces a refresh of the screen whenever the events take place and leads to a smoother motion.

Of course, you can link the custom button with the custom pointer so that it automatically changes to frame 2 (the hand) when you rollover the movie clip. This results in a perfect emulation of the real button-pointer relationship in Flash. It's easy to do, so I leave to you the implementation.

What about key events?

So far we've discussed mainly mouse events. However, you can use keypresses to achieve a similar result.

Assigning a key event to a button is dead simple. If you want an event to occur either when you click the mouse or when you press, let's say the SPACE bar, just write:

```
on (release, keyPress "<SPACE>") {
        //your actions
}
```

This technique works fine for simple actions, but has one big drawback: you can't get continuous actions. If you use buttons to move a movie clip on the screen with a script like

```
on (release, keyPress "<Right>") { root.myClip._x += 10; }
```

Then you need to tap on the right arrow key every time you want the movie clip to move, you can't keep the key pressed and expect the clip to keep moving. Also, you can't use key combinations, such as keeping both right and up arrow keys pressed to move your clip in a NE direction.

To address these limitations, we'll use clip events.

Start a new movie and create a new movie clip. As graphics you can do anything you like or just draw a circle if you're lazy. Because we're ambitious, we don't just want our ship to move, we want it to have some feel of inertia. We also want to write as little code as possible, so just let's attach the following code to the clip:

```
// event occurs when the clip is loaded - we just initialize some variables -------------
onClipEvent (load) {
    thrust = 5;                                 /* speed increase per key pressed */
    xspeed = 0;                                 /* initial horizontal speed */
    yspeed = 0;                                 /* initial vertical speed */
}

// this event occurs periodically - we recompute the current position -------------------
onClipEvent (enterFrame) {
    if ((xspeed !=0) || (yspeed !=0)) {         /* if the clip is moving... */
        xnew = _x + xspeed;                     /* ... we calculate the new theoretical */
        ynew = _y + yspeed;                     /* position on X and Y... */

        if ((xnew < 500) && (xnew > 50))        /* ... but only if the new coordinates */
            _x = xnew;                          /* are not outside the boundary */
        if ((ynew < 400) && (ynew > 40))
            _y = ynew;

        xabs = Math.abs (xspeed);               /* now we break down each speed in two: */
        yabs = Math.abs (yspeed);               /* the absolute value and the sign (+/-) */
        xsign = xspeed / xabs;
        ysign = yspeed / yabs;
        if (xabs > 0) {                         /* here we decrease the speed in its absolute */
            xabs -=1;                           /* value and we recompose the speed. */
            xspeed = xsign * xabs;              /* E.g.:  xspeed = -5 */
        }                                       /*      => xabs = 5; xsign = -1 */
        if (yabs > 0) {                         /*         xabs -=1; => xabs = 4 */
            yabs -=1;                           /*      => xspeed = 4 * (-1) = -4 */
            yspeed = ysign * yabs;
        }
    }
}

// this event occurs if at least one key is currently down, we then find one which one(s)...
onClipEvent (keyDown) {
        if (Key.isDown(Key.RIGHT)) {            /* we know that at least one key is down */
                xspeed +=thrust;                /* and we adjust the horizontal/vertical */
        }                                       /* speed depending on each key */

        if (Key.isDown(Key.LEFT)) {
                xspeed -=thrust;
        }

        if (Key.isDown(Key.UP)) {
                yspeed -=thrust;
        }

        if (Key.isDown(Key.DOWN)) {
                yspeed +=thrust;
        }
        updateAfterEvent();
}
```

Don't feel intimidated if this code seems to complicated to digest. The comments on the side should prove helpful to you. We have two variables, `xspeed` and `yspeed` and a constant, named `thrust`, which will increase/decrease xspeed/yspeed by a certain value. These are initialized when the clip is loaded. When an arrow key is pressed, the horizontal/vertical speed is constantly increased/decreased.

So, if you keep the right arrow key pressed, the xspeed will increase from 0 to 5, 10, 15 and so on. If the up arrow key is also down, the yspeed will decrease from 0 to –5, -10, -15... resulting in an accelerated motion. Of couse, if you keep both keys pressed, the clip will move in NE diagonal.

The enterFrame event takes care of the actual repositioning of the clip if it is inside the bondary. Also here we have a slowdown effect. The speeds are decreased (in their absolute values, by 1) but this effect is visible only when you release the key. Then the clip is no longer accelerated and the xspeed drops from 15 to 14... until it stops.

To be continued...

That's all. By now, you should have understood how to handle almost any kind of event, for both buttons and movie clips. In the next chapter, I plan to talk about objects in detail and how to make them interact. For this to happen however, I will need your feedback, so please drop me a line at armandn@media-division.com and let me know if you've found this tutorial useful.

Armand Niculescu

Art Director

Digital Vision Multimedia

http://www.media-division.com