Animation for Flash Games (part 2): Sprite Manipulation for the Scripter
By Patrick Fram

This article—the second in the series—shows Adobe Flash* game scripters how to manipulate animated sprites for use in Flash* games. The Flash* player was recently updated to make it more efficient, and therefore more assessable to mobile devices and smaller computers like netbooks. Given the growing popularity of these devices, Flash* game development has become an increasingly valuable skill.

The previous article discussed how artists could create a game sprite in a way that would make the scripter’s job easier. This article uses that same game sprite to create a simple platform game engine.

Getting Started

![Figure 1. The robot sprite movie clip created in part 1](image)
To recap a bit, you created a game sprite of a robot character in the last article (see Figure 1). It was a single movie clip, with all of its animations existing in one timeline (see Figure 2). Each animation section was labeled in the timeline, so not only was it easy to visually tell what you were looking at, but the script would be able to use these labels to control the animation.

![Figure 2. The labeled timeline inside the game sprite movie clip](image)

To get started on your next task, create a new Adobe ActionScript® 3.0 file and add the robot movie clip from part 1 to the stage. Give it an instance name so that your code can refer to it; to do so, open the Properties panel, and name the instance `robot`.

### Scripting the Sprite’s MovieClip for Animation Control

Now, the artist built this sprite in an organized way, but it is not quite ready yet. You have to add a bit of ActionScript® code to it to get it to work properly.

The simplest thing you need to do is add code to control how animations play. Although you haven’t written it yet, the system that will control which animation is playing will look something like this:

```javascript
gotoAndPlay("still");
```

Now, if that line were used right now, the sprite would start playing at the `still` label, and then keep going right through and into the run animation. So, add a single line of code to the `still` frame:

```javascript
stop();
```
Now, when that frame is played, the sprite will freeze on it until told otherwise.

Next, think about the run animation. Unlike the static still animation, this one is a loop that’s expected to cycle through until told otherwise. So, add a single line of code at the end of the animation:

```javascript
gotoAndPlay("run");
```

Now, when this animation gets to the end, it will reset itself to the first run frame and keep looping until you tell it to do something else (see Figure 3).

![Figure 3. Adding code to the sprite’s timeline](image)

Notice the extra frame added to the animation at the end of the run section: It’s there, because Adobe Flash Player will interpret the code of that frame before updating the display, meaning that this frame will `gotoAndPlay("run")` immediately and will never be seen. So, by adding the extra frame, the animation’s timing stays consistent.

### Scripting the Sprite’s Movie Clip for Game Physics

Even though you haven’t programmed the game physics yet, now is a good time to set up all the code you’re going to need in the sprite. In this sprite, you are going to add a setup frame to the very beginning in which you declare a few variables. You declare two types of variables here: **spec constants** and **tracking variables**.

**Spec constants** are values on which your physics engine bases its calculations. Different game objects might have different specs, so it’s a good habit to make these constants part of the sprite;
that way, the physics engine can run the same script for all game objects and see different results. So, in the first frame, add the following script:

```ActionScript
//spec consts
const acceleration:Number = 0.7;
const maxSpeed:Number = 8;
const jumpForce:Number = 12;
```

These elements should be fairly self-explanatory. The engine uses the acceleration constant to determine the rate at which the sprite speeds up and slows down. The max speed constant caps the sprite's speed so that it doesn't go too fast. The jump force constant is the amount of force you apply in a vertical direction when the player jumps. Changing these values alters the behavior of the sprite without needing to touch the physics engine. If your game requires that these constants change during gameplay (for example, you want the acceleration to be reduced when the character is standing on ice), you can declare them as variables instead; constants cannot be changed during runtime.

Now, add a few tracking variables. These variables update constantly during gameplay:

```ActionScript
//tracking vars
var velX:Number = 0;
var velY:Number = 0;
var anim:String = "still";
var grounded:Boolean;
```

The velX and velY variables are the sprite's velocities in the X and Y directions. The anim variable is a string that will hold the name of the current-playing animation; start it off as still. The grounded variable is a Boolean variable that simply tracks whether the sprite is touching the ground.

Next, add one more line on that first frame:

```ActionScript
gotoAndPlay(anim);
```

Because you set anim to still, this setting will advance the sprites animation to the still section. You want the sprite to start the game like that.

You also want to add a jump animation (see Figure 4). Technically, the artist should have done this, so, as a scripter, you're just going to make a simple jump animation as a placeholder. Later, you can hand it back to the artist and get a good one made. The animation you made is just a single frame, like the still animation; that's all you need to work with. (Also, put a stop() on the jump frame, just like you did with still, because it is a static animation as well.)
Building a User Input Script

Now that your sprite is set up, you can start to build the main script that will control it. The first step is to create a way to read the user's input. Back on the main timeline, add the following code to the first frame (your main timeline should only have one frame):

```ActionScript
//add listeners for key presses
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyUp);
var arrowLeft:String = "up";
var arrowRight:String = "up";
var arrowUp:String = "up";

//handle key presses
function keyDown(e:KeyboardEvent):void{
    setState(e.keyCode, "down");
}
function keyUp(e:KeyboardEvent):void{
    setState(e.keyCode, "up");
}
function setState(keyCode:int, keyState:String):void{
    switch(keyCode){
        case 37:
            arrowLeft = keyState;
            break;
        case 38:
            arrowUp = keyState;
            break;
        case 39:
            arrowRight = keyState;
            break;
        case 40:
            arrowDown = keyState;
            break;
    }
}
```

**Figure 4.** Adding a setup frame and Jump animation to the sprite
Let's look at what's going on here: You are simply setting up a system that will recognize three keyboard keys (the up, left, and right arrow keys) and change the value of a few variables so that you can track which keys a user is pressing. The first few lines add event listeners to the stage:

```actionscript
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyUp);
```

The stage is where keyboard and mouse input occurs, so you have to add the listener there. You're listening for two types of events: `KeyboardEvent.KEY_DOWN` and `KeyboardEvent.KEY_UP`. When those events occur, the functions `keyDown()` and `keyUp()` are triggered, respectively. If you are new to listeners, there is an important thing to note here. Flash* automatically sends a `KeyboardEvent` object as the argument to the function that these listeners link to. That is why your `keyDown` and `keyUp` functions are expecting a `KeyboardEvent` as their argument.

The next section defines three variables: `arrowLeft`, `arrowRight`, and `arrowUp`.

```actionscript
var arrowLeft:String = "up";
var arrowRight:String = "up";
var arrowUp:String = "up";
```

All of these strings started out with the value of `up`. Basically, you want these three variables to be able to tell you if a keyboard key is “up” or “down” at any point in time.

Next are the two functions that the listeners link to:

```actionscript
function keyDown(e:KeyboardEvent):void{
   setKeyState(e.keyCode, "down");
}
function keyUp(e:KeyboardEvent):void{
   setKeyState(e.keyCode, "up");
}
```

These functions are automatically passed a `KeyboardEvent` object, which you assign to `e`. The `KeyboardEvent` object contains several properties that would tell you everything you wanted to know about the event, but all you really care about in this case is its `keyCode` property. The `keyCode` property is an `uint` (unsigned integer) that corresponds to the key that was pressed. Each key on the keyboard has a unique number, except for duplicate keys like the Shift keys, which will
share the same number. For now, you're simply passing this value to a function, `setKeyState()`, along with a string of down or up, depending on whether it was a key press or a key release.

The last part of this script is a function that will take the keyboard input and adjust the variables accordingly:

```javascript
function setKeyState(keyCode:int, keyState:String):void{
    switch(keyCode){
        case 37:
            arrowLeft = keyState;
            break;
        case 38:
            arrowUp = keyState;
            break;
        case 39:
            arrowRight = keyState;
            break;
        default:
            trace("unregistered key press: ", keyCode);
    }
}
```

The `keyCode` argument is passed e.keyCode from the KeyboardEvent, and the `keyState` is passed either down or up. Then, you run the `keyCode` through a `switch` statement and adjust the arrow variable that corresponds to the number. Now, knowing which `keyCode` number corresponds to which key isn’t intuitive, so I usually set up a `default` path in the `switch` statement that will print out the number of the `keyCode` if it is not detected by one of the `case` statements. That way, if I want to know the number of a key, all I have to do is run the game and press the key, and the number will be traced. In the interest of saving time, Figure 5 provides the `keyCode` numbers to some of the most commonly used keys for Flash* games.

![Keyboard Image](image_url)

**Figure 5.** Commonly used Flash* game keys and their `keyCode`

With a system in place to track which keys a user is pressing, you can begin interpreting these inputs to control your character sprite.
Building the Main Loop Script

This game is only one frame long, so typically, the ActionScript* code for this frame would run once and be done. You need to create a section of code that will be run every frame; I call that function the main loop. This is the majority of your game script; it takes all the user inputs into account and adjusts the position and animation of the character sprite every frame accordingly. An event listener powers this function:

```actionscript
addEventListener(Event.ENTER_FRAME, mainLoop);
function mainLoop(e:Event) {
}
```

The event you listen for is Event.ENTER_FRAME. This event is called every frame, even though the timeline is just sitting on this one frame. So, any script inside this mainLoop function will run constantly throughout runtime. Once again, with listeners, Flash* automatically sends an Event object to the linked function. You don’t even use this Event object, but you have to write the function to expect it, or Flash* will throw an error.

Scripting a Simple Platform Engine

Most Flash* games require a physics system of some sort. You’re going to write an extremely simple one. All it does is adjust the character’s position based on its velocity, apply a pull from gravity, and detect collisions with a ground plane.

Inside the mainLoop function, add the following script:

```actionscript
//apply gravity
robot.velY += 1;

//adjust robot’s position by factoring in velocities
robot.x += robot.velX;
robot.y += robot.velY;

//test collisions
var floorY:int = 300;
if(robot.y >= floorY){
    robot.y = floorY;
    robot.grounded = true;
    robot.velY = 0;
}
```

Remember the tracking variables and the spec constants you made in the sprite’s first frame? Now you’re putting them to work. The first part here applies gravity by adding 1 to the robot’s velY property each frame:
robot.velY += 1;

This setting simulates acceleration caused by gravity by increasing the downward velocity more and more each frame. You can make gravity pull more or less strongly by adjusting the amount to add more or less, respectively.

Next, adjust the position of the robot based on its velocities:

```javascript
robot.x += robot.velX;
robot.y += robot.velY;
```

If you ran that script, the robot would just fall off the bottom of the screen. So, you need to create a ground plane to act as a floor:

```javascript
var floorY:int = 300;
if(robot.y >= floorY){
    robot.y = floorY;
    robot.grounded = true;
    robot.velY = 0;
}
```

You're creating an imaginary ground plane 300 pixels from the top of the stage; if the robot’s $Y$ position ever exceeds 300, you pull it out and set it at 300, set `grounded` to `true`, and clear its $Y$ velocity. Now, the robot will fall until it reaches this ground plane, and then stop. Gravity will continue to affect it, but this script will always pull it out of the ground every frame and make it look like it is sitting still.

That’s actually all you need to do to create a very basic physics system. Now, if you do anything that affects the robot’s `velX` or `velY` property, this system moves it appropriately. The next step is to hook up the user input system to actually affect those velocities.

## Turning User Input into Sprite Control

This last step brings it all together. You need to use the user input to affect the velocities and animations of the sprite. First, add one more function to your script outside the `mainLoop`:

```javascript
function setAnimation(animState:String):void{
    if(robot.anim != animState){
        robot.anim = animState;
        robot.gotoAndPlay(animState);
    }
}
```

This function exists just to make it easier to control the sprite’s animations. Remember the `anim` property that you declared in the sprite’s first frame? Its purpose is simply to track what animation the sprite is playing. The function only adjusts the sprite’s animation if the value it gets is different (`!=`) from the `anim` value. Without that, the script might tell you to play the run
animation every frame, in which case it would keep restarting, and you would only ever see the first frame of it. This function prevents that from happening and keeps you from having to worry about it when you need to adjust the animation in the `mainLoop`.

Next, add the following code to the beginning of the `mainLoop` function. This script interprets the user input and applies it to the sprite:

```javascript
//test for arrow key presses and adjust velocities
if(arrowUp == "down" && robot.grounded){
    robot.velY -= robot.jumpForce; //jump!
    setAnimation("jump");
    robot.grounded = false;
}
if(arrowLeft == "down" && robot.velX > 0 - robot.maxSpeed){
    robot.scaleX = -1; //face the sprite to the left
    robot.velX -= robot.acceleration; //accelerate to the left
    if(robot.grounded){
        setAnimation("run");
    }
}
if(arrowRight == "down" && robot.velX < robot.maxSpeed){
    robot.scaleX = 1; //face the sprite to the right
    robot.velX += robot.acceleration; //accelerate to the right
    if(robot.grounded){
        setAnimation("run");
    }
}
//decelerate the robot if no left/right arrow keys are pressed
if(arrowRight != "down" && arrowLeft != "down" && robot.grounded){
    setAnimation("still");
    if(Math.abs(robot.velX) < robot.acceleration){
        robot.velX = 0;
    }else if(robot.velX > 0){
        robot.velX -= robot.acceleration;
    }else if(robot.velX < 0){
        robot.velX += robot.acceleration;
    }
}
```

Let's take a closer look at each part of that. The first part controls the robot's jump:

```javascript
if(arrowUp == "down" && robot.grounded){
    robot.velY -= robot.jumpForce; //jump!
    setAnimation("jump");
    robot.grounded = false;
}
```

The `if` statement determines whether the user is pressing the up arrow key while the robot is grounded. If both of these conditions test `true`, then you need to make the robot jump. To do that, subtract the robot's `jumpForce` value from its `Y` velocity. (Just as you add gravity to pull downward, subtracting a value causes upward motion.) The physics system will take care of the motion; you simply adjust the velocity. You also set the animation of the robot to `jump` and set its
The grounded property to false, because it is no longer on the ground. The grounded property is reset to true when the robot collides with the ground plane. Because the robot is required to be grounded for this jump script, it makes sure that it cannot jump while in midair.

The next two conditional statements are mirror images of each other. They control lateral movement:

```javascript
if(arrowLeft == "down" && robot.velX > 0 - robot.maxSpeed){
    robot.scaleX = -1; // face the sprite to the left
    robot.velX -= robot.acceleration; // accelerate to the left
    if(robot.grounded){
        setAnimation("run");
    }
}
if(arrowRight == "down" && robot.velX < robot.maxSpeed){
    robot.scaleX = 1; // face the sprite to the right
    robot.velX += robot.acceleration; // accelerate to the right
    if(robot.grounded){
        setAnimation("run");
    }
}
```

First, they each determine whether the user is pressing a left or right arrow key. They also check to make sure that the robot’s velX is within the maxSpeed value to prevent the robot from going too fast. If the conditions check out, you flip the sprite to the correct direction by setting its scaleX property to 1 or -1. (Setting it to 1 makes the sprite display normally; setting it to -1 reverses it along the X axis.) Because your sprite was drawn to face right, you need to flip it if the sprite is moving left. Next, you adjust the robot’s velX by either adding or subtracting its acceleration value. Finally, you set the robot’s animation to run, but only if it is grounded. (You wouldn’t want it to be running in midair!)

The last part of the script controls the sprite when the user is not pressing a key:

```javascript
// decelerate the robot if no left/right arrow keys are pressed
if(arrowRight != "down" && arrowLeft != "down" && robot.grounded){
    setAnimation("still");
    if(Math.abs(robot.velX) < robot.acceleration){
        robot.velX = 0;
    }else if(robot.velX > 0){
        robot.velX -= robot.acceleration;
    }else if(robot.velX < 0){
        robot.velX += robot.acceleration;
    }
}
```

First, you test to see if neither the left nor right arrow is being pressed as well as if the robot is grounded. If so, then you set the robot’s animation to still. The following conditional logic adjusts the robot’s velocity to slow it down or stop it. It determines whether you need to add or subtract the acceleration value or just set the velocity to 0. With this script, releasing the left and right arrow keys causes the robot to slow to a halt and play the still animation.
The Finished Script

Here’s what the finished script should look like:

```ActionScript
//add listeners for key presses
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyUp);
var arrowLeft:String = "up";
var arrowRight:String = "up";
var arrowUp:String = "up";

//handle key presses
function keyDown(e:KeyboardEvent):void{
    setKeyState(e.keyCode, "down");
}
function keyUp(e:KeyboardEvent):void{
    setKeyState(e.keyCode, "up");
}
function setKeyState(keyCode:int, keyState:String):void{
    switch(keyCode){
    case 37:
        arrowLeft = keyState;
        break;
    case 38:
        arrowUp = keyState;
        break;
    case 39:
        arrowRight = keyState;
        break;
    default:
        trace("unregistered key press: ", keyCode);
    }
}

//main loop
addEventListener(Event.ENTER_FRAME, mainLoop);
function mainLoop(e:Event) {
//test for arrow key presses and adjust velocities
    if(arrowUp == "down" && robot.grounded){
        robot.velY -= robot.jumpForce; //jump!
        setAnimation("jump");
        robot.grounded = false;
    }
    if(arrowLeft == "down" && robot.velX > 0 - robot.maxSpeed){
        robot.scaleX = -1; //face the sprite to the left
        robot.velX -= robot.acceleration; //accelerate to the left
        if(robot.grounded){
            setAnimation("run");
        }
    }
    if(arrowRight == "down" && robot.velX < robot.maxSpeed){
        robot.scaleX = 1; //face the sprite to the right
        robot.velX += robot.acceleration; //accelerate to the right
        if(robot.grounded){
            setAnimation("run");
        }
    }
}
```
// decelerate the robot if no left/right arrow keys are pressed
if(arrowRight != "down" && arrowLeft != "down" && robot.grounded){
  setAnimation("still");
  if(Math.abs(robot.velX) < robot.acceleration){
    robot.velX = 0;
  }else if(robot.velX > 0){
    robot.velX -= robot.acceleration;
  }else if(robot.velX < 0){
    robot.velX += robot.acceleration;
  }
}

// apply gravity
robot.velY += 1;

// adjust robot's position by factoring in velocities
robot.x += robot.velX;
robot.y += robot.velY;

// test collisions
var floorY:int = 300;
if(robot.y >= floorY){
  robot.y = floorY;
  robot.grounded = true;
  robot.velY = 0;
}

function setAnimation(animState:String):void{
  if(robot.anim != animState){
    robot.anim = animState;
    robot.gotoAndPlay(animState);
  }
}

If you run the Flash* movie now, you'll be able to control the robot with the arrow keys.

**Expanding the System**

Obviously, this system only sets up the bare minimum for a platform game. But the concepts that you'd need to make a more complicated system would follow the same pattern.

If you wanted to change the way the robot handles, you can easily go into the movie clip and adjust those spec constants to change its behavior. You could increase the `jumpForce` constant to make the sprite jump higher or drop the `acceleration` value to make it more slippery. You could also easily change any of the animations, like making a better jump sequence.

When expanding a system like this into a full-blown game, one possible improvement would be to write the `mainLoop` in a way that allows multiple sprites to be controlled by the same physics engine code rather than hard-coding it to control just the robot. Reusing code increases efficiency and reduces file size, which is increasingly important when creating Flash* games that will run smoothly on mobile phones and netbooks. This sample system keeps all the animations and
physics specs located in the movie clip, which would allow you to easily create other sprite objects with different animations and specs and hook them all up to the same physics system.

A grasp of the concepts covered here should allow you to begin planning out your own Flash* game engines and comfortably implement animated sprites into your games in a controlled and intuitive manner.

**About the Author**

Patrick Fram has been a 2D animator and game developer since childhood. His scripting background began with point-and-click authoring tools, then transitioned into BASIC, HTML, and PHP. Upon the discovery of Flash, Patrick's game development and animation background converged and the program became his primary focus. Patrick Fram is a graduate of the Savannah College of Art and Design with a major in animation and a minor in sequential art. Since college, Patrick has worked a variety of freelance jobs in the areas of illustration, graphic design, web design, advertising, animation for film, television, and gaming, and Flash development for gaming and the web. He has since branched into Flex and Air development and currently works as a professional Flash developer in Atlanta, Georgia.