

AS3 game tutorial

Written by Stephan Meesters, www.keepsake-games.com

Introduction

In this tutorial we will be creating a mouse/shooting game of intermediate difficulty. Basic OOP concepts will be explained as well as some higher-level coding techniques. It has basically got the structure that would work on most kind of flash games. It is advised to keep the source code next to you when reading the tutorial, because not every bit is explained.

The graphics for the player consists of a simple circle with a line attached to it, that rotates according to the mouse movement. Red balls come in a random wave manner and shooting them gives the player points. Touching a red ball will decrease your health.

The finished product can be seen here:
www.keepsake-games.com/games/gametutorial/

The source code is available here:
www.keepsake-games.com/games/gametutorial/gametutorialsource.zip

In case you are using this code to create your own game, please add my name to the credits.

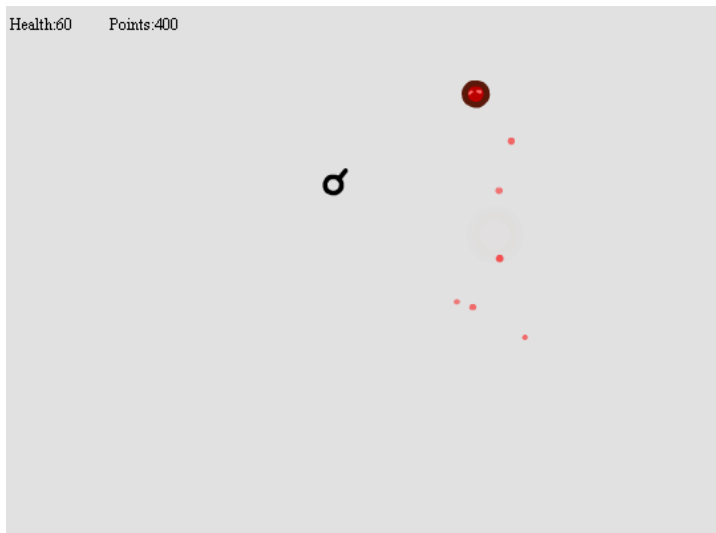


Figure 1: The finished product

1.1 Setting up the document class

In the flash authoring tool (Adobe Flash CS3 in this tutorial) you can configure a document class by clicking on the stage and selecting “document class” in properties. In this tutorial we will name our class “GameBasis”.

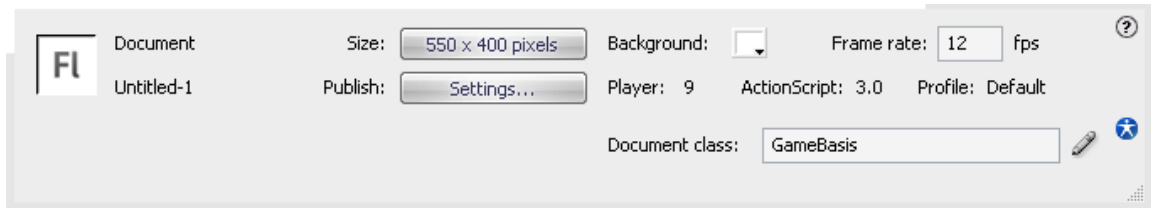


Figure 2: Setting up the document class

The document class is an extension of the main timeline that we see in our FLA file. In this class we have the direct access to `stage` which is useful for getting certain properties like `stage.stageWidth`, `stage.quality`, `stage.frameRate` etc. In the game we will want the stage instance to be accessible anywhere. Because the class extends the timeline, we are obliged to extend `MovieClip` to make it work.

```
package{
    import flash.display.MovieClip;

    public class GameBasis extends MovieClip {

        function GameBasis() {

            // Set stage instances
            STAGE = stage;

        }

        // Static vars
        public static var STAGE;
        public static var STAGE_WIDTH = 550;
        public static var STAGE_HEIGHT = 400;

    }
}
```

Code example 1: The document class in its simplest form [GameBasis.as]

In the above code, starting from the top we see that `package{` has been used because the class is located at the top-level of our project. If for example a class was placed in the folder “banana”, it would be coded as `package banana{`. The constructor `function GameBasis()` is called whenever a new class is instantiated. We set the value of `STAGE` as a static variable. A static variable is a property of a class and is accessible from anywhere in the program as `GameBasis.STAGE`. Static variables and methods are powerful because classes don't necessarily have to be instantiated first before we can access them.

1.2 Adding a button for the main menu

```
// Start button
but_start.x = STAGE_WIDTH/2;
but_start.y = STAGE_HEIGHT/2;

function showMainMenu():void {
    addChild(but_start);
}
function hideMainMenu():void {
    removeChild(but_start);
}

...

var but_start = new start_game_gfx();
```

Code example 2: Creating and placing the start button [GameBasis.as]

For the game to start we will need a start button. Create some text in the stage, select it and press “Convert to symbol”. We name our button “start_game” and by analogy we export it named “start_game_gfx”. By exporting to Actionscript we can use this object anywhere in the game by creating a new instance of it using `new start_game_gfx()`. You can create other graphics for the button in the “over” and “down” state in the flash authoring tool.

To provide a good way to add and remove our main menu when we want, we’ve created two functions that add or delete the button to the display list. The display list is unique for every class. Because we are in the document class, the button will be added to the main timeline.



Figure 3: Start game button “up” graphics.



Figure 4: Start game button “over” graphics.



Figure 5: Start game button “hit” graphics.

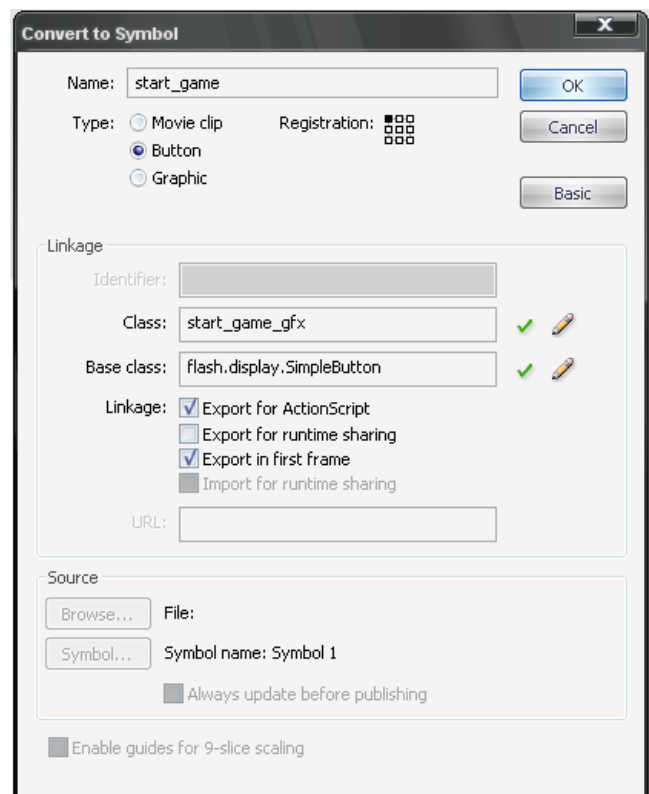


Figure 6: Start game button Export settings

1.3 Game control functions

```
import flash.events.Event;
import flash.events.MouseEvent;
import game.*;

...

but_start.addEventListener(MouseEvent.CLICK, startGame);

function startGame(e:MouseEvent) {

    // Game init
    g = new initGame();
    g.addEventListener("GAME_OVER", resetGame);

    g.start();
    addChild(g);

    hideMainMenu();
}
function resetGame(e: Event) {
    removeChild(g);
    g.removeEventListener("GAME_OVER", resetGame);
    g = null;
    showMainMenu();
}

var g:initGame;
```

Code example 3: The game control functions in the document class [GameBasis.as]

Next we are going to extend the game with a class called `initGame`. This class contains everything we need to make a game function, and creating a new instance of it creates a new “round”. When the player dies and returns to the main menu, the variable `g` (holding the `initGame` instance) is deleted and old variables will be disposed of. This allows us to start fresh each time we click the “start game” button.

An event listener is added for “GAME_OVER”, which responds to a custom event that is thrown inside the `initGame` class. A custom event can be thrown by `DispatchEvent(new Event("GAME_OVER"))`. This is just a way for the `initGame` class to communicate with its parent class `GameBasis`. Notice that `resetGame` expects the `Event` datatype as parameter, and `startGame` the `MouseEvent` datatype.

2.1 Creating the initGame class

```
package game{
    import flash.display.MovieClip;
    import flash.events.Event;
    import flash.ui.Mouse;

    public class initGame extends MovieClip{

        //---> Game control

        public function start():void {

            addEventListener(Event.ENTER_FRAME,update,false,0,true);
            Mouse.hide();

            // Add graphics
            create_enviroment();

        }
        public function pause():void {
            removeEventListener(Event.ENTER_FRAME,update);
        }

        //---> Graphics control

        private function create_enviroment():void {
            // Add background
            addChild(bg);

            // Add player
            addChild(p);
            p.x = GameBasis.STAGE_WIDTH/2;
            p.y = GameBasis.STAGE_HEIGHT/2;
            p.mouseChildren = false;
            p.mouseEnabled = false;
        }

        private function update(e:Event):void {

            // Timer
            t++;
        }

        private var t:int;
        private var p:player = new player();
        private var bg:MovieClip = new background_gfx;

    }
}
```

Code example 4: initGame class in its simplest form [initGame.as]

Here the most basic functions of `initGame` are shown. You can see that `start()` is public while the rest are private, which means that only `start()` can be accessed by its parent class `GameBasis`. If a class attribute (i.e. `private`, `protected`) is omitted then it is configured as `internal`, which means that it can only be accessed from within its own package.

In `start()` we've added a listener that triggers the `update()` function every new keyframe: the gameloop. `false,0,true` has been added to set the listener as a weak listener, meaning that it is automatically removed by garbage collection when all references to the `initGame` class are gone. Everything in the game will depend on this gameloop, so we can easily start and stop the game whenever we want to.

2.2 Making the player move and rotate

We can see that the player got his own class. Open up `player.as` and you will see that it is nothing more than a class with the player graphics added to its display list. This is done for the purposes of easy expansion later on. Inside `create_enviroment()` we set `mouseChildren` and `mouseEnabled` as false, which means that it is ignored entirely for mouse clicks.

```
private function update(e:Event):void {
    movePlayer();

    ...
}
private function movePlayer():void {
    var calc:Object = calcPlayerDif();
    if(!calc) return;

    p.x += calc.dx/10;
    p.y -= calc.dy/10;

    p.rotation = Math.atan2(calc.dx,calc.dy)*57;
}
private function calcPlayerDif():Object {
    var newX:Number = mouseX;
    var newY:Number = mouseY;

    var dx:Number = newX - p.x;
    var dy:Number = p.y - newY;

    // Bugfix for unwanted behaviour at low dx and dy.
    if(dx * dx + dy * dy < 9) return false;
    else {
        lastPlayerDX = dx;
        lastPlayerDY = dy;
    }

    return {dx:dx, dy:dy};
}
private var lastPlayerDX = 0;
private var lastPlayerDY = 0;
```

Code example 5: Player movement functions [`initGame.as`]

To make the player move, we add the `movePlayer()` function to the gameloop. In `calcPlayerDif()` we first calculate the difference in the x and y direction between the current mouse position and the current player position. This difference is divided by 10 later on, which means that the player will move faster when the difference is larger. The speed decreases exponentially which gives a very smooth effect.

We make the player rotate according to its latest difference in position by using `Math.atan2(calc.dx,calc.dy)*57`. The 57 is simply $180 / \pi$ already precalculated, which saves us one calculation each loop. Because dx and dy can become 0 when the player does not move, the rotation can start to mess up. This is why a bugfix has been added to `calcPlayerDif()` that checks when the combined value of dx and dy is low, and complete skips the player movement update if that is the case.

Notice that the `calcPlayerDif()` functions returns a Object as value. This is a good way to return multiple variables from a function as it is easily accessible. It would be faster to merge `calcPlayerDif()` with `movePlayer()` as this decreases overhead and therefore increases speed, but for the purposes of this tutorial we have not done this.

2.2 Adding linear moving objects

Lets first consider the basic class used for our moving objects, which originates from my own KSG code library. The first use for it will be for player bullets.

```
package lib.KSG{
    public class KSGLinearMO extends KSGDisplayObject{
        function KSGLinearMO(gfx, dx, dy, speed) {
            this.dx = dx;
            this.dy = dy;
            this.speed = speed;

            calcSlope();

            super(gfx);
        }
        private function calcSlope(){
            var abs = Math.sqrt(Math.pow(dx,2)+Math.pow(dy,2));
            dx /= abs;
            dy /= abs;
            dx *= speed;
            dy *= speed;
        }
        public function move(){
            this.x += dx;
            this.y -= dy;
        }
        private var gfx;
        private var speed;

        public var dx;
        public var dy;
    }
}
```

Code example 6: The class KSGLinearMo, used for linear movement [KSGLinearMO.as]

The class uses dx and dy to calculate the direction the bullet moves. These two combined gives the velocity vector of the bullet. The vector is normalised by dividing it by its own length, which makes the vector of length one. Then it is multiplied by speed. The advantage of this method is that it moves equally fast in any position the bullet needs to go, and that there are no problems like zero-division.

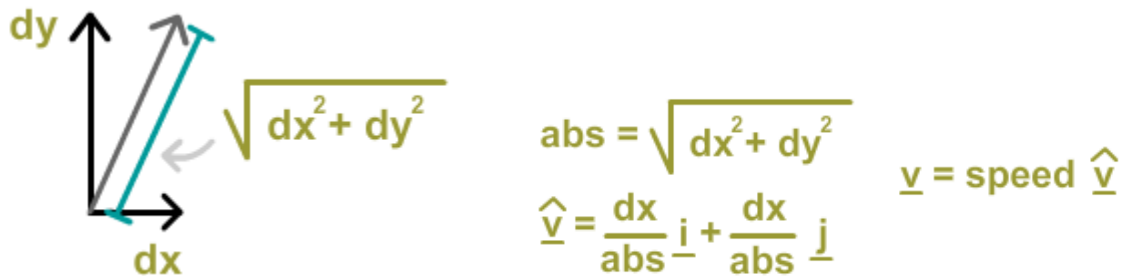


Figure 7: dx, dy and their combined length Figure 8: The speed vector in formula form

The graphics for the linear moving object are also supplied as a parameter, and then passed on to the parent function [KSGDisplayObject](#) by [super\(gfx\)](#). This is an example of the OOP concept polymorphism, which means that you extend another class and inhere its properties and methods. By using [super\(\)](#) you pass the parameter on to the constructor of its parent function. This will be demonstrated in the following example.

```
package game{
    import flash.display.MovieClip;
    import lib.KSG.KSGLinearMO;
    public class PlayerBullet extends KSGLinearMO{
        function PlayerBullet(dx,dy){
            speed += (dx+dy)/25;
            super(gfx,dx,dy,speed);
        }
        var gfx:MovieClip = new player_bullet_gfx;
        var speed = 10;
    }
}
```

Code example 7: The class `PlayerBullet` [`PlayerBullet.as`]

Here we see our `PlayerBullet` class, extending `KSGLinearMO`. The reason the player bullet has its own class, is because we can give them certain properties like graphics and speed. Also it is useful because of its datatype. When we later create an array of all kinds of objects, we can test if an object is of the type `PlayerBullet`, and by those means identify the object. You can see that we use the `super` to pass all the variables needed, and that it matches up to the parameters that `KSGLinearMO` requires.

```
public function start():void {  
  
    addEventListener(MouseEvent.MOUSE_DOWN, shootBullet);  
  
    ...  
  
}  
  
private function shootBullet(e:MouseEvent):void {  
  
    var bullet:PlayerBullet = new  
    PlayerBullet(lastPlayerDX, lastPlayerDY);  
    mov_ob_array.push(bullet);  
    bullet.x = p.x;  
    bullet.y = p.y;  
    addChild(bullet);  
}  
  
private var mov_ob_array:Array = new Array();
```

Code example 8: Code to shoot a `PlayerBullet` [`initGame.as`]

In the `shootBullet` function we create a new instance of `PlayerBullet` and add this to `mov_ob_array`, the array that will contain all the moving objects. Each of these moving objects will contain the function `move()`, which can be found in `KSGLinearMO`. In the gameloop we will loop over the function and invoke the `move()` function which puts everything in motion.

2.3 Adding enemies

As we are in the subject of adding moving objects, we will talk about adding enemies.

```
private function update(e:Event):void {  
  
    ...  
  
    if(t%(65+int(Math.random()*20))==0 && t>30){  
        SpawnEnemy();  
    }  
  
    ...  
}  
private function SpawnEnemy():void {  
    var angle = Math.random()*2*Math.PI;  
    var xpos = Math.cos(angle)*500 + GameBasis.STAGE_WIDTH/2;  
    var ypos = -Math.sin(angle)*500 + GameBasis.STAGE_HEIGHT/2;  
  
    var dx = p.x - xpos + Math.random()*5-2.5;  
    var dy = ypos - p.y + Math.random()*5-2.5;  
  
    var enemy:Enemy = new Enemy(dx,dy);  
    mov_ob_array.push(enemy);  
    enemy.x = xpos;  
    enemy.y = ypos;  
    addChild(enemy);  
}
```

Code example 9: Code to add enemies [initGame.as]

Enemies are added based on the current time t . By using the modulo operator $\%$ we can see when the rest value after division equals zero, and give us a way to add enemies (i.e. doing $6\%4$ will divide 6 by 4, and the rest value is 2, so $6\%4=2$). My adding `Math.random()` to it, we get a different number in each loop and therefore a way to randomly spawn enemies.

We want to make the start point of a spawned enemy randomly on a circle of radius 500 with its center in the middle of the screen. It will move towards the players position, with a slight randomness added to it. The Enemy class is almost the same as the PlayerBullet class, but with different properties.

2.4 Moving the moving objects

The largest bit of code so far, here it goes...

```
// Process the moving objects. (bullets, enemies etc.)
for (var i=0; i<mov_ob_array.length; i++){
    var ob = mov_ob_array[i];
    ob.move();

    if(ob is Enemy){
        var bRemoving:Boolean = false;

        // Check enemy with player position
        if( (ob.x-p.x)*(ob.x-p.x) + (ob.y-p.y)*(ob.y-p.y) < 150 ){
            GameBasis.stats.health -= ENEMY_DAMAGE;
            GameBasis.stats.points += ENEMY_CRASHPOINTS;
            textfield_health.refresh();
            textfield_points.refresh();
            bRemoving = true;
            if(GameBasis.stats.health<=0) gameOver();
        }
        // Else check enemy with player bullets positions
        else{
            for (var j=0; j<mov_ob_array.length; j++){
                var ob2 = mov_ob_array[j];
                if(ob2 is PlayerBullet){
                    if( (ob2.x-ob.x)*(ob2.x-ob.x) + (ob2.y-
ob.y)*(ob2.y-ob.y) < 150 ){
                        GameBasis.stats.points +=
ENEMY_POINTS;
                        textfield_points.refresh();
                        bRemoving = true;
                        break;
                    }
                }
            }
        }

        // If collision is found, remove the enemy. The tweener
        // class is used as an easy way for extra graphics when the
        // enemy dies.
        if(bRemoving){
            Tweener.addTween(ob, {alpha:0, scaleX:2, scaleY:2,
time:1,onComplete:function(){ if(contains(ob))
removeChild(ob); }});
            var sp = new KSGLinearParticles(ob.x,ob.y,new
red_particle_gfx);
            addChild(sp);
            mov_ob_array.splice(i,1);
            continue;
        }
    }

    // Remove when out of screen
    if(ob.x < -750 || ob.x > GameBasis.STAGE_WIDTH+750 ||
```

```

        ob.y < -750 || ob.y > GameBasis.STAGE_HEIGHT+750) {
            mov_ob_array.splice(i,1);
            if(contains(ob)) removeChild(ob);
        }
    }

    // Game constants
    private const ENEMY_DAMAGE = 20;
    private const ENEMY_POINTS = 50;
    private const ENEMY_CRASHPOINTS = 100;

```

Code example 10: The loop over the moving object array, mov_ob_array [initGame.as]

We start by adding custom behavior for the [Enemy](#) objects. We do this by checking if the object is of the [Enemy](#) datatype. First we check the distance to the player to see if there is a collision. In case it is true, we add points to the total list of points and decrease the health of the player. Then we update the text fields showing them. How these are made will be shown later. For checking distance we would normally use a square root, but this can be omitted since we are only looking at distance and don't require exact numbers to work with. To find the right value you have to play around with it a bit.

If the enemy does not collide with the player, it goes on to check if it hits any [PlayerBullet](#) objects. When bRemoving becomes set to true, the Enemy object is spliced from the array and removed from the display list. We make use of Tweeners, a 3rd party AS3 library, to create some smooth graphics for the enemy. A class from my own KSG library is used to create the particles. How the particles work is outside the scope of this tutorial.

Lastly, if any object in the list goes far outside the screen it will be spliced and removed. This frees up memory and stops the game from running slower in time.

2.5 Game over and return to menu

```
private function gameOver():void {
    GameBasis.stats.health = 0;
    pause();
    removeEventListener(MouseEvent.MOUSE_DOWN, shootBullet);
    Mouse.show();
    addChild(gameOverScreen);
    gameOverScreen.alpha = 0;
    Tweener.addTween(gameOverScreen, {alpha:1, time:3,
    transition:"easyOutQuint", onComplete:returnToMenu});
    Tweener.addTween(p, {alpha:0, scaleX:2, scaleY:2, time:.5});
}
private function returnToMenu(){
    // Empty the display list of this instance
    var _scope:DisplayObjectContainer = this;
    while(_scope.numChildren > 0)
        _scope.removeChildAt(_scope.numChildren-1);

    // Dispatch event to be caught by GameBasis
    dispatchEvent(new Event("GAME_OVER"));
}
```

Code example 11: Game over and return to menu functions [initGame.as]

As can be seen in the moving object array loop, when the health drops down below 0 the `gameOver()` function is called. We don't want the health to drop below 0, so it is set at 0 in the function. Furthermore the gameloop is paused and the mouse shown again. We pop up the game over screen and use Tweener to animate it. After the animation is done, the `returnToMenu()` function is called. To ensure that we don't leave any unwanted objects in the display list, we loop over our displaylist and remove items until the number of children is zero. Finally an event is dispatch that will take us back to the main menu.

2.6 Game instance variables and textfields

We've seen how the variables `GameBasis.stats.health` and `GameBasis.stats.points` got used, with their textfields, but have not yet discussed how there are created.

In `GameBasis.as` we add the following code:

```
public static var stats = new statsHolder();
```

Code example 12: Adding stats [GameBasis.as]

Which creates an instance of [statsHolder](#), seen here:

```
package game{
    public class statsHolder{
        public function set health(v){
            varOb["health"] = v;
        }
        public function get health(){
            return varOb["health"];
        }
        public function set points(v){
            varOb["points"] = v;
        }
        public function get points(){
            return varOb["points"];
        }
        public function getVar(name){
            return varOb[name];
        }
        private var varOb:Object = new Object();
    }
}
```

Code example 13: The statsHolder class [statsHolder.as]

In statsHolder we use the so called setter and getter functions. Through these functions we can directly access health and points by [stats.health](#) and [stats.points](#). When setting a variable, it is added to the [varOb](#) object.

Using a class to hold variables is handy, because you can pass an instance of stats to another class and save it there. To get the variable you want, you only have to pass “health” or “points” along with it. By doing this you can get the newest values without having to send any parameters along.

I.e. when instantiating the health textfield class, you send the stats object along with “health” to its constructor, and to update the textfield you only have to call [textfield_health.refresh\(\)](#). It is advised to only update the textfield when you know the variable has actually changed.

The health textfield is instantiated in initGame.as as follows:

```
private function create_enviroment():void {
    ...
    textfield_health = new KSGTextField(5,5,GameBasis.stats,
                                     "health","Health:");
    addChild(textfield_health);
}

// Graphic objects
private var textfield_health:KSGTextField;
```

Code example 14: Adding textfields to the game [initGame.as]

Here the parameters the KSGTextField class expects are (x, y, statobject, statname, textvalueBegin, textvalueEnd, font, size, color)

3.1 Adding sounds

There is a handy class available in the KSG library for sound effects. We create a sound factory in GameBasis.as like this:

```
public static var sfxFactory = new KSGSoundFX(bMuted);
sfxFactory.addSound(["player_shoot", new shooting_sound()]);
sfxFactory.addSound(["enemy_hit", [new enemy_hit(),
                                   new enemy_hit2(),
                                   new enemy_hit3()
                                   ]]);
```

Code example 15: Adding the sound factory [GameBasis.as]

We make the sfxFactory static so it can be accessed from anywhere. In the FLA symbol library we have imported 4 sounds and exported them for actionscript. To add the sounds to the factory, we add the name of the sound and all the sound objects associated with it. Adding more sound objects to one name makes it randomize it automatically.

To play the sound effect, we simply use this code for example:

```
GameBasis.sfxFactory.play("player_shoot");
```

Code example 15: Playing a sound [initGame.as]

That's all!

I hope this tutorial has been helpful to you. If you have any questions, you can mail them to [keepsakegames at gmail dot com](mailto:keepsakegames@gmail.com)

Stephan Meesters © 2009