Sign up for our free weekly Web Developer Newsletter.

CODE PROJECT For those who code

We've got your backend covered. Avoid these data access debacles for your next app FREE WHITE PAPER IBM Cloudant

Mario5

Florian Rappl, 5 Jun 2012 CPOL

4.99 (149 votes) Rate:

Recreating a famous jump and run game for playing and creating own levels in the webbrowser.

Download the demo project including the source - 0.99 MB

Introduction

In the history of computer games some games have created and carried whole companies on their shoulders. One of those games is certainly *Mario Bros*. The Mario character first appeared in the game *Donkey Kong* and became very famous within its own game series starting with the original *Mario Bros.* in 1983. Nowadays a lot of spin-offs and 3D jump and runs are being produced centering the Mario character. In this article we will develop a very simple *Super Mario* clone, which is easily extendible with new items, enemies, heros and of course levels.

The code of the game itself will be written in object oriented JavaScript. Now that sounds like a trap since JavaScript is a prototype based scripting language, however, there are multiple object oriented like patterns possible. We will investigate some code which will give us some object oriented constraints. This will be very useful to stay in the same pattern through the whole coding.

Background

The original version of this application was developed by two students who took my lecture on "Programming Web applications with HTML5, CSS3 and JavaScript". I gave them a basic code for the engine and they developed a game including a level editor, sounds and graphics. The game itself did not contain a lot of bugs, however, the performance was rather poor and due to rare usage of the prototype properties the extensibility was also limited. The main performance burner was the usage of the jQuery plug-in Spritely (which can be found here). In this case I am the one to blame, since I recommended using it for simplicity. The issue here is that Spritely itself does a good job on doing one animation, but not a hundred or more. Every new animation (even though spawned at the same moment) will get its own timed interval recall loop.

For this article I decided to focus on the main things of the game. Through this article we will rewrite the whole game - with the benefits as explained above:

- The game will be easily extendible
- The performance will not suffer much by animating objects
- The start or pause action of the game will have a direct impact on all elements
- The game will not rely on external elements like sounds, graphics, ...

The last statement sounds like a maniac writing this article. However, this is in my opinion quite important. Let's take the following code to show my point:

Hide Copy Code

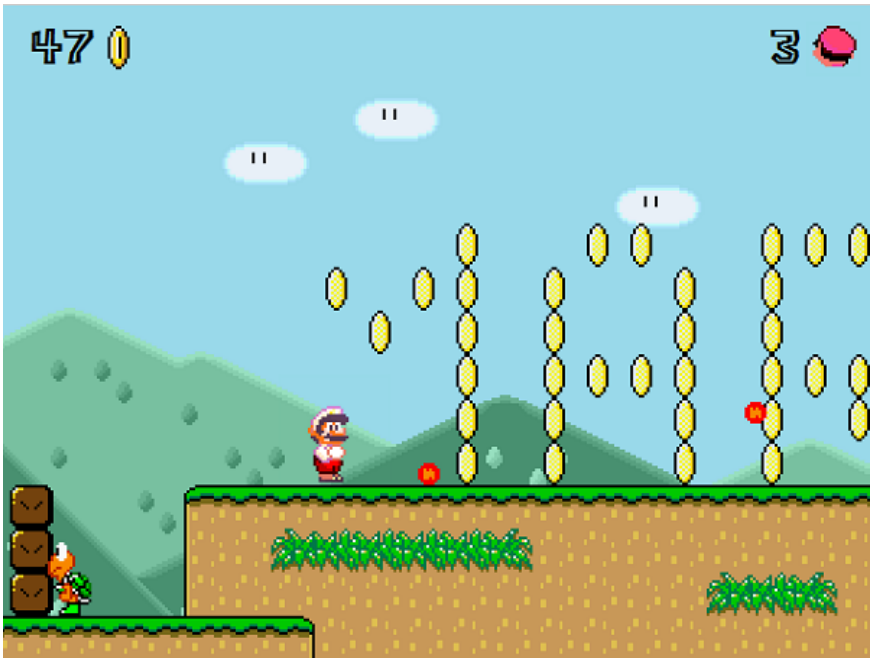
```
$(document).ready(function() {
    var sounds = new SoundManager();//**
    var level = new Level('world');//world is the id of the corresponding DOM container
    level.setSounds(sounds);//*
    level.load(definedLevels[0]);
    level.start();
    keys.bind();
});
```

Now this does not look so nasty, but this is in fact everything that is needed to play a game of Mario with HTML5. The details we have left out here will all be explained later. Back to the statement of above we see the line marked with a two star comment (`//**`): Here a new instance of the sound manager class is created. This one will also load sound effects. What if we want to skip this line? We would not have a working instance of the sound manager. Now the next thing to notice is that the instance of the sound manager is not saved in a global scope, but just locally. We can do that because no object in the whole game requires a certain instance of this class. What is done instead? If an object wants to play a sound it calls a method that is provided by the level (every object has to belong to a level in order to exist; since a level class is the only place where the game objects are created).

Now this is where the line with the single star comment (`//*`) comes into play. If we do not call the `setSounds()` method of a level instance the level will not have a proper sound manager class instance attached. Therefore all requests to play a sound by any object will be trashed. This makes the sound manager class plugable, since we just have to remove two lines of code to completely remove the sound manager. On the other side we only have to add two lines of code. This of course would be something that can be achieved more elegant within C# by using reflection (as required by dependency injection or other patterns).

The rest of this small code is just to load a starting level (here we use the first one in a list of predefined levels) and start it. The global keyboard object called `keys` can `bind()` or `unbind()` all key events in the document.

The basic design



We will skip the sound manager implementation in this article. There will be another article about a good level editor and various other interesting things, one of them will be the implementation of the sound manager. The basic document outline for the Super Mario game looks like the following:

Hide Copy Code

```
<!doctype html>
<html>
<head>
<meta charset=utf-8 />
<title>Super Mario HTML5</title>
<link href="Content/style.css" rel="stylesheet" />
</head>
<body>
<div id="game">
<div id="world">
</div>
<div id="coinNumber" class="gauge">0</div>
<div id="coin" class="gaugeSprite"></div>
<div id="liveNumber" class="gauge">0</div>
<div id="live" class="gaugeSprite"></div>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
<script src="Scripts/testlevels.js"></script>
<script src="Scripts/oop.js"></script>
<script src="Scripts/keys.js"></script>
<script src="Scripts/sounds.js"></script>
<script src="Scripts/constants.js"></script>
<script src="Scripts/main.js"></script>
</body>
</html>
```

So after all we do not have much markup here. We see that the world is contained within a game area. The game area does include the gauges (and some sprites to animate the gauges). This simple mario game only contains two gauges: one for the coins and another one for the number of lives.

A really important section is the list of JavaScripts that will be included. For performance reasons we place them at the bottom of the page. This is also the reason why we get jQuery from a CDN (which is Google in this case). The other scripts should be packed into one and minimized (this is called bundling and is one of the included features of ASP.NET MVC 4). We will not bundle these scripts for this article. Let's have a look at the content of those script files:

- The *testlevels.js* file is quite big and should be minimized. It contains the premade levels. All of those levels where implemented by the two students of my lecture. The first level is a clone of the first level of Super Mario Land for the classic GameBoy (if I am not wrong here).
- The *oop.js* file contains the code to simplify object oriented JavaScript. We will discuss that in a minute.
- The *keys.js* file creates the `keys` object. If we want to design a multiplayer or other features we should also modulize this script (as we did with the sound manager class).
- The sound manager is written in the *sounds.js* file. The basic idea is that the Web Audio API (Official Website) could possibly overtake the current API. Right now the problem is that the Web Audio API is limited to Google Chrome and some nightly builds of Safari. If the distribution is right this could be the way to get the audio we need for games in the browser.
- The file *constants.js* contains enumerations and very basic helper methods.
- All other objects are bundled in the file *main.js*.

Before we go into details of the implementation we should have a look at the CSS file:

Hide Shrink Copy Code

```
@font-face {
   font-family: 'SMB';
   src: local('Super Mario Bros.'),
        url('fonts/Super Mario Bros.ttf') format('truetype');
   font-style: normal;
}

#game {
    height: 480px; width: 640px; position: absolute; left: 50%; top: 50%;
    margin-left: -321px; margin-top: -241px; border: 1px solid #ccc; overflow: hidden;
}
#world {
    margin: 0; padding: 0; height: 100%; width: 100%; position: absolute;
    bottom: 0; left: 0; z-index: 0;
}
.gauge {
    margin: 0; padding: 0; height: 50px; width: 70px; text-align: right; font-size: 2em;
    font-weight: bold; position: absolute; top: 17px; right: 52px; z-index: 1000;
    position: absolute; font-family: 'SMB';
}
.gaugeSprite {
    margin: 0; padding: 0; z-index: 1000; position: absolute;
}
#coinNumber {
    left: 0;
}
#liveNumber {
    right: 52px;
}
#coin {
    height: 32px; width: 32px; background-image : url(mario-objects.png);
    background-position: 0 0; top: 15px; left: 70px;
}
#live {
    height: 40px; width: 40px; background-image : url(mario-sprites.png);
    background-position : 0 -430px; top: 12px; right: 8px;
}
.figure {
    margin: 0; padding: 0; z-index: 99; position: absolute;
}
.matter {
    margin: 0; padding: 0; z-index: 95; position: absolute; width: 32px; height: 32px;
}
```

Now this not very long again (but we did not have much markup either). On top we introduce some fancy font in order to give our game a Mario like look (type-wise). Then we just set up everything for the characteristic 640 x 480 pixel resolution. It is quite important that the game does hide any overflow. Therefore we can just move our world in the game. This means the game acts like a kind of view. The level itself is placed in the world. The gauges are placed in the first rows of the view. Every figure will have a CSS class called `figure` attached. The same goes for matter like ground, decoration elements or items: those elements have a CSS class called `matter`. Quite important is the property `z-index`. We always want a dynamic object to be in front of a static one (there are exceptions, but we will come to that later).

Object oriented JavaScript

Doing object oriented coding with JavaScript is not hard, but a little bit messy. One of the reasons for this is that there are multiple ways to do it. Every way has its own advantages and disadvantages. For this game we want to follow one pattern strictly. Therefore I am proposing the following way:

Hide Shrink Copy Code

```
var reflection = {};

(function(){
    var initializing = false, fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/;

    // The base Class implementation (does nothing)
    this.Class = function(){ };

    // Create a new Class that inherits from this class
    Class.extend = function(prop, ref_name) {
        if(ref_name)
            reflection[ref_name] = Class;

        var _super = this.prototype;

        // Instantiate a base class (but only create the instance,
        // don't run the init constructor)
        initializing = true;
        var prototype = new this();
        initializing = false;

        // Copy the properties over onto the new prototype
        for (var name in prop) {
        // Check if we're overwriting an existing function
```

```
        prototype[name] = typeof prop[name] == "function" &&
            typeof _super[name] == "function" && fnTest.test(prop[name]) ?
            (function(name, fn) {
                return function() {
                    var tmp = this._super;

                    // Add a new ._super() method that is the same method
                    // but on the super-class
                    this._super = _super[name];

                    // The method only need to be bound temporarily, so we
                    // remove it when we're done executing
                    var ret = fn.apply(this, arguments);
                    this._super = tmp;

                    return ret;
                };
            })(name, prop[name]) :
            prop[name];
        }

        // The dummy class constructor
        function Class() {
            // All construction is actually done in the init method
            if ( !initializing && this.init )
                this.init.apply(this, arguments);
        }

        // Populate our constructed prototype object
        Class.prototype = prototype;

        // Enforce the constructor to be what we expect
        Class.prototype.constructor = Class;

        // And make this class extendable
        Class.extend = arguments.callee;

        return Class;
    };
})();
```

This code is highly inspired by prototype and has been constructed by John Resig. He wrote an article about the whole coding issues at his blog (Article from John Resig about the OO JavaScript code). The code is wrapped in a directly executed anonymous method in order to scope internal variables. The `Class` object is an extension to the `window` object (which should be the underlying object, i.e., `this` if this script file is executed from a web browser).

My extension to this code is the possibility to name the class. JavaScript does not have the powerful reflection properties, which is why we are required to put a little bit more effort into the class's description process. When assigning the constructor to a variable (which we will see in a moment) we have the option to pass a name of the class as second argument. If we do this then a reference to the constructor is being placed in the object `reflection` with the second argument as the property name.

A simple class construction is the following:

Hide Copy Code

```
var TopGrass = Ground.extend({
    init: function(x, y, level) {
        var blocking = ground_blocking.top;
        this._super(x, y, blocking, level);
        this.setImage(images.objects, 888, 404);
    },
}, 'grass_top');
```

Here we are creating a class called `TopGrass`, which inherits from the `Ground` class. The `init()` method represents the constructor of the class. In order to call the base constructor (which is not required) we have to call it over the `this._super()` method. This is a special method which can be called within any overridden method.

One important remark here: It is important to distinguish here between real polymorphism (which can be done in object oriented languages with static types like C#) and the one presented here. It is obviously not possible to access the parent's method from outside (since we cannot change how we see the object - it's always a dynamic object). So the only way to access the method of the parent is to call the `this._super()` method within the corresponding overridden method. It should also be noted that this statement is not absolute, however, it is true with the code presented above.

The `Ground` class is quite uninteresting (just a middle layer). So let's have a look at the base class of `Ground`, called `Matter`:

Hide Copy Code

```
var Matter = Base.extend({
    init: function(x, y, blocking, level) {
        this.blocking = blocking;
        this.view = $(DIV).addClass(CLS_MATTER).appendTo(level.world);
        this.level = level;
        this._super(x, y);
        this.setSize(32, 32);
```

```
        this.addToGrid(level);
    },
    addToGrid: function(level) {
        level.obstacles[this.x / 32][this.level.getGridHeight() - 1 - this.y / 32] = this;
    },
    setImage: function(img, x, y) {
        this.view.css({
            backgroundImage : img ? c2u(img) : 'none',
            backgroundPosition : '-' + (x || 0) + 'px -' + (y || 0) + 'px',
        });
        this._super(img, x, y);
    },
    setPosition: function(x, y) {
        this.view.css({
            left: x,
            bottom: y
        });
        this._super(x, y);
    },
});
```

Here we extend the `Base` class (which is one of the top classes). All classes that inherit from `Matter` are static 32 x 32 pixel blocks (they cannot move) and contain a blocking variable (even though it could be set to no blocking, which is the case e.g. for decorations). Since every `Matter` as well as `Figure` instance represents a visible object we need to create a proper view for it (using jQuery). This also explains why the `setImage()` method has been extended in order to set the image on the view.

The same reason can be applied for overriding the `setPosition()` method. The `addToGrid` method has been added in order to give child classes the possibility to deactivate the standard behavior of adding the created instance to the `obstacles` array of the given level.

Controlling the game

The game is basically controlled by the keyboard. Therefore we need to bind the corresponding event handlers to the document. We are just interested in a few keys, which can be pressed or released. Since we need to monitor each key's state we just extend the object `keys` with the corresponding properties (like `left` for the left key, `right` for the right key and so on). All we need to call are the methods `bind()` to bind the keys to the document or `unbind()` to release them. Again we are using jQuery to do the actual work in the browser for us - giving us more time to spent on our problems instead of any cross or legacy browser issues.

Hide Shrink Copy Code

```
var keys = {
    //Method to activate binding
    bind : function() {
        $(document).on('keydown', function(event) {
            return keys.handler(event, true);
        });
        $(document).on('keyup', function(event) {
            return keys.handler(event, false);
        });
    },
    //Method to reset the current key states
    reset : function() {
        keys.left = false;
        keys.right = false;
        keys.accelerate = false;
        keys.up = false;
        keys.down = false;
    },
    //Method to delete the binding
    unbind : function() {
        $(document).off('keydown');
        $(document).off('keyup');
    },
    //Actual handler - is called indirectly with some status
    handler : function(event, status) {
        switch(event.keyCode) {
            case 57392://CTRL on MAC
            case 17://CTRL
            case 65://A
                keys.accelerate = status;
                break;
            case 40://DOWN ARROW
                keys.down = status;
                break;
            case 39://RIGHT ARROW
                keys.right = status;
                break;
            case 37://LEFT ARROW
                keys.left = status;
                break;
            case 38://UP ARROW
                keys.up = status;
                break;
            default:
                return true;
        }
```

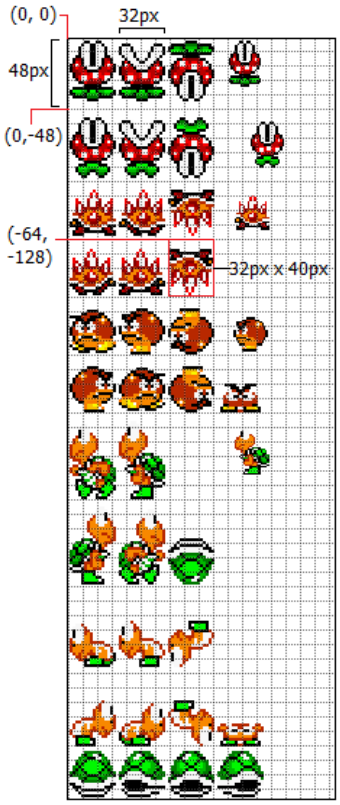
```
        event.preventDefault();
        return false;
    },
    //Here we have our interesting keys
    accelerate : false,
    left : false,
    up : false,
    right : false,
    down : false,
};
```

Another way to set this up would be to use the power of jQuery. We could assign the same method to the up and down key event with some custom argument. This argument would then determine which event was responsible for the method call. However, this way it is a little bit cleaner and more understandable.

CSS Spritesheets

All graphics in the game are done by using CSS spritesheets. This feature is quite simple to use if we know the following lines. First of all in order to use an image as a spritesheet we need to assign the image as a background image to the corresponding element. It is important to not disable the background repeat, since this will give us the advantage of periodic boundary conditions. Usually this would not result in any bad consequences, however, with periodic boundary conditions we can do a lot more.

The next step is then to set a kind of offset to the background image we just assigned. Usually the offset is just (0, 0). This means that we top left coordinate of our element is also the top left coordinate of our spritesheet. The offset we will enter is relative to the element, i.e. by setting an offset of (20, 10) we would set the top left (0, 0) coordinate of the spritesheet to 20 pixels to the left side and 10 pixels to the top side of the element. If we use (-20, -10) instead, we will have the effect of having the top left (0, 0) coordinate of the spritesheet outside of our element. Therefore the visible part of the image will be within the image (and not on the border).



This illustrates how spritesheets work in CSS. All we need are just our coordinates and we are good to go. Overall we could distinguish between homogeneous spritesheets and heterogeneous spritesheets. While the first one does have a fixed grid (e.g. 32px times 32px for each element, resulting in easy offset calculations), the latter one does not have a fixed grid. The illustration shows a heterogeneous spritesheet. If we create a spritesheet for our homepage in order to increase performance by decreasing HTTP requests, we will usually end up with a heterogeneous spritesheet.

For animations we should stick to homogeneous spritesheets in order to decrease the amount of information necessary for the animation itself. The game uses the following piece of code to execute spritesheet animation:

Hide Shrink Copy Code

```
var Base = Class.extend({
    init: function(x, y) {
```

```
        this.setPosition(x || 0, y || 0);
        this.clearFrames();
    },
    /* more basic methods like setPosition(), ... */
    setupFrames: function(fps, frames, rewind, id) {
        if(id) {
            if(this.frameID === id)
                return true;

            this.frameID = id;
        }

        this.frameCount = 0;
        this.currentFrame = 0;
        this.frameTick = frames ? (1000 / fps / constants.interval) : 0;
        this.frames = frames;
        this.rewindFrames = rewind;
        return false;
    },
    clearFrames: function() {
        this.frameID = undefined;
        this.frames = 0;
        this.currentFrame = 0;
        this.frameTick = 0;
    },
    playFrame: function() {
        if(this.frameTick && this.view) {
            this.frameCount++;

            if(this.frameCount >= this.frameTick) {
                this.frameCount = 0;

                if(this.currentFrame === this.frames)
                    this.currentFrame = 0;

                var $el = this.view;
                $el.css('background-position', '-' + (this.image.x + this.width *
                  ((this.rewindFrames ? this.frames - 1 : 0) - this.currentFrame)) +
                  'px -' + this.image.y + 'px');
                this.currentFrame++;
            }
        }
    },
});
```

We included the spritesheet functionality in the most basic (game) class, since every more specialized class like figures or items will inherit from this class. This guarantees the availability of spritesheet animation. Basically every object has a function to setup the spritesheet animation called `setupFrames()`. The only parameters that need to be specified are the number of frames per second (`fps`) and the number of frames in the spritesheet (`frames`). Usually animation is executed from left to right - therefore we included the parameter `rewind` to change the animation to be from right to left.

One important thing about this method is the optional `id` parameter. Here we can assign a value that could identify the current animation. This can then be used in order to distinguish if the animation that is about to be set up is already running. If this is the case we will not reset the `frameCount` and other internal variables. How can this code be used within some class? Let's see with an example of the `Mario` class itself:

Hide Shrink Copy Code

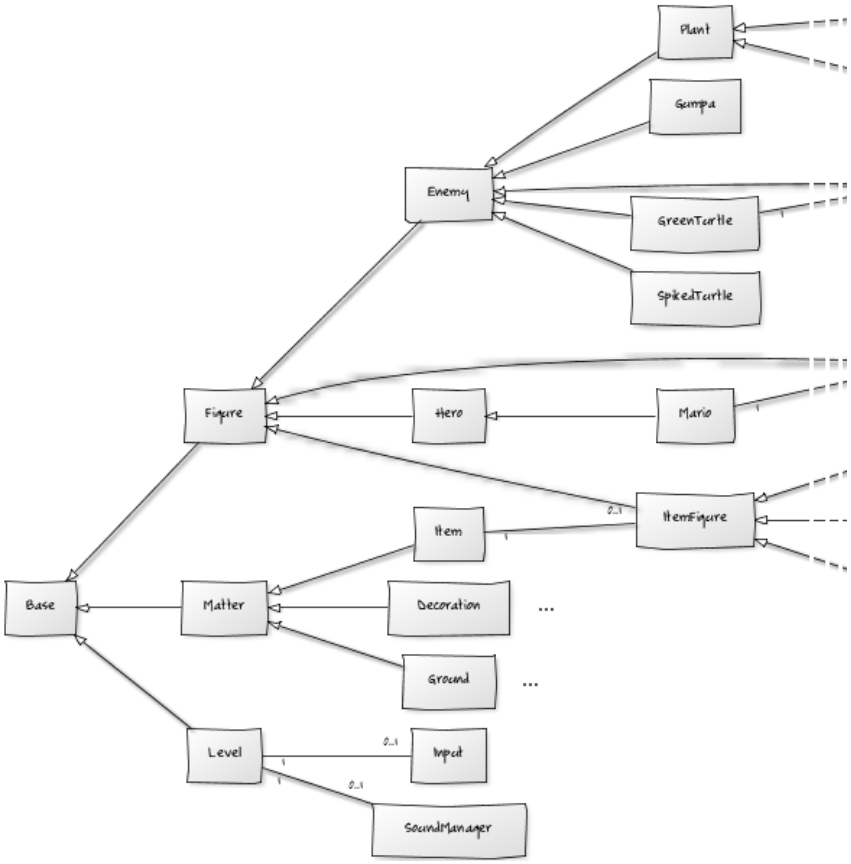
```
var Mario = Hero.extend({
    /*...*/
    setVelocity: function(vx, vy) {
        if(this.crouching) {
            vx = 0;
            this.crouch();
        } else {
            if(this.onground && vx > 0)
                this.walkRight();
            else if(this.onground && vx < 0)
                this.walkLeft();
            else
                this.stand();
        }

        this._super(vx, vy);
    },
    walkRight: function() {
        if(this.state === size_states.small) {
            if(!this.setupFrames(8, 2, true, 'WalkRightSmall'))
                this.setImage(images.sprites, 0, 0);
        } else {
            if(!this.setupFrames(9, 2, true, 'WalkRightBig'))
                this.setImage(images.sprites, 0, 243);
        }
    },
    walkLeft: function() {
        if(this.state === size_states.small) {
            if(!this.setupFrames(8, 2, false, 'WalkLeftSmall'))
                this.setImage(images.sprites, 81, 81);
        } else {
            if(!this.setupFrames(9, 2, false, 'WalkLeftBig'))
                this.setImage(images.sprites, 81, 162);
        }
```

```
    },
    /* ... */
});
```

Here we override the `setVelocity()` method. Depending on the current state we execute the corresponding function like `walkRight()` or `walkLeft()`. The function then looks at the current state in order to decide which animation to apply. Here we bring the optional `id` parameter into play. We change the current spritesheet location only then when we could apply a new animation. Otherwise the current animation seems still to be valid, resulting in a valid spritesheet location as well.

The class diagram

One of the purposes of rewriting the whole game was the incentive to describe everything in an object oriented manner. This will make the coding more interesting as well as simpler. Also the final game will contain less bugs. The following class diagram was planned before creating the game:



The game has been structured to show relations and dependencies. One of the benefits for such a structure is the ability to extend the game. We will investigate the extension process in the next section.

Inheritance is just one of the factors that writing object oriented JavaScript brings us. Another one is the ability to have something like types (instances of our classes). Therefore we can for instance ask if an object is an instance of a certain class. Let's have a look at the following code as an example:

Hide Copy Code

```
var Item = Matter.extend({
    /* Constructor and methods */
    bounce: function() {
        this.isBouncing = true;

        for(var i = this.level.figures.length; i--; ) {
            var fig = this.level.figures[i];

            if(fig.y === this.y + 32 && fig.x >= this.x - 16 && fig.x <= this.x + 16) {
                if(fig instanceof ItemFigure)
                    fig.setVelocity(fig.vx, constants.bounce);
                else if(fig instanceof Enemy)
                    fig.die();
            }
        }
    },
})
```

The code snippet shows part of the `Item` class. This class contains a new method `bounce()`, which let's the box go up and down a bit by setting the `isBouncing` property to `true`. Like in the original Mario game you can kill enemies which are unluckily standing on the bouncing item. The other popular case is to give instances of `ItemFigure` (like the mushroom) additional momentum in *y* direction (if above the bouncing item).

Extending the game

One thing that can always be included are new sprites (images) and movements. One example would be a proper suit for Mario in fire mode. The demo just uses the same images as the big Mario. The following image shows the fire suited Mario being victorious:

There are several extension points in the game itself. An obvious extension point is to build a new class and give it a proper reflection name. Levels can then use this name, which results in the level creating an instance of this class. We start with an easy example of implementing new kind of decoration: A left hanging bush!

Hide Copy Code

```
var LeftBush = Decoration.extend({
    init: function(x, y, level) {
        this._super(x, y, level);
        this.setImage(images.objects, 178, 928);
    },
}, 'bush_left');
```

This was pretty easy. We just have to inherit from the `Decoration` class and set another image over the `setImage()` method. Since decorations are non-blocking we cannot specify a blocking level here (as with classes inheriting from `Ground`). We name this new decoration class `bush_left`.

Now let's consider the case of extending the game with a new enemy: the ghost (not included in the source code)! This is a little bit harder, but not from the principle. The problems just come with the rules that this specific type of enemy has to follow. The basic construction is straight forward:

Hide Copy Code

```
var Ghost = Enemy.extend(
    init: function(x, y, level) {
        this._super(x, y, level);
        this.setSize(32, 32);
    },
    die: function() {
        //Do nothing here!
    },
});
```

So the first thing to notice here is that we do not call the `_super()` method in the `die()` method. This results in the `Ghost` not being able to die (since he or she is already dead). This is actually one of the rules. The other rules are:

- The ghost moves towards Mario (once it is able to see Mario)
- If Mario looks directly at the ghost (direction of Mario is the opposite of the ghost's direction) the ghost will not move
- Even if Mario has a star or shoots the ghost cannot die

While other routines just abuse the `setVelocity()` method it would be quite helpful to override the `move()` method in this case. There are two reasons for that:

- Gravity does not have any effect on the ghost
- The ghost just moves if certain rules (see above) are fulfilled

With this knowledge we can now include the rest of the ghost enemy, resulting in the following code:

Hide Shrink Copy Code

```
var Ghost = Enemy.extend({
    init: function(x, y, level) {
        this._super(x, y, level);
        this.setSize(33, 32);
        this.setMode(ghost_mode.sleep, directions.left);
    },
    die: function() {
            //Do nothing here!
        },
    setMode: function(mode, direction) {
        if(this.mode !== mode || this.direction !== direction) {
            this.mode = mode;
            this.direction = direction;
            this.setImage(images.ghost, 33 * (mode + direction - 1), 0);
        }
    },
    getMario: function() {
        for(var i = this.level.figures.length; i--; )
            if(this.level.figures[i] instanceof Mario)
                return this.level.figures[i];
    },
    move: function() {
        var mario = this.getMario();

        if(mario && Math.abs(this.x - mario.x) <= 800) {
            var dx = Math.sign(mario.x - this.x);
            var dy = Math.sign(mario.y - this.y) * 0.5;
            var direction = dx ? dx + 2 : this.direction;
            var mode = mario.direction === direction ? ghost_mode.awake : ghost_mode.sleep;
            this.setMode(mode, direction);

            if(mode)
                this.setPosition(this.x + dx, this.y + dy);
        } else
            this.setMode(ghost_mode.sleep, this.direction);
    },
    hit: function(opponent) {
        if(opponent instanceof Mario) {
            opponent.hurt(this);
        }
    },
}, 'ghost');
```

Here all of our rules have been applied. The ghost will only do a move if Mario is within a certain range (800 pixels in this case). In order to work coherent we introduce a new enumeration object, called `ghost_mode`:

Hide Copy Code

```
var ghost_mode = {
    sleep : 0,
    awake : 1,
};
```

We also need to introduce some new sprites. In this case we've just added a new image that includes all the sprites. The path is saved in `images.ghost` and leads to the following image:

Points of Interest

This article is named Super Mario for HTML5, however, there is not really much HTML5 in this demonstration. The two features that are provided by HTML5 are the `<canvas>` element and the `<audio>` element. Both have not been used for this demonstration. While the first one is interesting for the level editor (we will have a look at that in the next article), the latter is of course already interesting for the game as well. I decided to exclude it in this demonstration in order to keep the size of the source as small as possible.

Even though JavaScript is a dynamic language we can still use flag enumeration like variables. For instance the blocking variables are defined in a flag enumeration like way, e.g.:

Hide Copy Code

```
var ground_blocking = {
    none   : 0,
    left   : 1,
    top    : 2,
    right  : 4,
```

```
    bottom : 8,
    all    : 15,
};
```

Variables that contain values from `ground_blocking` by using `var blocking = ground_blocking.left + ground_blocking.top;` (could be also achieved by a bit operation, however, due to the statement written in JavaScript we would not have any benefits) or something similar can be read out easily. The process of reading out atomic values can be done by using:

Hide Copy Code

```
//e.g. check for top-blocking
function checkTopBlocking(blocking) {
    if((ground_blocking.top & blocking) === ground_blocking.top)
        return true;

    return false;
}
```

The original version (including sounds and code) can be found at http://www.florian-rappl.de/html5/projects/SuperMario/. This version will be also online soon.

History

- **v1.0.0** | Initial release | 01.06.2012.
- **v1.1.0** | Fixed some typos, including YouTube video, extended the ghost code | 05.06.2012.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Share

EMAIL

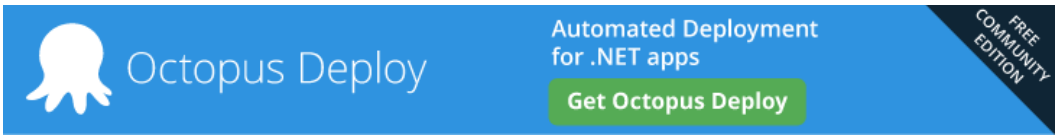
About the Author

Florian Rappl
Chief Technology Officer
Germany

Florian is from Regensburg, Germany. He started his programming career with Perl. After programming C/C++ for some years he discovered his favorite programming language C#. He did work at Siemens as a programmer until he decided to study Physics. During his studies he worked as an IT consultant for various companies.

Florian is also giving lectures in C#, HTML5 with CSS3 and JavaScript, and other topics. Having graduated from University with a Master's degree in theoretical physics he is currently busy doing his PhD in the field of High Performance Computing.

Follow on Google



Comments and Discussions

Add a Comment or Question

Search Comments Go

Profile popups Spacing Tight Noise Medium Layout Normal Per page 50 Update

First Prev Next

Woww... Super Article new	**Shemeemsha RA**	**6-Oct-14 22:56**
Re: Woww... Super Article new	Florian Rappl	7-Oct-14 13:38
碰撞检测 new	**queenking**	**18-Apr-14 1:12**
Re: 碰撞检测 new	Yang Kok Wah	18-Jul-14 1:27
Re: 碰撞检测 new	Florian Rappl	18-Jul-14 13:54
Life new	**Member 9968865**	**23-Jul-13 1:14**
Re: Life new	Florian Rappl	23-Jul-13 4:23
Re: Life new	Member 9968865	23-Jul-13 5:00
sound? new	**Member 9968865**	**16-Jul-13 2:26**
Re: sound? new	Florian Rappl	16-Jul-13 8:37
Re: sound? [modified] new	Member 9968865	16-Jul-13 23:46
Re: sound? new	Chris Ward	12-Nov-14 7:33
Re: sound? new	Florian Rappl	12-Nov-14 8:44
Re: sound? new	Chris Ward	12-Nov-14 8:55
Re: sound? new	Florian Rappl	12-Nov-14 9:14
Re: sound? new	Chris Ward	17-Nov-14 6:31
Re: sound? new	Florian Rappl	17-Nov-14 6:54
My vote of 5 new	**Curtdawg99**	**1-Jul-13 15:15**
Re: My vote of 5 new	Florian Rappl	1-Jul-13 22:34
My vote of 5 new	**Martin Arapovic**	**24-Apr-13 7:39**
Re: My vote of 5 new	Florian Rappl	24-Apr-13 10:13
Beautiful code, well done new	**Member 9938708**	**25-Mar-13 20:24**
Re: Beautiful code, well done new	Florian Rappl	26-Mar-13 1:05
Excellent article! Thank you! new	**teknedugunu**	**1-Mar-13 15:06**
Re: Excellent article! Thank you! new	Florian Rappl	1-Mar-13 19:49
My vote of 5 new	**neoraltech**	**23-Feb-13 0:07**
Re: My vote of 5 new	Florian Rappl	24-Feb-13 9:05
My vote of 5 new	**Anand1008**	**5-Feb-13 1:20**
Re: My vote of 5 new	Florian Rappl	11-Feb-13 1:42
My vote of 5 new	**wesley_jr40**	**21-Jan-13 8:38**
Re: My vote of 5 new	Florian Rappl	27-Jan-13 3:28
My vote of 5 new	**Ahmed Ibrahim Assaf**	**18-Dec-12 23:47**
Re: My vote of 5 new	Florian Rappl	19-Dec-12 0:29
My vote of 5 new	**ALIJavany**	**29-Nov-12 8:23**
Re: My vote of 5 new	Florian Rappl	29-Nov-12 8:28
Oh My GOD new	**ALIJavany**	**29-Nov-12 8:21**
Re: Oh My GOD new	Florian Rappl	29-Nov-12 8:28
My vote of 5 new	**DotNetMastermind**	**24-Nov-12 16:00**
Re: My vote of 5 new	Florian Rappl	25-Nov-12 2:10
Great! new	**Alex Loizou**	**2-Nov-12 15:37**
Re: Great! new	Florian Rappl	3-Nov-12 4:56
Very nice program new	**Ommrudraksha Shri**	**29-Oct-12 2:13**
Re: Very nice program new	Florian Rappl	29-Oct-12 2:57
Fantastic new	**Steve_Hardeman**	**28-Sep-12 13:49**
Re: Fantastic new	Florian Rappl	28-Sep-12 14:31
thank new	**juuknet**	**11-Sep-12 12:16**
Re: thank new	Florian Rappl	11-Sep-12 15:25
My vote of 5 new	**Aamer Alduais**	**4-Aug-12 0:31**
Re: My vote of 5 new	Florian Rappl	13-Aug-12 4:55

My vote of 5 new newton.saber 3-Aug-12 12:01

Last Visit: 31-Dec-99 21:00 Last Update: 13-Mar-15 16:06 Refresh 1 2 3 4 Next »

General News Suggestion Question Bug Answer Joke Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink | Advertise | Privacy | Terms of Use | Mobile
Web01 | 2.8.150310.1 | Last Updated 5 Jun 2012

Select Language

Layout: fixed | fluid

Article Copyright 2012 by Florian Rappl
Everything else Copyright © CodeProject, 1999-2015