Quintus

Juan Antonio Recio García (jareciog@fdi.ucms)
Guillermo Jiménez Díaz (gjimenez@ucm.es)
Apuntes creados por Pedro Antonio González Calero (pedro@fdi.ucm.es)

Introducciòn a Quintus

Un juego completo en 80 líneas de código

El juego está en la página de Quintus

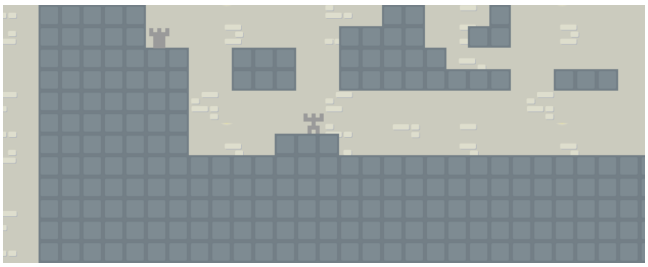

Figure 1: Un plataformas en Quintus

Una página HTML que lo carga:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Un juego en 80 líneas</title>
    <script src='lib/quintus.js'></script>
    <script src='lib/quintus_sprites.js'></script>
    <script src='lib/quintus_scenes.js'></script>
    <script src='lib/quintus_2d.js'></script>
    <script src='lib/quintus_input.js'></script>
    <script src='lib/quintus_ui.js'></script>
    <script src='lib/quintus_touch.js'></script>
    <script src='lib/quintus_anim.js'></script>
    <script src='game.js'></script>
    <style>
      canvas { background-color:white; }
    </style>
  </head>
  <body>
  <script type="text/javascript">
    window.onload = game;
  </script>
  </body>
</html>
```

y en el fichero `game.js` se incluye el código del juego, donde se define la función `game` que es la que hemos asociado con el evento `window.onload` y que será la

encargada de configurar una instancia del motor, cargar los recursos y lanzar el juego.

Se inicializa el motor

```
var game = function() {

// Set up an instance of the Quintus engine  and include
// the Sprites, Scenes, Input and 2D module. The 2D module
// includes the `TileLayer` class as well as the `2d` componet.
var Q = window.Q = Quintus()
        .include("Sprites, Scenes, Input, 2D, Anim, Touch, UI")
        // Maximize this game to whatever the size of the browser is
        .setup({ maximize: true })
        // And turn on default input controls and touch input (for UI)
        .controls().touch()
```

Se definen las entidades

```
// ## Player Sprite
// The very basic player sprite, this is just a normal sprite
// using the player sprite sheet with default controls added to it.
Q.Sprite.extend("Player",{

  // the init constructor is called on creation
  init: function(p) {

    // You can call the parent's constructor with this._super(..)
    this._super(p, {
      sheet: "player",  // Setting a sprite sheet sets sprite width and height
      x: 410,           // You can also set additional properties that can
      y: 90             // be overridden on object creation
    });

    // Add in pre-made components to get up and running quickly
    // The `2d` component adds in default 2d collision detection
    // and kinetics (velocity, gravity)
    // The `platformerControls` makes the player controllable by the
    // default input actions (left, right to move,  up or action to jump)
    // It also checks to make sure the player is on a horizontal surface before
    // letting them jump.
    this.add('2d, platformerControls');

    // Write event handlers to respond hook into behaviors.
```

```
    // hit.sprite is called everytime the player collides with a sprite
    this.on("hit.sprite",function(collision) {

      // Check the collision, if it's the Tower, you win!
      if(collision.obj.isA("Tower")) {
        Q.stageScene("endGame",1, { label: "You Won!" });
        this.destroy();
      }
    });

  }

});


// ## Tower Sprite
// Sprites can be simple, the Tower sprite just sets a custom sprite sheet
Q.Sprite.extend("Tower", {
  init: function(p) {
    this._super(p, { sheet: 'tower' });
  }
});

// ## Enemy Sprite
// Create the Enemy class to add in some baddies
Q.Sprite.extend("Enemy",{
  init: function(p) {
    this._super(p, { sheet: 'enemy', vx: 100 });

    // Enemies use the Bounce AI to change direction
    // whenver they run into something.
    this.add('2d, aiBounce');

    // Listen for a sprite collision, if it's the player,
    // end the game unless the enemy is hit on top
    this.on("bump.left,bump.right,bump.bottom",function(collision) {
      if(collision.obj.isA("Player")) {
        Q.stageScene("endGame",1, { label: "You Died" });
        collision.obj.destroy();
      }
    });

    // If the enemy gets hit on the top, destroy it
    // and give the user a "hop"
    this.on("bump.top",function(collision) {
      if(collision.obj.isA("Player")) {
```

```
        this.destroy();
        collision.obj.p.vy = -300;
      }
    });
  }
});
```

Se definen las escenas

El nivel se genera a partir de los recursos gráficos que se muestran en la figura 2

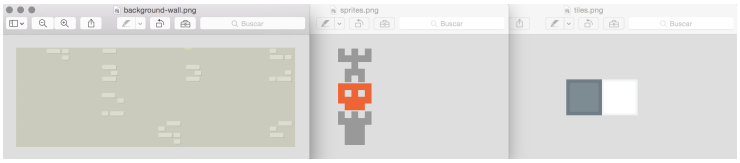

Figure 2: Recursos gráficos

y la información que se almacena en el fichero `level.json`:

```
[
[ 1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,1,1,1,1,1,1,1,0,0,1,1,1,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,1,1,1,1,1,1,1,0,0,1,1,1,0,0,1,1,1,1,1,1,1,1,1,0,0,1,1,1,0,0,1,0,0,0,1,1,1],
[ 1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1],
[ 1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,1],
[ 1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,1,1],
[ 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
...
]
```

y el fichero `sprites.json`:

```
{
  "player":
    {"sx":0,"sy":0,"cols":1,"tilew":30,"tileh":30,"frames":1},

  "enemy":
    {"sx":0,"sy":31,"cols":1,"tilew":30,"tileh":24,"frames":1},

  "tower":
    {"sx":0,"sy":56,"cols":1,"tilew":30,"tileh":30,"frames":1}
}
```

Este es el código que se encarga de configurar las escenas:

```
// ## Level1 scene
// Create a new scene called level 1
Q.scene("level1",function(stage) {

  // Add in a repeater for a little parallax action
  stage.insert(new Q.Repeater({ asset: "background-wall.png", speedX: 0.5, speedY: 0.5 }));

  // Add in a tile layer, and make it the collision layer
  stage.collisionLayer(new Q.TileLayer({
                             dataAsset: 'level.json',
                             sheet:     'tiles' }));

  // Create the player and add them to the stage
  var player = stage.insert(new Q.Player());

  // Give the stage a moveable viewport and tell it
  // to follow the player.
  stage.add("viewport").follow(player);

  // Add in a couple of enemies
  stage.insert(new Q.Enemy({ x: 700, y: 0 }));
  stage.insert(new Q.Enemy({ x: 800, y: 0 }));

  // Finally add in the tower goal
  stage.insert(new Q.Tower({ x: 180, y: 50 }));
});

// To display a game over / game won popup box,
// create a endGame scene that takes in a `label` option
// to control the displayed message.
Q.scene('endGame',function(stage) {
  var container = stage.insert(new Q.UI.Container({
    x: Q.width/2, y: Q.height/2, fill: "rgba(0,0,0,0.5)"
  }));

  var button = container.insert(new Q.UI.Button({ x: 0, y: 0, fill: "#CCCCCC",
                                                  label: "Play Again" }))
  var label = container.insert(new Q.UI.Text({x:10, y: -10 - button.p.h,
                                               label: stage.options.label }));
  // When the button is clicked, clear all the stages
  // and restart the game.
  button.on("click",function() {
    Q.clearStages();
    Q.stageScene('level1');
  });
```

```
  // Expand the container to visibily fit it's contents
  // (with a padding of 20 pixels)
  container.fit(20);
});
```

Carga de recursos e inicio del juego

```
// ## Asset Loading and Game Launch
// Q.load can be called at any time to load additional assets
// assets that are already loaded will be skipped
// The callback will be triggered when everything is loaded
Q.load("sprites.png, sprites.json, level.json, tiles.png, background-wall.png", function() {
  // Sprites sheets can be created manually
  Q.sheet("tiles","tiles.png", { tilew: 32, tileh: 32 });

  // Or from a .json asset that defines sprite locations
  Q.compileSheets("sprites.png","sprites.json");

  // Finally, call stageScene to run the game
  Q.stageScene("level1");
});

}; // var game = ...
```

Mecanismos básicos

Uso

Lo más fácil es incluirlo de su CDN (*content delivery network*):

```
<!-- Production minified ~20k gzipped -->
<script src='http://cdn.html5quintus.com/v0.2.0/quintus-all.min.js'></script>

<!-- Full Source ~40k gzipped -->
<script src='http://cdn.html5quintus.com/v0.2.0/quintus-all.js'></script>
```

> Minification: in computer programming languages and especially JavaScript, is the process of removing all unnecessary characters from source code without changing its functionality.

o también se puede descargar de GitHub y compilarlo con npm (node packaged modules), el gestor de paquetes de Node.js, que permite ejecutar JavaScript en el servidor, y generar los ficheros con grunt (una especie de make para JavaScript):

```
$ git clone git://github.com/cykod/Quintus.git
$ cd Quintus
```

```
$ npm install
$ grunt
```

o bien usar la versión que incluye Guillermo en el enunciado de las prácticas ...

Instanciar el motor

Para crear una instancia del motor se utiliza el constructor `Quintus()`

```
var Q = Quintus();
```

durante el desarrollo es buena idea activar el modo de desarrollo que se encarga de recargar todo el contenido con cada recarga evitando así los problemas de caché:

```
var Q = Quintus({ development: true });
```

La funcionalidad en Quintus se organiza en *módulos* que cargamos opcionalmente al construir la instancia del motor:

```
var Q = Quintus().include("Sprites, Scenes, Input");
```

Podemos definir nuestros propios módulos en Quintus sin más que añadir un método que recibe una instancia del motor como parámetro y le asigna atributos, así por ejemplo podemos añadir un módulo `Random` que incluya una función `random(min,max)`:

```
Quintus.Random = function(Q) {

  Q.random = function(min,max) {
    return min + Math.random() * (max - min);
  }

};
```

Enlazar con un canvas

El método `Q.setup([id],[options={}])` es el responsable de vincular la instancia del motor con un elemento de tipo canvas en la página y configurar su tamaño. Si no especificamos nada, se crea un canvas de 320 * 420 píxeles:

```
var Q = Quintus().setup();
```

Los métodos de configuración de la instancia del motor devuelven la propia instancia por lo que es posible encadenar mensajes de configuración:

```
var Q = Quintus().include("Sprites, Scenes, Input")
                 .setup();
```

Si ya hay un elemento canvas en la página entonces podemos vincularlo a la instancia de Quintus pasándole su id:

```
<canvas id='myGame' width='500' height='500'></canvas>

< script >
  var Q = Quintus().setup("myGame");
</script>
```

También se puede especificar que ocupe todo el espacio de la ventana:

```
// Always maximize
var Q = Quintus().setup({ maximize: true });
```

o sólo en dispositivos móviles:

```
// Maximize only on touch devices
var Q2 = Quintus().setup({ maximize: "touch" });
```

y existen además parámetros para controlar las dimensiones por defecto y la resolución:

```
var Q = Quintus.setup({
  width:   800, // Set the default width to 800 pixels
  height:  600, // Set the default height to 600 pixels
  upsampleWidth:  420,  // Double the pixel density of the
  upsampleHeight: 320,  // game if the w or h is 420x320
                        // or smaller (useful for retina phones)
  downsampleWidth: 1024, // Halve the pixel density if resolution
  downsampleHeight: 768  // is larger than or equal to 1024x768
});
```

El contexto del canvas

El contexto del canvas está disponible en la variable `Q.ctx` aunque, a no ser que queramos definir nuestro propio bucle de dibujado, no es necesario acceder a `Q.ctx` directamente porque ya se se pasa como parámetro a los métodos `draw` de los `Sprite`.

Además, si se usa la gestión de escenas de Quintus (módulo `Scenes`), no es buena idea dibujar directamente en el contexto, por ejemplo en un método `step`, porque su contenido se borra cada vez antes de llamar a los métodos `draw`.

Creación de clases

Quintus incluye una implementación de clases con herencia basada en el esquema de herencia simple en JavaScript propuesto por John Resig (http://ejohn.org/blog/simple-javascript-inheritance/).

Las clases heredan de `Q.Class` y se crean subclases son el método `extend`: `Q.Class.extend(ClassName, { .. methods .. }, { .. class methods .. })`. El constructor es el método `init`:

```
Q.Class.extend("MyClass", {
   init: function() { console.log("MyClass instance created"); },
   doIt: function() { alert("Doin it!"); }
});

var myInstance = new Q.MyClass(); // MyClass instance created

myInstance.doIt(); // Doin it!

console.log(myInstance.className); // 'MyClass'
console.log(myInstance instanceof Q.Class); // true
console.log(myInstance instanceof Q.MyClass); // true
```

Se puede redefinir un método definido en una superclase, y es posible invocar la versión redefinida con `this._super(..)`.

Carga de recursos

Quintus incluye dos métodos para cargar recursos, `Q.load` y `Q.preload`.

A `Q.load` se le pasa un array con los nombres de los recursos a cargar y una función que se ejecutará cuando todos hayan sido cargados:

```
Q.load([ "sprites.png", "sprites.json", "music.ogg" ], function() {
  // Start your game
});
```

Quintus define una estructura de directorios por defecto para los recursos: las imágenes se buscan en "images/" (debajo del directorio donde esté el fichero HTML que se ha cargado inicialmente); el audio se busca en "audio/" y cualquier otra cosa en "data/". Si se quieren usar rutas diferentes basta con especificarlo en la creación de la instancia del motor, por ejemplo, para cargarlo todo de "http://cdn.yourgame.com/assets/":

```
var Q = Quintus({ imagePath: "http://cdn.yourgame.com/assets/",
                  audioPath: "http://cdn.yourgame.com/assets/",
                  dataPath: "http://cdn.yourgame.com/assets/" });
```

`Q.preload` permite especificar sucesivas cargas y finalmente una función que será llamada cuando todas las cargas estén hechas:

```
Q.preload("sprites.png");
Q.preload([ "music1.ogg", "music2.ogg" ]);

Q.preload(function() {
```

```
  // Go time
});
```

Sprites

Los Sprites son los elementos centrales para el desarrollo con Quintus y se suelen usar en combinación con los escenarios (*stages*). Para usarlos hay que incluir el módulo `Sprites`.

La clase base de los sprites es `Sprite`, que hereda de `GameObject` (que a su vez hereda de `Evented` y este de la raíz de la jerarquía `Class`) por lo que ya lleva incorporado el soporte para eventos y componentes. El comportamiento de un sprite se establece a través de tres métodos fundamentales: `init(p,defaults)`, `step(dt)` y `draw(ctx)`.

Inicialización de sprites

La inicialización se hace a través del método `init(p,defaults)` que recibe un objeto con los atributos a asignar al sprite que se está creando y los mezcla con las propiedades de otro objeto que contiene valores por defecto. Los datos de un sprite se guardan en el atributo `p` del sprite. Estos atributos son públicos para hacer más eficiente su modificación y consulta (sin necesidad de usar accedentes y mutadores).

Por ejemplo:

```
Q.Sprite.extend("Player",{
  init: function(p) {
    this._super(p, {
        hitPoints: 10,
        damage: 5,
        x: 5,
        y: 1
    });
});
```

Ahora podemos crear instancias con los atributos por defecto, o especificar un valor diferente para un atributo concreto:

```
var player1 = new Q.Player();

console.log(player1.p.hitPoints); // 10
console.log(player1.p.damage); // 5

var player2 = new Q.Player({ hitPoints: 20 });
```

```
console.log(player2.p.hitPoints); // 20
console.log(player2.p.damage); // 5
```

Propiedades de los sprites

Las propiedades predefinidas de los sprites soportan el dibujado y la detección de colisiones. La detección de colisiones se lleva a cabo en el módulo Scenes, pero utiliza las propiedades de los sprites:

- p.x - coordenada x del centro del sprite. Por defecto 0
- p.y - coordenada y del centro del sprite. Por defecto 0
- p.z - orden en el que se apilan los sprites en el eje z, a mayor valor más arriba (lo utiliza el módulo Scenes). Por defecto 0
- p.sort: si su valor es true entonces fuerza a que los hijos de este sprite se dibujen de acuerdo con su valor de p.z. Por defecto false
- p.w - ancho del sprite. Su valor se toma de p.asset o p.sheet
- p.h - alto del sprite. Su valor se toma de p.asset o p.sheet
- p.cx - distancia desde el centro al extremo izquierdo del sprite. Por defecto w/2
- p.cy - distancia desde el centro a la parte superior del sprite. Por defecto h/2
- p.points - un array de puntos que definen una figura de colisión convexa con los puntos en sentido horario. Por defecto, es un cuadro delimitador en forma de paralelepípedo: [[-w/2, -h/2], [w/2, -h/2], [w/2, h/2], [-w/2, h/2]]
- p.opacity - opacidad con la que se dibuja la imagen del sprite. Por defecto 1
- p.scale - la escala con la que se dibuja el sprite. Por defecto 1
- p.angle - el ángulo de rotación del sprite en grados. Por defecto 0
- p.type - la máscara de bits que se usa para clasificar al sprite en la detección de colisiones. Por defecto Q.SPRITE_DEFAULT | Q.SPRITE_ACTIVE
- p.asset - el nombre del asset que contiene la imagen que pinta el sprite. Por defecto null
- p.sheet - el sprite sheet que contiene la(s) imagen(es) que pinta el sprite. Si este está definido entonces no se utiliza el valor de p.asset. Por defecto null
- p.frame - si el sprite pinta una imagen de un sprite sheet entonces este atributo contiene el número de fotograma. Por defecto 0
- p.name - un identificador para el sprite. Por defecto “”
- p.hidden - si está a true entonces se esconde el sprite. Por defecto false
- p.flip - Puede tomar los valores “x”, “y”, or “xy” para voltear el sprite según el eje indicado. Por defecto “”

Estas propiedades se pueden modificar directamente o se puede utilizar el método `set` que además permite modificar varias propiedades al tiempo con `Sprite.set({ ... })` (aunque esto es más ineficiente porque se crea un objeto

que luego será basura y tendrá que ser recolectado)

Los Sprites recalculan sus matrices de traslación y su cuadros delimitadores en cada frame si es necesario.

Asociar una imagen con un sprite

A un sprite se le puede asociar una imagen a través del atributo `asset` que se debe fijar en el método init. El sprite obtendrá de la imagen la información sobre su alto y ancho además de las coordenadas del centro.

Es importante que la imagen se haya cargado antes de intentar pintar el sprite por primera vez:

```
var Q = Quintus().include("Sprites").setup();

Q.Sprite.extend("Penguin", {
  init: function(p) {
    this._super({
      asset: "penguin.png"
    });
  }
});

// Make sure penguin.png is loaded
Q.load("penguin.png",function() {
   var penguin = new Q.Penguin();

   Q.gameLoop(function(dt) {
     Q.clear();
     penguin.update(dt);
     penguin.render(Q.ctx);
   });
 });
```

Creación de Sprite sheets

Es habitual que un mismo archivo (sprite sheet) contenga varias imágenes (sprites) del juego, en el módulo `Sprites` se incluye también una clase `Spritesheet` que gestiona la información sobre una imagen dentro de un sprite sheet. Para crear una imagen a partir de un sprite sheet se utiliza el método `Q.sheet`:

```
Q.sheet("player",
        "player.png",
        {
          tilew: 40,  // Each tile is 40 pixels wide
```

```
    tileh: 40,  // and 40 pixels tall
    sx: 0,   // start the sprites at x=0
    sy: 0    // and y=0
  });
```

que en este ejemplo hace que a partir de este momento esté disponible la imagen `Q.sheet("player")`

Como generar los datos de las imágenes de un sprite es un proceso tedioso y donde es fácil equivocarse, es mejor idea utilizar una herramienta que lo haga por nosotros. Quintus permite cargar los datos de un sprite sheet a partir de un fichero JSON con esta estructura (básicamente la misma de la función `Q.sheet`):

```
{
  "snail":{"sx":0,"sy":32,"cols":1,"tilew":72,"tileh":42,"frames":3,"rows":3,"spacingY":32},
  "slime":{"sx":72,"sy":32,"cols":1,"tilew":72,"tileh":42,"frames":3,"rows":3,"spacingY":32},
  "fly":{"sx":144,"sy":32,"cols":1,"tilew":72,"tileh":42,"frames":3,"rows":3,"spacingY":32}
}
```

que corresponde a la información de la imagen 3

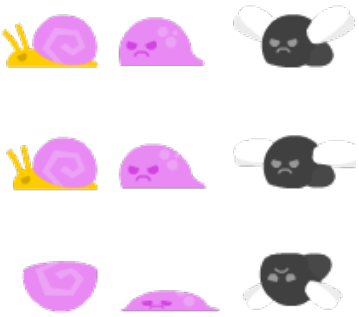

Figure 3: enemies.png

Con la imagen y archivo JSON como este, se puede indicar a Quintus que los cargue y los procese de la siguiente manera:

```
Q.load(["enemies.png","enemies.json"], function() {

  // this will create the sprite sheets snail, slime and fly
  Q.compileSheets("enemies.png","enemies.json");
}
```

Y el fichero JSON con los datos y el sprite sheet con las imágenes agrupadas se pueden generar utilizando la herramienta de código abierto cykod/Spriter (https://github.com/cykod/Spriter) a partir de las imágenes contenidas en un directorio y que se hayan nombrado siguiendo el patrón `nombreXXX.png` (o .jpg)

```
assets/snail01.png
assets/snail02.png
...
assets/slime01.png
assets/slime02.png
...

...
```

Uso de sprite sheets

Para asociar una imagen de un sprite sheet con un sprite, se utiliza la propiedad `sheet` de la clase Sprite en lugar de la propiedad `asset`. Además se puede especificar el número de fotograma que se quiere usar con la propiedad `frame`. Por ejemplo, podemos asociar con un sprite, el octavo fotograma del sprite sheet "player":

```
Q.Sprite.extend("Penguin", {
     init: function(p) {
       this._super({
         sheet: "player",
         frame: 7
       });
     }
   });
```

La clase Sprite también es capaz de inicializar el ancho, alto y centro del nuevo sprite a partir de la información del sprite sheet.

También es posible dibujar una imagen de un sprite sheet directamente:

```
Q.sheet("spriteOne").draw(ctx, x, y, frameNum);
```

Escenas

El módulo `Scenes` introduce dos clases fundamentales: `Scene` (escena) y `Stage` (escenario). Una escena básicamente contiene instrucciones para configurar un escenario, de forma que es posible asignarle un nombre a una configuración de escenario (una escena) y reutilizarla en distintos lugares del juego. Las escenas, además de un nombre, sólo tienen dos propiedades: una retrollamada que añade elementos a un escenario y, opcionalmente, un objeto con opciones que controlan el comportamiento del escenario. Las escenas se crean con el método `Q.scene`:

```
Q.scene("level1",function(stage) {
  stage.insert(new Q.Ball());
});
```

Escenarios

Los escenarios (instancias de `Stage`) son capas de entidades del juego. Los escenarios se dibujan siempre en el mismo orden, empezando por el de nivel 0, luego el 1 y así sucesivamente. Aunque las entidades de distintas capas pueden interactuar a través de mensajes que se generan en unas capas y se consumen en otras, las capas son independientes por lo que se refiere a la detección de colisiones: las entidades de una capa no pueden colisionar con las entidades de otra.

Para añadir una escena a un escenario se utiliza el método `Q.stageScene`, que por defecto añade la escena a la capa 0, sustituyendo la escena que pudiese haber ahí antes. Normalmente la capa 0 se utiliza para el juego propiamente dicho, mientras que las capas superiores se utilizan para el HUD y las pantallas de la interfaz de usuario.

```
Q.scene("level1",function(stage) {
  stage.insert(new Q.Ball());
});

Q.stageScene("level1");
```

`stageScene` al insertar la escena en el escenario ejecuta la retrollamada de la escena que se encarga de configurar el escenario en el que está siendo insertada.

La clase `Stage` hereda de `GameObject`, por lo que puede tener componentes y lanzar y responder a eventos.

Además del nombre de la escena, a `stageScene` le podemos pasar la capa donde queremos que se añada:

```
Q.stageScene("level1",1);
```

y un objeto con atributos que establecen opciones del escenario, que se almacenan en el atributo `options` del escenario y que se pueden utilizar en la retrollamada de la escena:

```
Q.scene("endGame",function(stage) {
  var label = stage.insert(new Q.UI.Text({
    x: Q.width/2,
    y: Q.height/2,
    label: stage.options.label
  }));
});

// Stage a scene on stage 1 and pass in a label
Q.stageScene("endGame",1, {
  label: "This is the label"
});
```

El método `stage` permite acceder al objeto escenario. Si lo invocamos sin parámetros, devolverá el escenario activo que por defecto es el 0, pero si lo invocamos desde el método `step` de un sprite entonces será el escenario donde ese sprite esté incluido:

```
Q.stage(); // returns the active stage
Q.stage(0); // always returns stage 0
Q.stage(1); // always return stage 1
```

Jerarquías de sprites

Como ya hemos visto en la retrollamada de las escenas, las entidades se insertan en un escenario con el método `insert`, que además devuelve una referencia al sprite que insertamos:

```
var ball = stage.insert(new Q.Ball());
```

El método `insert` admite un segundo parámetro que permite definir *jerarquías* de sprites. Podemos especificar que un sprite es *hijo* de otro, lo que implica que se dibujará siempre encima de su padre. Esto se utiliza habitualmente en la interfaz de usuario o cuando tenemos partes de un sprite que queremos cambiar dinámicamente. Por ejemplo, podemos tener una nave sobre la que podamos seleccionar distintos tipos de cañones que se deben pintar por encima de la nave. Por ejemplo:

```
var car = stage.insert(new Q.Car());
var wheel1 = stage.insert(new Q.Wheel(), car);
```

También es posible eliminar todas las entidades de un escenario, con `Q.clearStage(num)`, o de todos los escenarios con `Q.clearStages()`.

Pausa de un escenario

Con los métodos `pause` y `unpause` podemos detener y reanudar la actualización de la lógica de un escenario (`step`) sin detener el pintado:

```
// Pause the stage, Sprite will no longer be stepped
// but will render each frame
Q.stage().pause();

// Unpause the stage
Q.stage().unpause();
```

Esto es útil cuando queremos mostrar una ventana con información al usuario, deteniendo el juego mientras tanto.

Acceso a los sprites de una escena

El método `Stage.locate` permite localizar el sprite que está en unas coordenadas dadas, pudiéndose especificar también una máscara de colisión opcionalmente:

```
// Find any object that collides with the point 50,50
var obj = Q.stage().locate(50,50);

// Find any enemies that collide with the point 50,50
var obj2 = Q.stage().locate(50,50,Q.ENEMY_TYPE);
```

Aunque en general no es muy útil identificar un sprite por sus coordenadas exactas:

```
Q.Sprite.extend("Manager", {
  init: function(p) {
    this._super(p);
    Q.input.on("fire", this, "fire");
  },

  fire: function() {
    var obj = Q.stage().locate(150,464);
    obj.p.hidden = ! obj.p.hidden;
  },
});
```

Los métodos `Stage.invoke` y `Stage.detect` son funciones de orden superior que permiten aplicar una función dada a todos los sprites de un escenario, `invoke`, o recorrerlos hasta encontrar el primero que hace cierta la función, `detect`:

```
// Call methodName on every sprite in the stage
Q.stage().invoke("methodName",arg1,arg2);

// Return the first sprite for which method returns truthy
var sprite = Q.stage().detect(method);
```

Selectores

También es posible acceder a todos los sprites de un cierto tipo, o los que contienen un componente determinado:

```
var balls = Q("Ball"); // returns all instances of Q.Ball

var two_d = Q(".2d"); // return all sprites with the 2d component
```

Las entidades se seleccionan del escenario activo, pero también podemos indicarlos explícitamente:

```
var balls = Q("Ball",0); // stage 0

var two_d = Q(".2d", 1); // stage one.
```

y también podemos acceder al que ocupa una determinada posición dentro de la colección de entidades de un cierto tipo:

```
var ball = Q("Ball").first();

var ball2 = Q("Ball").last();

var ball3 = Q("Ball").at(4);
```

que devuelve `null` si no hay ningún objeto en la posición especificada.

Es importante señalar que estas operaciones son eficientes porque Quintus mantiene listas por tipo de Sprite y no se requiere hacer una búsqueda exhaustiva entre todos los sprites de un escenario.

Lo que devuelven las llamadas de la forma `Q("selector")` son en realidad instancias de la clase `Q.StageSelector`, que ofrece métodos que nos permiten iterar sobre la colección de entidades, para, por ejemplo, establecer uno o más atributos en todas ellas:

```
// Set y=50 on all Q.Ball's
Q("Ball").set("y",50);

// Set y=50 and vy=0 on all Q.Ball's on the active stage
Q("Ball").set({ y: 50, vy: 0 });
```

También es posible lanzar un evento sobre todos ellos, invocar un método o aplicar una función dada:

```
// Return all instances of Q.Ball
var balls = Q("Ball");

// Trigger a "shine" event on each ball
balls.trigger("shine");

// Call the doSomething() method on each ball
balls.invoke("doSomething");

// If you need more complicated functionality, use each
balls.each(function() {
  if(this.testSomething()) {
    this.doSomething();
  }
});
```

```
// Finally, get rid of all the balls
balls.destroy();
```

por ejemplo, podemos mostrar y ocultar alternativamente los Marios de nuestro juego:

```
Q.Sprite.extend("Manager", {
  init: function(p) {
    this._super(p);
    Q.input.on("fire", this, "fire");
  },

  fire: function() {
    var marios = Q("Mario");
    marios.each( function () {
      this.p.hidden = ! this.p.hidden;
    })
  },
});
```

El módulo `2D`

El módulo `2D` añade una serie de recursos de alto nivel que facilitan la construcción de juegos en 2D: ventana de visualización (*viewport*), fondos basados en baldosas y gestión del movimiento y de las colisiones en 2D.

Ventana de visualización

El componente `viewport` se añade al escenario y define una cámara que muestra una parte de dicho escenario, normalmente centrada en la posición del personaje que controla el jugador. Este es un mecanismo típico en juegos de scroll lateral o vertical.

Una vez añadido el componente al escenario se le envía el mensaje `follow` pasándole el sprite al que debe seguir la cámara:

```
Q.scene("myLevel", function(stage) {
  var player = stage.insert(new Player());

  stage.add("viewport")
       .follow(player);
});
```

También se puede especificar que la cámara se mueva sólo en uno de los ejes:

```
stage.add("viewport")
     .follow(player,{ x: true, y: false });
```

El componente `viewport` añade también algunos métodos al escenario:

- `stage.unfollow()` - la ventana de visualización deja de moverse
- `stage.centerOn(x,y)` - centra la ventana de visualización en un posición dada (sólo es útil cuando no se está siguiendo a un sprite)
- `stage.moveTo(x,y)` - mueve el vértice superior izquierdo de la ventana de visualización a las coordenadas indicadas (sólo es útil cuando no se está siguiendo a un sprite)

Por último, hay ciertas propiedades de la ventana de visualización que se pueden establecer directamente:

- `stage.viewport.scale` - un factor de escala que se aplica a la zona visualizada
- `stage.viewport.offsetX` - desplazamiento en el eje X al centrar en una posición o seguir a un sprite
- `stage.viewport.offsetY` - desplazamiento en el eje Y al centrar en una posición o seguir a un sprite

Fondos basados en baldosas

Usar escenarios creados a partir de baldosas es un recurso muy habitual en videojuegos. Quintus define la clase `TileLayer` en el módulo `2D` para este fin.

Para crear un fondo con baldosas, se instancia la clase `TileLayer` y el resultado se inserta en un escenario. Un `TileLayer` necesita como mínimo

- un atributo `sheet` con un sprite sheet que contiene las imágenes a utilizar en los distintos tipos de baldosa y
- un atributo `dataAsset` que especifica el estado inicial de las baldosas. El estado de las baldosas se especifica como un array de arrays que contienen números identificando el tipo de baldosa dentro del spriteSheet especificado en `sheet`.

Por defecto el fondo de baldosas no produce colisiones, para que lo haga, en lugar de añadirlo al escenario con el método `stage.insert(tileLayer)` se añade con `stage.collisionLayer(tileLayer)` que está configurado para que por defecto no se colisione con las baldosas identificadas con 0.

Por ejemplo, usando los sprites de la figura 4

Figure 4: Baldosas

y definiendo el fondo así en el fichero `level.json`:

```
[
[ 1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,2,2,2,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,2,2,2,1,1,1,0,0,1,1,1,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
[ 1,2,2,2,1,1,1,0,0,1,1,1,0,0,1,1,1,1,1,1,1,1,0,0,1,1,1,0,0,1,0,0,0,0,1,1,1],

...

]
```

Este fragmento de código:

```
Q.scene("level1",function(stage) {

  // Add in a repeater for a little parallax action
  stage.insert(new Q.Repeater({
                       asset: "background-wall.png",
                       speedX: 0.5, speedY: 0.5 }));

  // Add in a tile layer, and make it the collision layer
  stage.collisionLayer(new Q.TileLayer({
                             dataAsset: 'level.json',
                             sheet:     'tiles' }));

  ...
```

producirá el resultado que se muestra en la figura 5

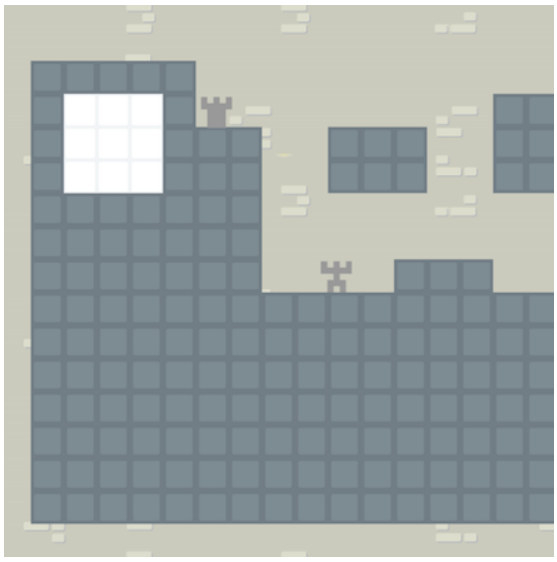

Figure 5: Fondos con baldosas

Gestión del movimiento

El componente `2d` añade a cualquier sprite primitivas de movimiento y detección de colisiones. Añade también un componente de aceleración vertical que simula la gravedad, de forma que si no lo remediamos, un sprite con el componente `2d` caerá indefinidamente.

Para controlar la gravedad, podemos modificar los valores que afectan a todos los sprites, `Q.gravityX` y `Q.gravityY`, o modificar localmente a un sprite el factor `p.gravity` que se multiplica por la gravedad global para obtener el valor específico para un sprite. Por ejemplo, asignando el valor 0 a `p.gravity` desaparece el efecto de la gravedad sobre ese sprite.

El componente `2d` añade también algunos eventos a los objetos, que permiten diferenciar el tratamiento de colisiones, dependiendo de si se ha colisionado con algo por encima (`bump.top`), por abajo (`bump.bottom`), a la izquierda (`bump.left`) o a la derecha (`bump.right`).

Por ejemplo, podemos destruir los enemigos sobre los que salte el jugador:

```
Q.Sprite.extend("Player", {
  init: function(p) {
    this._super(p);

    this.add("2d");
    this.on("bump.bottom",this,"stomp");
  },

  stomp: function(collision) {
    if(collision.obj.isA("Enemy")) {
      collision.obj.destroy();
      this.p.vy = -500; // make the player jump
    }
  }

});
```

Mapas en formato TMX

Como especificar el mapa usando un array de números es un tanto laborioso, Quintus ofrece también la posibilidad de cargar mapas en formato TMX (*Tile Map XML*) que se generan con el editor de mapas Tiled (http://www.mapeditor.org) que se muestra en la figura 6

Para que puedan ser cargados en Quintus, en los mapas creados con Tiled la capa de colisión ha de tener una propiedad `collision=true` y sólo puede haber una capa de *background*. Si la capa de *background* es una imagen *tileada* entonces

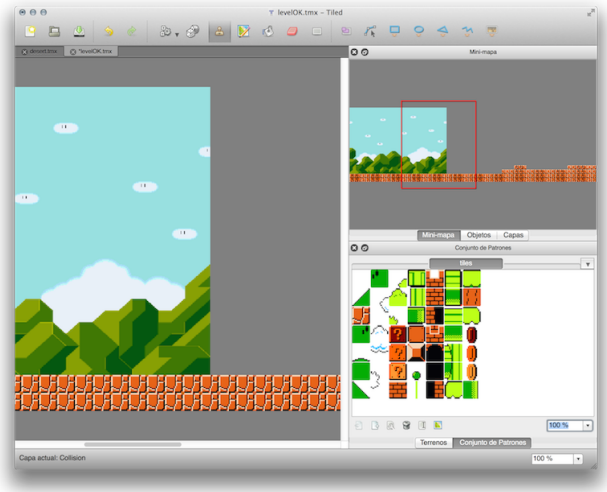

Figure 6: Editor de mapas Tiled

necesitaremos incluir el módulo `Anim` para usar el mapa en Quintus, además de cargar, claro está, el propio módulo `TMX`.

Finalmente, para añadir el mapa a un escenario, es necesario cargar el recurso con la función `loadTMX` y añadirlo a un escenario con `stageTMX`:

```
Q.scene("level1",function(stage) {
      Q.stageTMX("level.tmx",stage);
});

Q.loadTMX("level.tmx", function() {
    Q.stageScene("level1");
});
```

Estado global del juego

Este es un mecanismo que permite gestionar una serie de variables que definen el estado global del juego (número de vidas, puntos, inventario, ...) y además permiten que distintas entidades se registren a los cambios de estado de esa variable global. El estado se guarda en el atributo `Q.state` que almacena una instancia de `Q.GameState`.

Cada vez que se inicie una partida podemos llamar a `Q.state.reset({ ... props ... })` para fijar los valores iniciales del estado global y de paso limpiar la lista de objetos suscritos a los cambios del estado global:

```
Q.state.reset({ score: 0, lives: 2 });
```

Vamos a crear una etiqueta que se actualiza cada vez que cambia la puntuación:

```
Q.UI.Text.extend("Score",{
  init: function(p) {
    this._super({
      label: "score: 0",
      x: 0,
      y: 0
    });

    Q.state.on("change.score",this,"score");
  },

  score: function(score) {
    this.p.label = "score: " + score;
  }
});
```

Para que se generen los eventos de cambio, es necesario que los valores del estado global se establezcan con los métodos designados al efecto: `set`, `inc` y `dec`:

```
Q.state.set("score",50); // set the score to fifty
Q.state.set({ score: 50, lives: 1 }); // alternative object syntax

Q.state.inc("score",50); // add 50 to the score
Q.state.dec("score",50); // remove fifty from the score
```

Las anteriores llamadas desencadenan los eventos "change.score" y "change".

También hay un método para consultar por el valor de un atributo del estado global: `Q.state.get(prop)`:

```
Q.state.get("score"); // return the score
```

Manejo de la entrada

Quintus incluye dos módulos relacionados con la gestión de la entrada de usuario:

- `Quintus.Input`, para procesar las pulsaciones de teclado y pintar teclas y procesar su pulsación en pantallas táctiles.
- `Quintus.Touch`, para controlar la entrada de ratón y los eventos de pulsación y arrastre de un sprite concreto en dispositivos con pantalla táctil.

Configuración por defecto

La función `controls()` del módulo `Input` proporciona automáticamente una configuración por defecto que puede ser útil en muchos casos:

```
var Q = Quintus().include("Input")
                 .setup()
                 .controls();
```

Este método, además de configurar 6 valores de entrada, en dispositivos móviles dibuja en la pantalla 4 botones que representan: el desplazamiento a la izquierda y a la derecha y los botones de acción “a” y “b”, como se muestra en la figura 7

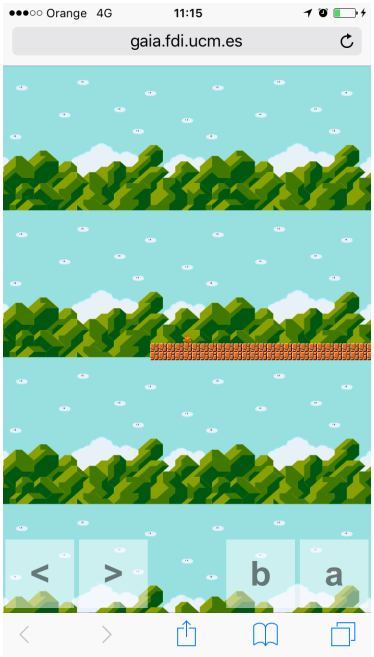

Figure 7: Controles en móvil

En Quintus, se escribe el código que procesa la entrada en términos de “nombres lógicos” de las entradas (`up`, `fire`, ...) para así abstraer del mecanismo de entrada concreto. El método `controls()` configura estos 6 tipos de entrada:

- up - para la pulsación de la flecha arriba
- down - flecha abajo
- left - flecha izquierda o botón izquierdo en la pantalla táctil
- right - flecha derecha o botón derecha en la pantalla táctil
- fire - barra espaciadora o tecla “z” pulsada, o bien botón “a” pulsado en pantalla táctil
- action - tecla “x” pulsada o botón “b” pulsado en pantalla táctil

Para saber si se ha producido una pulsación de teclado, se consulta una variable booleana, con el nombre lógico de la entrada, en el objeto `Q.inputs`. Así, por ejemplo, podríamos comprobar en el método `step` del sprite de usuario si una determinada tecla ha sido pulsada, para realizar las acciones correspondientes:

```
if(Q.inputs['action']) {
  // do something
}
```

Este método de consultar periódicamente por los eventos de teclado se denomina "encuesta" (*polling*) y aunque es el más sencillo de implementar, puede resultar ineficiente si se consulta con demasiada frecuencia sobre eventos que no ocurren tan frecuentemente. En esos casos es mejor gestionarlo por medio de eventos.

Eventos de teclado

Para recibir los eventos de pulsación de teclado es necesario registrarse en el objeto `Q.input` (no confundir con `Q.inputs` que almacena el estado de las teclas) al evento con el mismo nombre lógico que la tecla a la que queremos atender. Por ejemplo:

```
Q.Sprite.extend("Player",{
  init: function(p) {
    this._super(p);

    Q.input.on("fire",this,"fireWeapon");
  },

  fireWeapon: function() {
    // Do something
  }
});
```

Para no generar notificaciones de eventos innecesarias y teniendo en cuenta que `Q.input` es una variable global que persiste durante toda la ejecución del juego, es buena idea invocar el método `destroy` (heredado de la clase `Component`) sobre un sprite que esté registrado para recibir eventos de teclado cuando no lo necesitemos más. En particular, como una escena no destruye los sprites que contiene cuando es destruida, podemos incluir código que se encargue de ello:

```
Q.scene("testerScene",function(stage) {
    var player = new Q.Player();

    stage.on("destroy",function() {
        player.destroy();
        // or you can call player.debind() directly.
    });
});
```

Mapeo del teclado

Si queremos ir más allá de la configuración por defecto que proporciona `Q.controls`, podemos usar la función `Q.input.keyboardControls` para asociar nombres lógicos con las teclas que nos interesa procesar. La configuración se hace a través de un objeto que asocia al código de la tecla el nombre que nos interesa:

```
var Q = Quintus().include("Input").setup();

Q.input.keyboardControls({
  LEFT: "izquierda",
  RIGHT: "derecha"
});
```

Para identificar las teclas podemos usar sus códigos de teclado o los identificadores que se definen al principio del fichero `quintus_input.js` (LEFT, RIGHT, UP, DOWN, SPACE, ...):

```
Quintus.Input = function(Q) {
  /**
   * Provided key names mapped to key codes - add more names and key codes as necessary
   */
  var KEY_NAMES = Q.KEY_NAMES = {
    LEFT: 37, RIGHT: 39,
    UP: 38, DOWN: 40,
    SPACE: 32, Z: 90, X: 88, ENTER: 13,
    ESC: 27, P: 80, S: 83
  };

  var DEFAULT_KEYS = {
    LEFT: 'left', RIGHT: 'right',
    UP: 'up',     DOWN: 'down',
    SPACE: 'fire', Z: 'fire',
    X: 'action', ENTER: 'confirm',
    ESC: 'esc', P: 'P', S: 'S'
  };
```

Control táctil y eventos de ratón

El módulo `Touch` incluye controles que permiten manipular sprites individuales en pantallas táctiles o con el ratón en el ordenador. Para utilizarlo es necesario incluir los módulos `Sprite` y `Touch` y activarlo invocando al método `Q.touch()`:

```
var Q = Quintus().include("Sprites,Touch")
                 .setup()
                 .touch();
```

Aunque no queramos utilizar este tipo de control, es imprescindible incluir este módulo e inicializarlo si tenemos alguna escena con elementos de interfaz de usuario que han de pulsarse con el ratón (por ejemplo botones).

El método `touch` recibe dos parámetros opcionales que permiten especificar el tipo de sprites que pueden recibir eventos táctiles (por defecto sólo `Q.SPRITE_UI`, del módulo `UI`) y un array que especifica las capas (`stages`) en las que se procesarán eventos de pulsación (Por defecto solo se comprueban en las capas 0, 1 y 2): `Q.touch(tipo1 | tipo2..., [i1, 12...])`. Para poder acceder a la constante `Q.SPRITE_ALL` es necesario separar la instanciación de Quintus de su configuración:

```
var Q = window.Q = Quintus()
    .include("Sprites, Scenes, Input, 2D, Anim, Touch, UI, TMX");

  Q.setup({ maximize: true })
    .controls()
    .touch(Q.SPRITE_ALL);
```

Los sprites que puedan recibir eventos táctiles recibirán tres eventos: `touch`, `drag` y `touchEnd`, con un parámetro que contiene información sobre el evento, como, por ejemplo, las coordenadas donde ha tenido lugar, o, si se trata de un evento `drag` la distancia que se ha arrastrado en cada eje:

```
Q.Sprite.extend("Mario",{

  init: function(p) {
    this._super(p, {
      sheet: "marioR",
      frame: 1,
      scale: 3
    });
    this.add('2d, platformerControls, animation, tween');
    this.on("drag");

  },

  drag: function(touch) {
    this.p.x = touch.origX + touch.dx;
    this.p.y = touch.origY + touch.dy;
  },

});
```

En la práctica, lo más sencillo es añadir el componente `platformerControls` a la entidad que queramos controlar y de esta forma podremos moverlo con las teclas en un ordenador y arrastrándolo con el dedo en una pantalla táctil (quizás ...).

Animaciones

Los componentes `animation` y `tween` del módulo `Anim` proporcionan a un sprite la capacidad de reproducir animaciones. `animation` secuenciando sprites de un spritesheet y `tween` desplazando y rotando una imagen dada.

Animaciones fotograma a fotograma

El primer paso es crear una hoja de animaciones con la función `Q.animations` donde se pasa el nombre de la hoja de animaciones, al que se le asocia un objeto donde se da nombre y se describen distintas animaciones. En la definición de la hoja de animaciones no se fija la imagen con la que trabajará (el sprite sheet), que se fija después al usar la hoja de animaciones en un sprite concreto y establecer su propiedad `sheet`.

```
Q.animations("player_anim", {
      walk_right: { frames: [0,1,2,3,4,5,6,7,8,9,10], rate: 1/15,
                    flip: false, loop: true },
      walk_left: { frames:  [0,1,2,3,4,5,6,7,8,9,10], rate: 1/15,
                   flip:"x", loop: true },
      jump_right: { frames: [13], rate: 1/10, flip: false },
      jump_left: { frames:  [13], rate: 1/10, flip: "x" },
      stand_right: { frames:[14], rate: 1/10, flip: false },
      stand_left: { frames: [14], rate: 1/10, flip:"x" },
      duck_right: { frames: [15], rate: 1/10, flip: false },
      duck_left: { frames:  [15], rate: 1/10, flip: "x" },
      climb: { frames:  [16, 17], rate: 1/3, flip: false }
    });
```

Donde se ha definido la hoja de animaciones “player” y se le han asociado 9 animaciones distintas.

Una vez añadida la entrada a `Q.animations` ya se pueden añadir esas animaciones a un sprite. Para ello, es necesario:

- establecer la propiedad `sprite` del sprite con el nombre de la hoja de animaciones que hemos añadido a `Q.animations`
- añadir el componente `animation` a la clase del sprite
- invocar el método `play(nombre_animación)` sobre el sprite

Por ejemplo:

```
Q.Sprite.extend("Player", {
  init: function(p) {
    this._super(p,{
      sprite: "player_anim",
      sheet: "player"
    });
```

```
    this.add("animation");
  }
});

var player = new Q.Player();

player.play("fire_right");
```

El sprite debe tener una propiedad `sheet` con un sprite sheet que contenga al menos el número de frames al que se hace referencia en la definición de animaciones de la propiedad `sprite`, como es el caso de este ejemplo del juego de plataformas en Quintus (http://www.html5quintus.com/quintus/examples/platformer_full/) que se muestra en la figura 8

Figure 8: sprite sheet del player

que corresponden al json:

```
{
  "player":{"sx":0,"sy":1,"cols":18,"tilew":72,"tileh":97,
            "frames":18}
}
```

Cada animación se configura con los siguientes parámetros (sólo `frames` es obligatorio), que permiten encadenar animaciones y generar eventos cuando estas terminan, la base de la IA de un juego:

- `frames`: un array con los números de los fotogramas que componen la animación
- `rate`: tiempo que debe transcurrir entre un fotograma y otro (en segundos)
- `loop`: por defecto es cierto, lo que hace que la animación se repita indefinidamente. Si es falso entonces sólo se ejecuta una vez
- `next`: la animación que hay que ejecutar cuando esta termine (automáticamente pone `loop` a falso)
- `trigger`: el evento que hay que lanzar cuando termine la animación (útil, por ejemplo, para lanzar una bala después de ejecutar la animación de disparar). Evidentemente `loop` tiene que estar a falso para que la animación termine.

Las animaciones de este ejemplo permiten coordinar la animación de disparo con la creación de la bala:

```
var Q = Quintus().include("Sprites, Anim");

Q.animations('player_anim', {
```

```
  run_right: { frames: [7,6,5,4,3,2,1], rate: 1/15},
  run_left: { frames: [19,18,17,16,15], rate: 1/15 },
  fire_right: { frames: [9,10,10], next: 'stand_right', rate: 1/30,
                trigger: "fired" },
  fire_left: { frames: [20,21,21], next: 'stand_left', rate: 1/30,
                trigger: "fired" },
  stand_right: { frames: [8], rate: 1/5 },
  stand_left: { frames: [20], rate: 1/5 },
  fall_right: { frames: [2], loop: false },
  fall_left: { frames: [14], loop: false }
});
```

El método `play` recibe un segundo parámetro opcional con la prioridad de la animación, asignando 0, la prioridad más baja, como valor por defecto. Aprovechándose de este hecho y de que invocar a `play` con una animación que ya se está ejecutando no tiene ningún efecto, una técnica habitual en la gestión de animaciones es lanzar las animaciones repetitivas (andar, correr, ...) en el método `step` y utilizar los eventos de disparo o colisión para ejecutar otras animaciones con una prioridad más alta.

```
Q.Sprite.extend("Player", {
  init: function(p) {
    this._super(p,{
      sprite: "player_anim",
      sheet: "player"
    });

    this.add("2d, platformerControls, animation");

    Q.input.on("fire");

    // Wait until the firing animation has played until
    // actually launching the bullet
    this.on("fired",this,"launchBullet");
  },

  fire: function() {
    // Play the fire animation at a higher priority
    this.play("fire_" + this.p.direction);
  },

  launchBullet: function() {
    var bullet = new Q.Bullet({  ... });
    this.stage.insert(bullet);
  },

  step: function(dt) {
```

```
    if(this.p.vx > 0) {
      this.play("run_right");
    } else if(this.p.vx < 0) {
      this.play("run_left");
    } else {
      this.play("stand_" + this.p.direction);
    }
  }
});
```

Este ejemplo funciona porque en la hoja de animaciones `player` hemos especificado que las animaciones `fire_right` y `fire_left` lanzan el evento `fired` cuando se terminan de ejecutar, lo que en este sprite lanzará la ejecución del método `launchBullet`.

Animación por interpolación

El componente `tween` permite animar una imagen desde una posición y ángulo iniciales hasta una posición y ángulo finales, pudiéndose especificar así mismo la duración de la animación y una función que controla la velocidad de la transición punto a punto. Por ejemplo, para mover una imagen de la posición `0,0` a la `500,500` dando una vuelta completa sobre sí misma, en 1 segundo, e interpolando de forma lineal entre el estado inicial y el final:

```
var sprite = new Q.Sprite({ asset: "image.png",
                            x:0, y:0, angle: 0 });
sprite.add("tween");

sprite.animate({ x: 500, y: 500, angle: 360 });
```

El método `animate` permite especificar los parámetros adicionales:

```
function animate(properties,[duration,] [easing,] [options])
```

donde `properties` es un objeto con las propiedades del sprite al final de la animación, `duration` es la duración en segundos, y `easing` es una función que toma un valor en [0..1] y devuelve otro valor en el mismo intervalo y que sirve para determinar la velocidad de la animación punto a punto. Se incluyen algunas funciones de transición:

- `Q.Easing.Linear` - la transición se ejecuta linealmente entre 0 y 1
- `Q.Easing.Quadratic.In` - empieza lentamente y luego acelera
- `Q.Easing.Quadratic.Out` - empieza rápido y luego se frena
- `Q.Easing.Quadratic.InOut` - empieza despacio, acelera, desacelera y termina despacio

Por ejemplo

```
Q.Sprite.extend("Mario",{

  ...

  fire: function() {
    this.p.angle = 0;
    this.animate({ x: 700, y: 400, angle: 360 },
                 3, Q.Easing.Quadratic.In);
  },

  ...
}
```

Por último, en `options` se puede pasar un objeto que especifique los atributos:

- `delay` - los segundos que hay que esperar antes de empezar a ejecutar la animación
- `callback` - una función a la que invocar una vez terminada la animación

El componente `tween` añade otros dos métodos a la entidad donde se incluye: `chain` y `stop`. `chain` hace lo mismo que `animate` excepto que espera a iniciarse que acabe la animación que se esté ejecutando, y `stop` detiene todas las animaciones.

Audio

El soporte de audio es todavía el punto débil de HTML 5, en parte debido a la competencia entre soluciones abiertas (Ogg) y propietarias (MP3). Por ello es conveniente que los recursos de audio se incluyan en ambos formatos, .ogg y .mp3, para así cubrir todos los navegadores compatibles con Quintus. Además, Quintus incluye soporte para las APIs de HTML5 Audio y Web Audio (especificación promovida por el W3C). Si está disponible Quintus utiliza Web Audio, que es más eficiente, y si no pasa a HTML5 Audio.

Para incluir el soporte de audio en un proyecto, es necesario incluir el módulo `Audio`, e iniciarlo con la llamada `enableSound`, que se debe ejecutar antes de cargar ningún sonido:

```
var Q  = Quintus().include("Audio")
                  .enableSound();
```

En la instanciación de Quintus se puede especificar los tipos de fichero de audio que se incluyen en el juego. Por ejemplo:

```
var Q = Quintus({ audioSupported: [ 'wav','mp3' ] });
```

donde se indica que cada fichero de audio vendrá en los formatos .wav y .mp3, y que primero se intentará reproducir la versión .wav.

La carga se realiza con la función `Q.load` a la que le debemos pasar el nombre del fichero con una de las versiones del audio, y será ese mismo nombre el que luego usemos para reproducirlo:

```
Q.load([ "fire.mp3" ], function() {
  ...
});
```

Nótese que si hemos especificado que el orden de preferencia del audio es `audioSupported: [ 'wav','mp3' ]`, aunque carguemos usando el nombre `fire.mp3`, Quintus intentará primero ejecutar la versión .wav si el navegador la soporta.

Reproducción de audio

Para reproducir un sonido se utiliza la función `Q.audio.play` pasándole el mismo nombre que se utilizó al cargarlo:

```
Q.audio.play('fire.mp3');
```

`Q.audio.play` admite un segundo parámetro que permite especificar el tiempo, en segundos, antes de que se pueda volver a reproducir este sonido.

Para reproducir el sonido en bucle (*loop*), se pasa un objeto con el atributo `loop` a `true`:

```
Q.audio.play('music.mp3',{ loop: true });
```

Que hará que se reproduzca el sonido hasta que lo paremos. `Q.audio.stop()`sin argumentos detiene todas las reproducciones de sonidos en marcha:

```
Q.audio.stop(); // Everything will stop playing
```

y si le pasamos un nombre de recurso de audio, detiene sólo a ese:

```
Q.audio.stop("music.mp3");
```

Interfaz de usuario

El módulo `UI` permite crear elementos de la interfaz de usuario, como botones, etiquetas y contenedores para agruparlos.

La clase `Q.UI.Container` es un contenedor de elementos de UI. Tiene las siguientes propiedades:

- border: grosor del borde (por defecto 0)
- radius: radio del borde redondeado (por defecto 5)
- stroke: color del borde (por defecto negro #000)
- w, h: ancho y alto del contenedor
- x, y: coordenadas de la esquina superior izquierda

- fill: color de relleno (por defecto null)
- shadow: si el contenedor tiene sombra (false por defecto)
- shadowColor: Color RGB de la sombra (falso por defecto)
- opacity: Para indicar si es o no transparente (por defecto 1, totalmente opaco)

Además dispone de los siguientes métodos:

- `insert(comp)`: inserta un componente en el contenedor.
- `fit(paddingX, paddingY)`: fija el tamaño del contenedor de acuerdo a los elementos que contiene. Los parámetros representan el relleno exterior del contenedor.

La clase `Q.UI.Text` representa una etiqueta de texto, con las siguientes propiedades:

- label: texto, que podemos modificar en cualquier momento.
- weight: grosor de texto (a la CSS) (por defecto es 800)
- size: tamaño en píxeles (por defecto 24)
- align: alineación horizontal (por defecto left, valores posibles center y right)
- family: fuente de letra (por defecto arial)
- color: color de la fuente (por defecto black)
- outline: color del borde del texto (por defecto black)
- outlineWidth: grosor de la línea (por defecto 0, sin borde)
- opacity: para indicar si es o no transparente (por defecto 1, totalmente opaco)

La clase `Q.UI.Button` permite crear botones. Hereda de `Q.UI.Container` por lo que tiene las propiedades del mismo, además de estas otras:

- keyActionName: nombre lógico de la tecla rápida asociada al botón
- label: texto del botón
- font: cadena con el tipo de letra (a la CSS) (por defecto `400 24px arial`)
- fontColor: color de la fuente (por defecto black)
- sheet o asset: imagen para representar al botón

Si la propiedad `sheet` tiene más de un fotograma, entonces se presenta el segundo mientras el botón permanece pulsado.

Cuando se construye el botón se puede pasar una función que se ejecutará cada vez que se pulse. Con cada pulsación el botón lanza el evento `click`, por lo que también podemos asociar el comportamiento del botón con ese evento.

Ejemplo de final de partida:

```
Q.scene('endGame',function(stage) {
  var container = stage.insert(new Q.UI.Container({
    x: Q.width/2, y: Q.height/2, fill: "rgba(0,0,0,0.5)"
  }));

  var button = container.insert(new Q.UI.Button({
```

```
    x: 0, y: 0, fill: "#CCCCCC", label: "Play Again"
  }));

  var label = container.insert(new Q.UI.Text({
    x:10, y: -10 - button.p.h, label: stage.options.label
  }));

  // When the button is clicked, clear all the stages
  // and restart the game.
  button.on("click",function() {
    Q.clearStages();
    Q.stageScene('mainTitle');
  });

  // Expand the container to visibly fit it's contents
  // (with a padding of 20 pixels)
  container.fit(20);
});
```

El resultado es el que se muestra en la figura 9

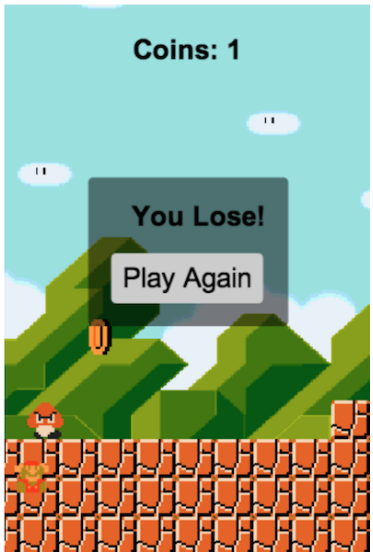

Figure 9: Intefaz de usuario

Gestión de eventos

Normalmente las clases que construimos no heredan directamente de `Q.Class`, sino que heredan de otras clases más específicas que ya incluyen cierta funcional-

idad como `Q.Sprite` o `Q.GameObject`. Lo normal es que como mínimo todas las clases en Quintus hereden de `Q.Evented` que es la clase que proporciona la gestión de eventos, un mecanismo que se usa extensivamente en Quintus, sobre todo en la gestión de componentes. `Q.Evented` incluye un mecanismo para registrarse a la espera de determinados eventos y para lanzar eventos.

Estos son los métodos para incluirse o eliminarse de la lista de objetos a la espera de un evento:

```
srcObj.on("eventName",[ targetObj, ] [ callback ]);
srcObj.off("eventName",[ targetObj, ] [ callback ]);
```

Si sólo se pasa el nombre del evento (una cadena), entonces se asume que `targetObject` es el mismo que `srcObj` y la función de `callback` es la misma que el nombre del evento. Si se proporciona un nombre de `callback` en lugar de una función literal, entonces se buscará la función en el atributo de ese nombre en `targetObj`.

Por ejemplo, el sistema de colisiones lanza un evento `hit` a un objeto cada vez que este colisiona con otro. Si queremos que un objeto ejecute su método `hit` cada vez que reciba ese evento es suficiente con:

```
srcObj.on("hit");
```

`Evented` también incluye un método `debind` que elimina un objeto de todos los eventos a los que estaba escuchando. Cuando se destruye un `Sprite` se invoca automáticamente a `debind` por lo que normalmente no es necesario que lo invoquemos nosotros explícitamente.

Un ejemplo del uso de eventos:

```
var spaceship = new Q.SpaceShip();

// Provide callback inline
spaceship.on("fire",function(gun) {
  console.log("Just fired gun: " + gun);
});
```

Los eventos se lanzan con el método `trigger` con el nombre del evento y hasta 3 parámetros adicionales, que se pasarán a la función que trata el evento. Debemos ser cuidadosos para que el mismo tipo de evento pase siempre el mismo tipo de parámetros:

```
spaceship.on("step", function() {
  if(Q.inputs['fire1']) {
    this.trigger("fire","gun1");
  }
  if(Q.inputs['fire2']) {
    this.trigger("fire","gun2");
  }
});
```

Quintus no soporta espacios de nombres en los eventos, por lo que puede ser buena idea utilizar prefijos propios en los nombres de evento.

Normalmente los eventos se asocian con un tipo de sprite definido por nosotros donde `srcObj` es el propio sprite, como en el ejemplo del juego de plataformas:

```
Q.Sprite.extend("Player",{

  init: function(p) {
    this.add('2d, platformerControls');
    this.on("hit.sprite",function(collision) {
      if(collision.obj.isA("Tower")) {
        Q.stageScene("endGame",1, { label: "You Won!" });
        this.destroy();
      }
    });
  }
});
```

Implementación del sistema de eventos

La gestión de eventos se implementa con el patrón *Observer* cuyo diagrama UML se muestra en la figura **??**

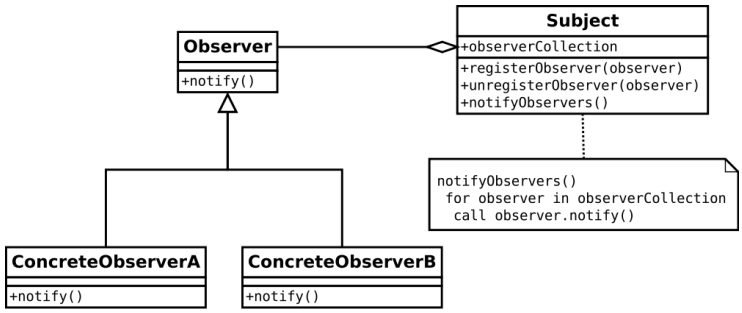

Figure 10: Patrón Observer

Este es el esqueleto de la clase `Evented`, que guarda la información sobre los observadores en el atributo `listeners` que es un objeto que a cada nombre de evento le asocia un array donde cada elemento es un array con el objeto destinatario de la retrollamada y la retrollamada propiamente dicha. Además en el observador, el `target`, mantiene un array `binds` con los eventos a los que observa para así poder borrarse de todos los `listeners` a los que esté registrado cuando el objeto sea destruido:

```
Q.Evented = Class.extend({
  on: function(event,target,callback) {
```

```
    // TODO: Fill in on code
    ...

    this.listeners[event].push([ target || this, callback]);

    ...

    target.binds.push([this,event,callback]);

    ...

  },

  trigger: function(event,data) {
    // TODO: Fill in trigger code
  },

  off: function(event,target,callback) {
    // TODO: Fill in off code
  },

  debind: function() {
   // TODO Fill in the debind code
  }
});
```

Componentes

La Programación orientada a objetos, a través de la herencia (jerarquías de tipos), el polimorfismo y la vinculación dinámica, soporta la vinculación tardía entre el código que implementa una abstracción y el que la utiliza, permitiendo así la evolución independiente de distintas partes de un sistema y facilitando la construcción de software más resistente al cambio, donde el coste de los cambios es lineal, en el sentido de que cambios pequeños en la especificación dan lugar a cambios pequeños en la implementación. Sin embargo, en este modelo basado en la existencia de una jerarquía de tipos, los cambios son más costosos cuando se producen en las clases base de la jerarquía, que son las que definen las abstracciones en las que se basa la vinculación tardía.

En un videojuego las *entidades* son los tipos de objetos que definen la lógica del juego, incluyendo el avatar del jugador, los enemigos o los elementos del escenario. Cuando en el desarrollo de videojuegos se adoptaron los lenguajes orientados a objetos (fundamentalmente C++) y se empezaron a utilizar jerarquías de clases para representar a las entidades del juego, pronto se dieron cuenta de que estas jerarquías eran demasiado rígidas para incorporar fácilmente los cambios de

diseño que se iban produciendo durante el desarrollo del juego. El resultado eran clases en la raíz de la jerarquía que incorporaban demasiada funcionalidad, que algunas de sus especializaciones utilizaban y otras no. Por ejemplo, la clase base de los *Sims* tenía más de 100 métodos y la de *Half Life 1* 87.

El problema de las clases base demasiado *gordas* es un indicio (*code smell*) de que el diseño abusa de la herencia y se debería apoyar más en la composición. El balance entre composición y herencia es el tema básico en los patrones de diseño, y la arquitectura basada en componentes usada mayoritariamente en los videojuegos hoy en día incorpora ideas de varios patrones (decorador y command, sobre todo).

Quintus incorpora un compromiso entre composición y herencia. Añade herencia basada en clases a JavaScript siguiendo la implementación propuesta por el creador de jQuery, John Resig (http://ejohn.org/blog/simple-javascript-inheritance/), pero construye una jerarquía de clases muy plana, donde básicamente los distintos tipos de entidades heredan directamente de la clase `Sprite` y luego se configuran variantes de las entidades utilizando componentes.

Los componentes son fragmentos de funcionalidad reutilizable que se pueden añadir o quitar de un objeto (una entidad del juego) dinámicamente. Son una alternativa a las jerarquías de clases porque permiten definir nuevos *tipos* de entidades sin necesidad de definir clases nuevas y además permiten cambiar el *tipo* de un objeto dinámicamente.

Uso de los componentes en Quintus

El mecanismo de los componentes en Quintus lo implementa la clase `GameObject`, que hereda de `Evented`. Los componentes normalmente envían un evento al objeto que los contiene cada vez que algo interesante ocurre, y a su vez, se registran para ser notificados cuando determinados eventos tienen lugar.

La manera de programar una entidad (un sprite) en Quintus consiste básicamente en configurarla con una cierta funcionalidad que proporcionan una serie de componentes y luego estar a la escucha de los eventos que lancen esos componentes. Los componentes se añaden con el método `GameObject.add` al que se le pasa una cadena con nombres de componentes separados por comas.

Por ejemplo, el módulo `2D` de Quintus define una serie de componentes que facilita la programación de juegos en 2D y el módulo `Input` define componentes que permiten que un sprite sea controlado por el usuario. Simplemente añadiendo un par de componentes a un Sprite conseguimos que se comporte como una entidad controlada por el jugador en un juego de plataformas 2D:

```
var Q = Quintus().include("Sprites, Scenes, 2D, Input");

 // Every instance of Q.Player will have these two components
 Q.Sprite.extend("Player", {
```

```
  init: function() {
    this.add("2d, platformerControls");
  }
});
```

Los componentes se pueden añadir y eliminar (con el método `GameObject.del`) dinámicamente, por ejemplo para cambiar el arma del jugador:

```
player.del("pistol");
player.add("rocketLauncher");
```

y el método `GameObject.has` nos permite determinar si un determinado objeto tiene un cierto componente:

```
player.has("pistol"); // false
player.has("rocketLauncher"); // true
```

Creación de componentes

Para crear un nuevo componente, se utiliza el método `Q.component`, al que se le pasa el nombre del componente y sus métodos. Para inicializar el estado de un componente normalmente no se usa el método `init` al que se llama cuando se crea el componente, sino el método `added` que es invocado cuando el componente ya ha sido creado y añadido a la entidad de la que formará parte. El objeto al que se añade un componente está en su propiedad `entity`.

Al crear un componente, además de al propio componente, se pueden definir métodos que serán añadidos al objeto donde se añada el componente, a través del atributo `extend` del componente. Por ejemplo, este componente tienen un método `refillAmmo` para recargar la munición y añade un método `fire` al objeto donde sea añadido:

```
Q.component("pistol", {
  added: function() {
    // Start the ammo out 1/2 filled
    this.entity.p.ammo = 30;
  },

  refillAmmo: function() {
    // We need to say this.entity because refillAmmo is
    // added on the component
    this.entity.p.ammo = 60;
  },

  extend: {
    fire: function() {
      // We can use this.p to set properties
      // because fire is called directly on the player
```

```
      if(this.p.ammo > 0) {
        this.p.ammo-=1;
        console.log("Fire!");
      }
    }
  }
});


// Example usage:

Q.Sprite.extend("Player");

var player = new Q.Player();

player.add("pistol");

// Call fire directly
player.fire(); // Fire!

// Call refillAmmo on the pistol component
player.pistol.refillAmmo();

// Remove the pistol component
player.del("pistol");

// Should cause an error
player.fire();
```

Los componentes se suelen enganchar también a los eventos de pintado, al bucle principal y a las colisiones, por ejemplo el componente `viewport` que define una cámara que sigue al jugador se registra en los eventos de pintado:

```
Q.component('viewport',{
    added: function() {
      this.entity.on('prerender',this,'prerender');
      this.entity.on('render',this,'postrender');

    ...

  }
```

el componente `2d` que proporciona movimiento en 2D se registra en el bucle principal:

```
Q.component('2d',{
    added: function() {
      var entity = this.entity;
```

```
      entity.on('step',this,"step");
      entity.on('hit',this,'collision');

    ...

  }
```

y el componente `aiBounce` que implementa el comportamiento de rebotar con las paredes, se registra a los eventos de colisión lateral:

```
Q.component('aiBounce', {
    added: function() {
      this.entity.on("bump.right",this,"goLeft");
      this.entity.on("bump.left",this,"goRight");

    ...

  }
```

La clase `GameObject`

La clase `GameObject` extiende a la clase `Evented` y es la encargada de incluir la funcionalidad de los "contenedores de componentes". Tiene 4 métodos, `has`, `add` y `del` para gestionar componentes y `destroy` para gestionar su destrucción:

- El método `has` comprueba si una entidad ya tiene incorporado un componente dado
- `add` añade uno o más componentes a una entidad. Para cada uno de ellos, instancia un objeto del tipo dado y lanza sobre la propia entidad el evento `addComponent`. La conexión entre entidad y componente la realiza el método `init` del componente que es invocado en la instanciación del objeto
- `del` elimina uno o más componentes de una entidad. Lanza el evento `delComponent` e invoca el método `destroy` del componente
- `destroy` elimina el objeto (teniendo cuidado de no eliminarlo más de una vez), envía el sobre sí mismo el evento `destroyed`, lo borra de las colas de eventos (invocando su propio método `debind`), lo elimina del escenario donde pueda estar, y si tiene un método `destroyed` lo ejecuta

`Sprite` extiende a `GameObject` y hereda así su funcionalidad

El bucle principal

Quintus ya incluye un bucle principal que se incorpora al juego de forma transparente cuando utilizamos el módulo `Scenes` y añadimos una escena al juego.

Podemos añadir un bucle principal en forma de una función `callback` utilizando el método `Q.gameLoop(callback)`, que se encargará de invocar a `callback` hasta 60 veces por segundo y que cada vez que la invoque le pasará el número de segundos que han transcurrido desde la última invocación. Por ejemplo:

```
Q.gameLoop(function(dt) {
    Q.clear();
    // .. do something ..
});
```

No es buena idea que un juego tenga más de un bucle principal así es que si se usa el modulo `Scenes` es mejor dejar la responsabilidad a la implementación por defecto.

Es posible parar y arrancar el juego congelando el bucle principal, que dejará de actualizar la lógica (`step`) y el dibujado:

```
Q.pauseGame();
// Do something ..
Q.unpauseGame();
```

Implementación del bucle principal

El bucle principal ejecuta la lógica y el dibujado cada cierto tiempo. El bucle principal utiliza un temporizador para, después de cada iteración, programar su ejecución un tiempo después y mientras tanto ceder el control al navegador que será el encargado de actualizar la página y gestionar los eventos de entrada.

Este es el bucle principal que utilizamos en el juego que desarrollamos sin Quintus:

```
  var boards = [];

  this.loop = function() {
    var dt = 30 / 1000;

    // Cada pasada borramos el canvas
    Game.ctx.fillStyle = "#000";
    Game.ctx.fillRect(0,0,Game.width,Game.height);

    // y actualizamos y dibujamos todas las entidades
    for(var i=0,len = boards.length;i<len;i++) {
      if(boards[i]) {
        boards[i].step(dt);
        boards[i].draw(Game.ctx);
      }
    }
```

```
    setTimeout(Game.loop,30);
  };

};
```

Que tiene algunos defectos:

- supone que efectivamente han pasado 30 milisegundos desde la última vez que se ejecutó
- aunque la pestaña del juego no sea la que esté activa en el navegador, el bucle principal se seguirá ejecutando, ralentizando así al ordenador.

Además, puede ocurrir que se pierdan fotogramas si intentamos ejecutar el bucle principal con demasiada frecuencia. Por ejemplo, si usamos un periodo de 10ms eso quiere decir que en un monitor con frecuencia de refresco de 60Hz se perderá uno de cada tres fotogramas, ya que con esa frecuencia de refresco la pantalla se repinta cada 16,7 milisegundos (1000/60) lo que implica que antes de poder pintar el segundo fotograma, ya se habrá generado el tercero.

La solución es utilizar el API `requestAnimationFrame` donde no se fija el tiempo en que se quiere volver a ejecutar el bucle principal sino que es el propio navegador en base a la velocidad de refresco del ordenador y a la visibilidad de la ventana quien se encarga de calcular el tiempo que debe transcurrir hasta la siguiente llamada. En la página http://ie.microsoft.com/TEStdrive/Graphics/RequestAnimationFrame/ se muestran dos relojes, uno implementado con `setTimeOut` y otro con `RequestAnimationFrame` donde se puede apreciar este efecto.

Este es el bucle principal de Quintus:

```
Q.gameLoop = function(callback) {
    Q.lastGameLoopFrame = new Date().getTime();

    // Keep track of the frame we are on (so that animations can
    //  be synced to the next frame)
    Q._loopFrame = 0;

    // Wrap the callback to save it and standardize the passed
    // in time.
    Q.gameLoopCallbackWrapper = function() {
      var now = new Date().getTime();
      Q._loopFrame++;
      Q.loop =
        window.requestAnimationFrame(Q.gameLoopCallbackWrapper);
      var dt = now - Q.lastGameLoopFrame;
      /* Prevent fast-forwarding by limiting the length of
      a single frame. */
      if(dt > Q.options.frameTimeLimit) {
        dt = Q.options.frameTimeLimit; }
```

```
      callback.apply(Q,[dt / 1000]);
      Q.lastGameLoopFrame = now;
    };

    window.requestAnimationFrame(Q.gameLoopCallbackWrapper);
    return Q;
  };
```

El párametro de configuración de Quintus `Q.options.frameTimeLimit` establece el tiempo máximo que hacemos avanzar la simulación entre dos fotogramas consecutivos. Este tiempo puede haber sido mucho mayor si la pestaña del juego ha dejado de ser la pestaña activa.

Aquí hay un ejemplo de uso de este parámetro que provoca que la bola que se está dibujando de un salto si activamos otra pestaña y volvemos al juego segundos después. Incluyendo un contador de tiempo de juego, frames por segundo y botones para parar y reanudar la caida de la pelota.

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport"
      content="width=640, user-scalable=0, minimum-scale=1.0,
               maximum-scale=1.0"/>
    <title>Ball</title>
    <script src='http://code.jquery.com/jquery-1.11.0.min.js'>
    </script>
    <script src='http://cdn.html5quintus.com/v0.1.6/quintus-all.js'>
    </script>
    <script src='game.js'></script>
    <style>
      body { padding:0px; margin:0px; background-color:black ;
             color:white; }
    </style>
  </head>
  <body>
   <div id='timer'>0</div>
    <div id='fps'>0</div>
    <button id='pause'>Pause</button>
    <button id='unpause'>Unpause</button>
    <script type="text/javascript">

    window.onload = game;
  </script>
  </body>
</html>
```

incluyendo este código en el fichero `game.js`:

```
var game = function () {

var Q = Quintus({frameTimeLimit:100}).include("Sprites").setup();

Q.Sprite.extend("Ball",{
  init:function(p) {
    this._super(p,{
      asset: "ball.png",
      x: 40,
      y: 100,
      vx: 5,
      vy: -10
    });
  },

  step: function(dt) {
    this.p.vy += dt * 0.8;

    this.p.x += this.p.vx * dt;
    this.p.y += this.p.vy * dt;
  }
});

Q.load(["ball.png"],function() {
  var ball = new Q.Ball();
  var totalTime=0;
  Q.gameLoop(function(dt) {
    totalTime += dt;
    ball.update(dt);
    Q.clear();
    ball.render(Q.ctx);
    $("#timer").text(Math.round(totalTime * 1000) + " MS");
    $("#fps").text(Math.round(Q._loopFrame / totalTime) + " FPS");
  });
});

$("#pause").on('click',Q.pauseGame);
$("#unpause").on('click',Q.unpauseGame);

}
```

El API `requestAnimationFrame` aparece en 2011 por lo que las versiones anteriores de los navegadores no lo soportan. Esta función asigna a `window.requestAnimationFrame` la implementación específica del navegador, si es que existe, y si no la define en términos de `setTimeOut`:

```
(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] ||
          window[vendors[x]+'CancelRequestAnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(
              function() { callback(currTime + timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
}());
```

Nótese que la implementación basada en `setTimeOut` intenta ejecutar el bucle principal cada 16 milisegundos. Si ha pasado más tiempo, `currTime`, del que estaba previsto, `lastTime`, entonces se resta la diferencia a los 16 ms para mantener la frecuencia.

Implementación de Sprite

El método `draw`

El dibujado de un objeto se realiza en el método `draw`. La preparación al dibujado se hace en el método `render`, que es el que se invoca desde el bucle principal del escenario donde está incluido el sprite. Entre otras cosas, el método `render` se encarga de aplicar una matriz de transformación que modifica las coordenadas del canvas:

```
/**
     @method render
     @for Q.Sprite
     @param {Context2D} ctx - context to render to
```

```
    */
    render: function(ctx) {
      var p = this.p;

      if(p.hidden) { return; }
      if(!ctx) { ctx = Q.ctx; }

      this.trigger('predraw',ctx);

      ctx.save();

        if(this.p.opacity !== void 0 && this.p.opacity !== 1) {
          ctx.globalAlpha = this.p.opacity;
        }

        this.matrix.setContextTransform(ctx);

        if(this.p.flip) { ctx.scale.apply(ctx,this._flipArgs[this.p.flip]); }

        this.trigger('beforedraw',ctx);
        this.draw(ctx);
        this.trigger('draw',ctx);

      ctx.restore();

      // Children set up their own complete matrix
      // from the base stage matrix
      if(this.p.sort) { this.children.sort(this._sortChild); }
      Q._invoke(this.children,"render",ctx);

      this.trigger('postdraw',ctx);

      if(Q.debug) { this.debugRender(ctx); }

    },
```

Con esta implementación de `render` los componentes se pueden suscribir a los eventos `predraw`, `beforedraw`, `draw` o `postdraw`.

El método `draw` sólo tiene que preocuparse de dibujar el sprite centrado en las coordenadas `(0,0)` con el ancho y el alto especificado en `p.w` y `p.h` respectivamente:

```
Q.Sprite.extend("Square",{
  init: function(p) {
    this._super(p,{
      color: "red",
```

```
      w: 50,
      h: 50
    });
  },

  draw: function(ctx) {
    ctx.fillStyle = this.p.color;
    // Draw a filled rectangle centered at
    // 0,0 (i.e. from -w/2,-h2 to w/2, h/2)
    ctx.fillRect(-this.p.cx,
                 -this.p.cy,
                 this.p.w,
                 this.p.h);

  }
});
```

Hay que tener cuidado si cambiamos las dimensiones de un sprite (`p.w` o `p.h`) de actualizar también las coordenadas de su centro (`p.cx` y `p.cy`).

El método `step`

Es el método `step` donde se avanza el estado del sprite en cada fotograma, por ejemplo actualizando la posición del jugador en base a la entrada o actualización la posición de los enemigos en base a su IA.

Al igual que en el dibujado, también hay dos métodos: `update` con el esquema genérico de actualización que no se suele redefinir, y `step` con las actualizaciones específicas del sprite. Esta es la implementación por defecto del método `update(dt)` que es invocado por el bucle principal del escenario donde está incluido el sprite:

```
/**
     @method update
     @for Q.Sprite
     @param {Float} dt - time elapsed since last call
    */
    update: function(dt) {
      this.trigger('prestep',dt);
      if(this.step) { this.step(dt); }
      this.trigger('step',dt);
      this.refreshMatrix();

      // Ugly coupling to stage - workaround?
      if(this.stage && this.children.length > 0) {
        this.stage.updateSprites(this.children,dt,true);
```

```
    }

    // Reset collisions if we're tracking them
    if(this.p.collisions) { this.p.collisions = []; }
  },
```

donde se lanza el evento `prestep`, se invoca el método `step`, se lanza el evento `step`, se actualiza la matriz de transformación de los sprites y se actualizan los sprites "hijos". La actualización de los hijos se hace a través del método `Stage.updateSprites` que básicamente recorre una lista de sprites e invoca sobre cada uno el método `update`.

Si redefinimos `update` debemos asegurarnos de, al menos, enviar los mensajes `step` y `prestep` que esperan los componentes.

Un ejemplo

Este sencillo ejemplo muestra una bola desplazándose en el aire con una trayectoria parabólica por efecto de la gravedad:

```
var Q = Quintus().include("Sprites").setup();

Q.Sprite.extend("Ball",{
  init:function(p) {
    this._super(p,{
      asset: "ball.png",
      x: 0,
      y: 300,
      vx: 50,
      vy: -400
    });
  },

  step: function(dt) {
    this.p.vy += dt * 9.8;

    this.p.x += this.p.vx * dt;
    this.p.y += this.p.vy * dt;
  }
});

Q.load(["ball.png"],function() {
  var ball = new Q.Ball();
  Q.gameLoop(function(dt) {
    ball.update(dt);
    Q.clear();
```

```
    ball.render(Q.ctx);
  });
});
```

Bucle principal y dibujado de un escenario

Si no hay otro bucle principal especificado, al cargar un escenario con `Q.stageScene` se establece este bucle principal:

```
  Q.stageGameLoop = function(dt) {
    var i,len,stage;

    if(dt < 0) { dt = 1.0/60; }
    if(dt > 1/15) { dt  = 1.0/15; }

    for(i =0,len=Q.stages.length;i<len;i++) {
      Q.activeStage = i;
      stage = Q.stage();
      if(stage) {
        stage.step(dt);
      }
    }

    if(Q.ctx) { Q.clear(); }

    for(i =0,len=Q.stages.length;i<len;i++) {
      Q.activeStage = i;
      stage = Q.stage();
      if(stage) {
        stage.render(Q.ctx);
      }
    }

    Q.activeStage = 0;

    if(Q.input && Q.ctx) { Q.input.drawCanvas(Q.ctx); }
  };
```

Que invoca a las funciones `step` y `render` de `Q.Stage`. Esta es la función que implementa el bucle principal `Q.Stage.step` :

```
    step:function(dt) {
      if(this.paused) { return false; }

      this.time += dt;
      this.markSprites(this.items,this.time);
```

```
    this.trigger("prestep",dt);
    this.updateSprites(this.items,dt);
    this.trigger("step",dt);

    if(this.removeList.length > 0) {
      for(var i=0,len=this.removeList.length;i<len;i++) {
        this.forceRemove(this.removeList[i]);
      }
      this.removeList.length = 0;
    }

    this.trigger('poststep',dt);
  },
```

que ejecuta `item.update(dt)` en cada elemento del escenario

```
  updateSprites: function(items,dt,isContainer) {
    var item;

    for(var i=0,len=items.length;i<len;i++) {
      item = items[i];
      // If set to visible only, don't step if set to visibleOnly
      if(!isContainer &&
        (item.p.visibleOnly &&
          (!item.mark || item.mark < this.time))) {
          continue;
      }

      if(isContainer || !item.container) {
        item.update(dt);
        Q._generateCollisionPoints(item);
        this.regrid(item);
      }
    }
  },
```

Esta es la función de dibujado de todos los elementos del escenario `Q.Stage.render`:

```
  render: function(ctx) {
    if(this.hidden) { return false; }
    if(this.options.sort) {
      this.items.sort(this.options.sort);
    }
    this.trigger("prerender",ctx);
    this.trigger("beforerender",ctx);
```

```
    for(var i=0,len=this.items.length;i<len;i++) {
      var item = this.items[i];
      // Don't render sprites with containers (sprites do that themselves)
      // Also don't render if not onscreen
      if(!item.container &&
        (item.p.renderAlways || item.mark >= this.time)) {
        item.render(ctx);
      }
    }
    this.trigger("render",ctx);
    this.trigger("postrender",ctx);
  }
});
```

Matrices de transformación

Cada sprite en Quintus lleva asociada una matriz de transformación que simplifica el dibujado y el cálculo de colisiones del sprite y además permite gestionar de forma incremental la información geométrica de las jerarquías de sprites.

```
Q.GameObject.extend("Sprite",{

  init: function(props,defaultProps) {
    this.p = Q._extend({
      x: 0,
      y: 0,
      z: 0,
      opacity: 1,
      angle: 0,
      frame: 0,
      type: Q.SPRITE_DEFAULT | Q.SPRITE_ACTIVE,
      name: '',
      spriteProperties: {}
    },defaultProps);

    this.matrix = new Q.Matrix2D();
    this.children = [];

    Q._extend(this.p,props);

    this.size();
    this.p.id = this.p.id || Q._uniqueId();

    this.refreshMatrix();
  },
```

. . .

Una matriz de transformación reune la información sobre las transformaciones geométricas que ha sufrido un sprite desde su, teórica, posición inicial centrada en 0,0. Las transformaciones básicas son:

- Traslación: cambios en la posición
- Rotación: cambios en la orientación
- Escalado: cambios en el tamaño

Los parámetros que definen una traslación son los que se muestran en la figura 11

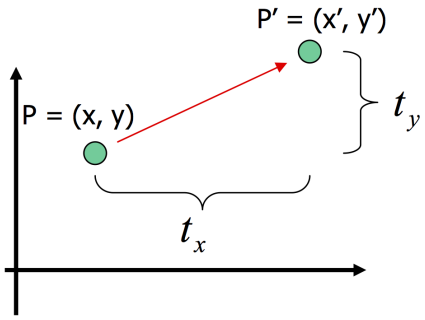

Figure 11: Traslación

La ecuación que permite calcular las nuevas coordenadas de cada punto después de la traslación es la que se muestra en la figura 12

$$P = (x, y) \qquad P' = (x', y') \qquad T = (t_x, t_y)$$

$$\boxed{P' = P + T}$$

Figure 12: Ecuación de la traslación

Los parámetros que definen una rotación con respecto al origen son los que se muestran en la figura 13

La ecuación que permite calcular las nuevas coordenadas de cada punto después de la rotación es la que se muestra en la figura 14

Muchas aplicaciones incluyen secuencias de transformaciones geométricas, por ejemplo si queremos generar animaciones que transformen la posición o el tamaño de un objeto entre fotogramas. De forma general cada transformación puede expresarse como una operación entre matrices de la forma:

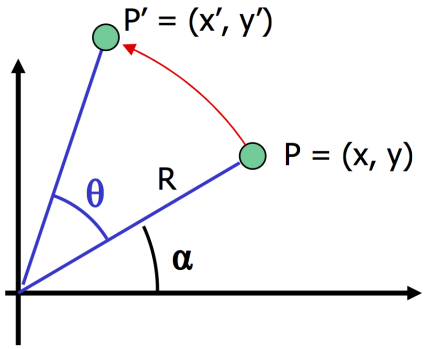

Figure 13: Rotación respecto al origen

$$P = (x, y) \qquad P' = (x', y') \qquad R = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

$$P' = P \cdot R$$

Figure 14: Ecuación de la rotación

$P' = PM_1 + M_2$

donde la matriz M_1 contiene la información de ángulos y factores de escala y la matriz M_2 contiene los términos de traslación asociados al punto fijo y al centro de rotación. Para producir una secuencia de transformaciones hay que calcular las nuevas coordenadas en cada transformación:

$P'' = P'M_3 + M_4 = ... = PM_1M_3 + M_2M_3 + M_4$

pero necesitamos una solución más eficiente que permita combinar las transformaciones para obtener directamente las coordenadas finales a partir de las iniciales. Para eliminar la matriz M_2 y poder representar cada transformación con una única matriz lo que se hace es pasar de matrices 2x2 a matrices 3x3, transformando cada punto `(x, y)` en otro con 3 componentes en *coordenadas homogéneas*.

Un punto `(x, y)` se representa en coordenadas homogéneas de la forma `(hx, hy, h)`, para cualquier h distinto de 0. Esto significa que un mismo punto tiene infinitas representaciones en coordenadas homogéneas, y así por ejemplo el punto `(4, 6)` puede expresarse como `(4, 6, 1)` `(8, 12, 2)` `(2, 3, 1/2)` `(8/3, 4, 2/3)`, es decir, el punto `(a,b,c)` en coordenadas homogéneas representa al punto `(a/c, b/c)` en dos dimensiones. Por simplicidad, lo habitual es tomar h=1, con lo que el punto `(x, y)` pasa a ser `(x, y, 1)`.

El uso de coordenadas homogéneas permite tratar todas las transformaciones geométricas como una multiplicación de matrices, como la composición de las transformaciones que se muestran en la figura 15

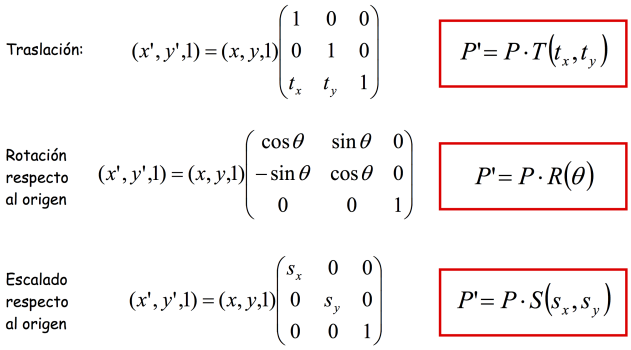

Figure 15: Transformaciones en coordenadas homogéneas

De esta forma, las nuevas coordenadas de la transformación se calculan simplemente multiplicando las matrices que las representan y aplicando esa matriz transformación a las coordenadas originales, ya sean traslaciones como se muestra en la figura 16

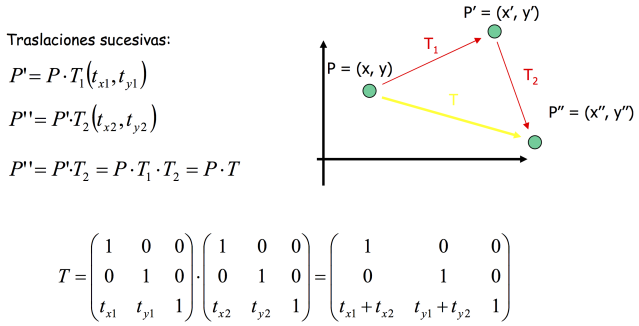

Figure 16: Composición de traslaciones

o rotaciones como se muestra en la figura 17

El objeto que se asigna a la propiedad `p.matrix` de un Sprite es una instancia de la clase `Matrix2D` que es la implementación de Quintus de las matrices de transformación en coordenadas homogéneas. Para aplicar las transformaciones al dibujado, Quintus se aprovecha de las transformaciones que implementa el elemento `canvas` de HTML 5.

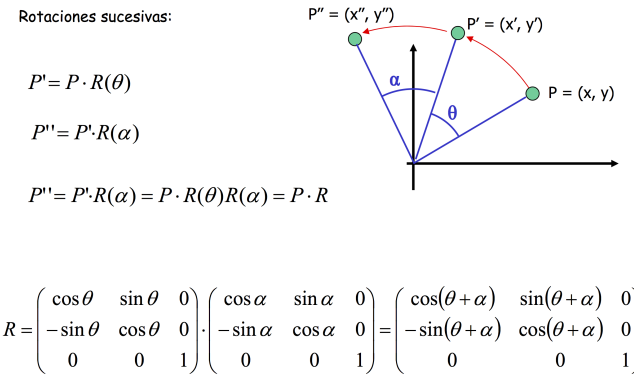

Figure 17: Composición de rotaciones

Transformaciones en el canvas

El canvas incluye las tres transformaciones básicas:

```
ctx.translate(x,y);

ctx.scale(sx,sy);

// Rotate takes an angle in radians
ctx.rotate(angle * Math.PI / 180);
```

y además permite aplicar una matriz de transformación arbitraria:

```
ctx.transform(a, b, c, d, e, f);
```

que se corresponde con la matriz:

```
a c e
b d f
0 0 1
```

lo que nos permite también volver a un estado donde no se aplica ninguna transformación:

```
ctx.setTransform(1,0,0,1,0,0);
```

que corresponde a la matriz

```
1 0 0
0 1 0
0 0 1
```

Es posible guardar un valor de la matriz de transformación del canvas para vover a restaurarla después:

```
ctx.save(); // Save the state

ctx.translate(...);
ctx.scale(...);
ctx.rotate(...);
...

ctx.restore(); // Restore the matrix
```

Lo que nos permite entender mejor el método `Sprite.render` que veíamos más arriba:

```
/**
     @method render
     @for Q.Sprite
     @param {Context2D} ctx - context to render to
    */
    render: function(ctx) {
      var p = this.p;

      if(p.hidden) { return; }
      if(!ctx) { ctx = Q.ctx; }

      this.trigger('predraw',ctx);

      ctx.save();

        if(this.p.opacity !== void 0 && this.p.opacity !== 1) {
          ctx.globalAlpha = this.p.opacity;
        }

        this.matrix.setContextTransform(ctx);

        if(this.p.flip) { ctx.scale.apply(ctx,this._flipArgs[this.p.flip]); }

        this.trigger('beforedraw',ctx);
        this.draw(ctx);
        this.trigger('draw',ctx);

      ctx.restore();

      // Children set up their own complete matrix
      // from the base stage matrix
      if(this.p.sort) { this.children.sort(this._sortChild); }
      Q._invoke(this.children,"render",ctx);

      this.trigger('postdraw',ctx);
```

```
      if(Q.debug) { this.debugRender(ctx); }

    },
```

Las transformaciones no son conmutativas

Es importante tener presente que las transformaciones en general no son conmutativas, como se puede ver ejecutando este ejemplo:

```
<html>
<head>
  <style>
    canvas { background-color: lightgrey;}
  </style>
</head>
<body>
  <canvas id="pruebas" width='400' height='400'></canvas>

  <script>
    var ctx = document.getElementById('pruebas').getContext('2d');
    ctx.fillRect(0,0,100,100);
    // Move the object to the correct spot
    ctx.translate(250,200);
    // Uncenter the element back to its original spot
    ctx.translate(50,50);
    // Rotate it
    ctx.rotate(45 * Math.PI / 180);
    // Center it
    ctx.translate(-50,-50);
    // Draw it
    ctx.fillRect(0,0,100,100);
    ctx.setTransform(1,0,0,1,0,0);
    ctx.fillRect(200,0,100,100);
    // Rotate it
    ctx.rotate(45 * Math.PI / 180);
    // Move the object to the correct spot
    ctx.translate(250,200);
    ctx.fillRect(0,0,100,100);
  </script>
</body>
</html>
```

que genera el resultado de la figura 18

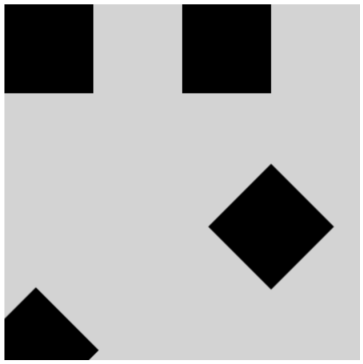

Figure 18: Las transformaciones no son conmutativas

Detección de colisiones

El código de gestión de colisiones se reparte entre las clases `Sprite` y `Stage`. `Sprite` mantiene la lista de puntos que definen el perímetro de colisión de la entidad (`p.points`), sobre la que se aplica la matriz de transformación, y la clase `Stage` gestiona colecciones de sprites y es capaz de determinar si un sprite dado colisiona con otros, a través de las métodos `Stage.collide` y `Stage.search`, realizando algunas optimizaciones en el proceso, como definir una rejilla y sólo hacer comprobaciones más finas si los dos objetos están en la misma cuadrícula.

- `Stage.collide`, genera el evento hit sobre el objeto que recibe como parámetro, si este colisiona con otro sprite de la escena, y
- `Stage.search`, devuelve la primera colisión que encuentra entre el objeto que recibe como parámetro y un sprite de la escena, si lo hay.

En el ejemplo http://www.html5quintus.com/docs/collision.html de la documentación de Quintus se puede ver el mecanismo de detección de colisiones en acción, como se muestra en la figura 19

donde se muestran tanto los cuadrados delimitadores, calculados a partir del ancho y el alto, como los perímetros de colisión definidos por la lista de puntos `p.points` de cada sprite.

Por eficiencia, podemos controlar el tipo de entidades con las que nos interesa que se calculen las posibles colisiones, utilizando máscaras binarias.

- En la propiedad `p.type` del sprite se almacena el tipo de un objeto desde el punto de vista de las colisiones, por defecto `Q.SPRITEDEFAULT | Q.SPRITEACTIVE`, y
- en la propiedad `p.collisionMask` del sprite se almacena el tipo de objetos con los que puede colisionar. Si no le damos valor a la propiedad `p.collisionMask` entonces el sprite colisionará con todos los tipos de sprites.

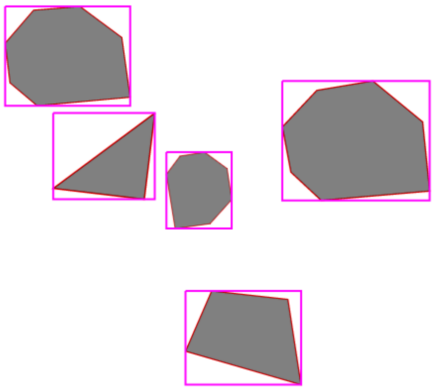

Figure 19: Demo de colisiones

En el módulo `Sprites` se definen una serie de constantes que podemos usar para controlar el tipo y la máscara de colisiones de un sprite:

```
Q.SPRITE_NONE     = 0;
Q.SPRITE_DEFAULT  = 1;
Q.SPRITE_PARTICLE = 2;
Q.SPRITE_ACTIVE   = 4;
Q.SPRITE_FRIENDLY = 8;
Q.SPRITE_ENEMY    = 16;
Q.SPRITE_UI       = 32;
Q.SPRITE_ALL   = 0xFFFF;
```

Si queremos que se detecten las colisiones de un sprite, entonces en el método `step` del sprite debemos invocar el método `collide` del escenario donde está incluido, pasándole el sprite como parámetro. Si hay colisión, el sprite recibirá el evento `hit` y en la retrollamada que responde al evento se pasa información geométrica sobre la colisión. Por ejemplo, podemos escribir el código para que un cierto tipo de sprite evite las colisiones:

```
Q.sprite.extend("MySprite",{
  init: function() {
    // Listen for hit event and call the collision method
    this.on("hit",this,"collision");

  },

  collision: function(col) {
    // .. do anything custom you need to do ..

    // Move the sprite away from the collision
    this.p.x -= col.separate[0];
```

```
    this.p.y -= col.separate[1];
  },

  step: function(dt) {
    // Tell the stage to run collisions on this sprite
    this.stage.collide(this);
  }
});
```

Este es precisamente el comportamiento que implementa el componente `2d`.

Esta es la información que se pasa en el objeto de colisión que recibe la retrollamada que proceso el evento `hit`:

```
sprite.on("hit",function(col) {
  col.normalX; // normal of the collision, x direction
  col.normalY; // normal of the collision, y direction
  col.obj; // Object we collided with
  col.distance; // Distance we had to move to resolve the collision
  col.separate[0]; // normalX multiplied by distance
  col.separate[1]; // normalY multiplied by distance
});
```

Implementación de la detección de colisiones

La gestión de colisiones se divide entre unas funciones auxiliares que implementan el método SAT de detección de colisiones y las funciones que se incluyen en la propia clase `Q.Stage`

El metodo SAT (Separating Axis Theorem) para determinar las colisiones básicamente se basa en la idea de que si es posible dibujar una línea que separe los dos polígonos entonces estos no colisionan, como se muestra en la figura 20

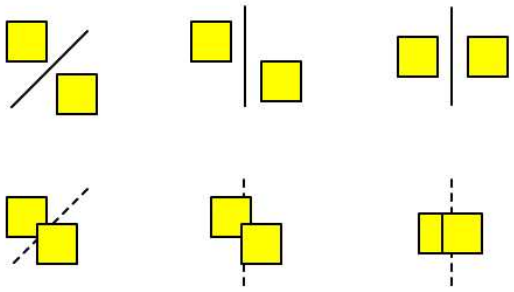

Figure 20: Detección de colisiones

La idea del algoritmo se basa en generar distintas proyecciones de las figuras y comprobar si hay intersección entre las proyecciones, como se muestra en la figura 21

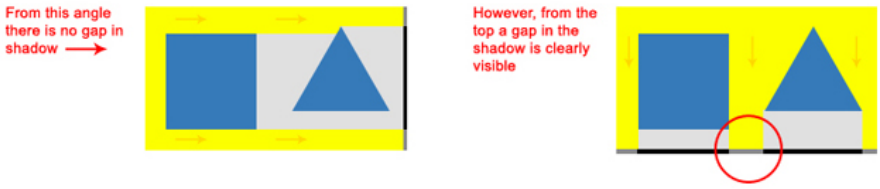

Figure 21: Idea del algoritmo

Si proyectando desde todas las caras no encontramos ninguna proyección que no solape entonces las figuras están colisionando. En cambio, en cuanto encontremos una proyección donde las proyecciones estén separadas ya podemos concluir que las figuras no intersectan.

El algoritmo, en seudocódigo, se aplica a cada uno de los lados de uno de los polígonos:

1. La recta sobre la que calculamos la proyección es la normal al lado elegido, como se muestra en la figura 22

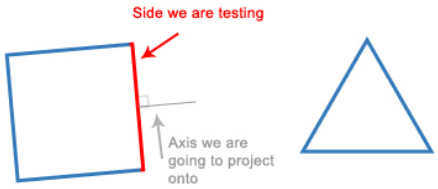

Figure 22: SAT: paso 1

2. Iteramos sobre cada punto del primer polígono y lo proyectamos sobre la recta, guardando los valores máximo y mínimo como se muestra en la figura 23
3. Se hace lo mismo con los puntos del segundo polígono, con los resultados que se muestran en la figura 24
4. Se comprueba si las proyecciones intersectan, como se muestra en la figura 25

Si encontramos una proyección que no da lugar a intersección ya podemos concluir que no colisionan, o de lo contrario tenemos que iterar sobre todos

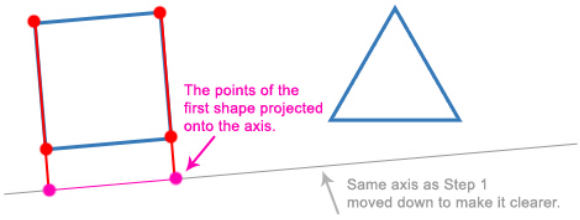

Figure 23: SAT: paso 2

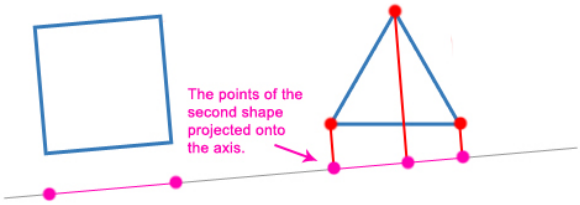

Figure 24: SAT: paso 3

los lados de los dos polígonos para concluir que sí colisionan porque no hemos encontrado ninguna proyección que no dé lugar a intersecciones.

Un resultado adicional del algoritmo es que nos permite encontrar, en el que caso de que colisionen, el desplazamiento mínimo que se ha de realizar para que dejen de hacerlo: es el que viene dado por la proyección que genera la mínima intersección.

El algoritmo también es aplicable para determinar si un polígono colisiona con un círculo. La recta de proyección que define el círculo es la que une el centro del

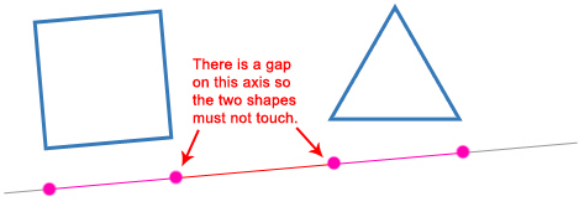

Figure 25: SAT: paso 4

mismo con el vértice más cercano del polígono, como se muestra en la figura 26

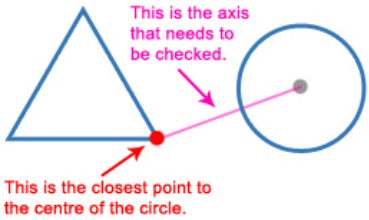

Figure 26: SAT: intersección con un círculo

Inconvenientes del método SAT:

- Sólo funciona con figuras convexas (todos los ángulos son menores de 180 grados y todas las diagonales interiores), con lo que si queremos aplicarlo a figuras más complejas, tendremos que separarlas en fragmentos convexos
- No nos dice cuáles son los lados que se están tocando

En Quintus la detección de colisiones se optimiza dividiendo el escenario con una rejilla de forma que sólo se comprueba si dos objetos colisionan cuando al menos en parte están en la misma celda de la rejilla.

Referencias

- http://html5quintus.com/guide/intro.md
- http://html5quintus.com/api/
- Juego de plataformas en Quintus