DESARROLLO DE UN JUEGO EN HTML5 CON QUINTUS

OBJETIVO

El objetivo de esta presentación es enseñar paso a paso cómo se crea un juego sencillo de plataformas 2D con Quintus.

QUÉ ES QUINTUS

Quintus es un conjunto de herramientas que nos permite programar de forma bastante sencilla videojuegos que se ejecutarán en un navegador de internet de cualquier dispositivo con HTML5. Utiliza el lenguaje de programación web JavaScript para crear una serie de librerías que contienen todos los métodos y funciones que utilizaremos para hacer el juego.

INICIO

Crear una carpeta en un servidor web (como apache), llamada, por ejemplo, "juego". Dentro, crear las siguientes subcarpetas y archivo:

juego/index.html ← El archivo html que contendrá el juego en sí

juego/data/ ← Ahí irá el mapa que crearemos

juego/lib/ ← Meter ahí las librerías de Quintus

juego/images/ ←Ahí irán las imágenes (archivos PNG) que servirán de base para el mapa y los personajes

EL MAPA

Lo primero de todo es crear un mapa donde se desarrollará la acción de nuestro juego:

El mapa será un escenario en 2D, **formado por bloques o patrones**, llamados "**tiles**".

¿Qué es un tile? Cada uno de los bloques que pueden formar un mapa → **Enseñar el archivo PNG con cada tile para mostrarlo claramente.**

Para ello usaremos un programa llamado **Tiled**, que crea mapas en formato TMX que pueden ser cargados por Quintus. La forma de crear los mapas es dibujándolos con el ratón.

En la pantalla de Tiled, pinchamos en **New Map**:

Orientación: ortogonal. Isométrica nos proporcionaría una perspectiva isométrica, útil en juegos de rol de tablero, por ejemplo, pero no para un juego de plataformas 2D que queremos hacer.

Tile Layer format: Formato de la capa de patrones, **será XML** porque es el que puede cargar Quintus.

Tile Render order: Simplemente elegimos Right Down

Map Size: Es el tamaño del mapa que crearemos en número de tiles. Elegimos el que nos interese (40x10 = 2800x700 pixel en nuestro ejemplo).

Tile Size: el tamaño de cada tile.

Ahora es necesario darle al programa las imágenes que contienen los tiles para dibujar nuestro mapa. Estos archivos suelen llamarse "tileset" y muchos se pueden descargar gratuitamente de **OpenGameArt.org**. Nuestro tileset debe estar en /images/ así que lo cargamos de ahí:

Map -> New Tileset -> Browse y cargamos el tileset de la carpeta /images/.

En Tile Width y height ponemos la misma anchura y altura que hemos elegido en Tile size al crear el mapa hace un momento. La separación debe ser 0 pixel.

IMPORTANTE: En la versión actual de Quintus **no debe haber separación entre los tiles** y estos **deben comenzar en el primer pixel de la imagen y acabar en el último** → Mostrar con Paint.

Ahora podemos pintar un mapa seleccionando un tile y pintando con el ratón, o rellenando toda la pantalla con un tile, como en Paint de Windows.

Nuestro juego de plataformas tendrá dos capas: una capa que será el fondo y solo está para decorar y otra capa de colisión que será lo que soporte al personaje y lo que interaccionará con el personaje. Creamos capas con el icono de crear en la zona de layers. A la capa que queremos usar de fondo la ponemos debajo del todo y la llamamos "background" y la capa de colisión debe estar encima de la de fondo. Ahora podemos pintar.

INICIALIZAR QUINTUS Y CARGAR LOS RECURSOS

Con nuestro mapa pintado lo siguiente es empezar a rellenar el archivo index.html

Ponemos esto en index.html:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script src='lib/quintus.js'></script>
        <script src='lib/quintus_sprites.js'></script>
        <script src='lib/quintus_scenes.js'></script>
        <script src='lib/quintus_input.js'></script>
```

```
        <script src='lib/quintus_anim.js'></script>
        <script src='lib/quintus_2d.js'></script>
        <script src='lib/quintus_touch.js'></script>
        <script src='lib/quintus_ui.js'></script>
    </head>
    <body style="background-color: black;">
        <script>
            var Q = Quintus()
                .include("Sprites, Scenes, Input, 2D, Touch, UI")
                .setup({
                    maximize: true,
                    scaleToFit: true
                }).controls().touch();
        </script>
    </body>
</html>
```

Como vemos lo que hemos hecho ha sido referenciar las librerías de Quintus que vamos a utilizar en el archivo html, para que sepa en todo momento donde están y podamos hacer uso de ellas.

Quintus.js: Es el núcleo, proporciona un esqueleto sobre el que se apoyarán el resto de librerías. Proporciona soporte para clases, eventos, componentes, etc

Dentro del body , entre <script> comenzamos a escribir el verdadero código del juego, que va en javascript.

Con **var Q = Quintus() creamos una nueva instancia de Quintus**. Esto es crear una variable de tipo Quintus, es decir, al crear esa variable inicializamos todo el motor del juego y tenemos así acceso a sus características como cargar mapas. Esa variable es como la puerta a Quintus.

Con include escogemos los módulos con los que vamos a trabajar. El módulo input nos proporciona los controles del juego, es decir, que cuando pulsemos en la flecha izquierda el muñeco se mueva a la izquierda, etc. El módulo 2D aporta características básicas de un juego en 2D como gravedad. Touch es para que se pueda jugar desde móviles o tablets con pantallas táctiles. Sprites para que soporte imágenes.

Si escribimos Q.setup sin nada dentro creamos un marco por defecto de 320 por 420 pixeles.

El siguiente paso es **cargar nuestros recursos**, tales como imágenes de patrones o el mapa que hemos creado. Esto se hace con **Q.load**.

Con **Q.sheet creamos la "hoja" (spritesheet)** con el mapa que hemos hecho donde tendrá lugar nuestro juego.

Q.stageScene nos crea un escenario o "stage", donde pondremos una escena. En un juego, puede haber diferentes stages. **Los stages pueden ser entendidos como un contenedor donde metemos una serie de sprites que interaccionan entre ellos**. Entonces, en un contenedor podría tener el juego propiamente dicho y en otro, un marcador, donde tenga información

sobre los puntos que llevo, o las vidas que me quedan. Los sprites que haya en un stage no interaccionarán nunca con los que haya en otro stage. Aquí solo tendremos un escenario.

Por otro lado, **una escena es todo lo que ocurre en un escenario**.

```
//load assets
Q.load("tiles_map.png, player.png, enemy.png, coinGold.png, level1.tmx", function() {
  Q.sheet("tiles","tiles_map.png", { tilew: 70, tileh: 70});
  Q.stageScene("level1");
});
```

EL NIVEL

Tras esto hemos creado un escenario llamado level1. Lo siguiente es iniciar la escena, es decir, definir qué harán cada uno de los elementos que habrá en la escena, tales como el mapa que hemos creado, el jugador, o los enemigos.

```
Q.scene("level1",function(stage) {
    var background = new Q.TileLayer({ dataAsset: 'level1.tmx', layerIndex: 0, sheet:
'tiles', tileW: 70, tileH: 70, type: Q.SPRITE_NONE });
    stage.insert(background);
    stage.collisionLayer(new Q.TileLayer({ dataAsset: 'level1.tmx', layerIndex:1,  sh
eet: 'tiles', tileW: 70, tileH: 70 }));
});
```

Aquí **estamos creando dos objetos de la clase TileLAyer**. Uno de ellos será el fondo y otro la capa de colisión. Empezamos definiendo el fondo y la capa de colisión. Hacemos referencia a nuestro mapa, el archivo tmx, y al layerIndex, que es el orden de la capa en el mapa de Tiled.

Con **type: Q.SPRITE_NONE** indicamos que la capa fondo no colisionará con nuestro jugador.

Ahora podemos cargar la página en el navegador y ver el mapa.

EL JUGADOR

Lo primero de todo antes de añadir un jugador, es crear una clase para el jugador:

```
Q.Sprite.extend("Player",{
    init: function(p) {
      this._super(p, { asset: "player.png", x: 110, y: 595, jumpSpeed: -380});
      this.add('2d, platformerControls');
    },
    step: function(dt) {
        if(Q.inputs['left'] && this.p.direction == 'right') {
            this.p.flip = 'x';
        }
        if(Q.inputs['right']  && this.p.direction == 'left') {
            this.p.flip = false;
        }
    }
```

```
});
```

Con **extend** creamos una **subclase** de la **clase Sprite**, que llamaremos jugador y que dotaremos de las características que nos interesen para nuestro jugador de plataformas 2D.

El constructor de nuestra subclase comienza ejecutando el constructor de la clase padre, lo que se hace con **this._super** con algunos parámetros como el archivo de imagen del jugador, la posición inicial que tendrá dentro del mapa o la velocidad de salto que queramos (si no especificamos ninguna tendrá una por defecto.

Las clases pueden tener sus propios componentes, que no son más que un conjunto de métodos y propiedades que proporciona Quintus, que es lo que hacemos en this.add. El componente 2d proporciona las físicas básicas (gravedad, aceleración, velocidad) y la detección de colisión. El componente platformercontrol nos permite mover al jugador con el teclado.

El método step nos permite acceder al bucle del juego. **En un juego hay instrucciones que se ejecutan muchas veces por segundo, se necesita pues un bucle que se ejecute muy frecuentemente.** Lo que introduzcamos dentro de ese bucle será ejecutado cada vez que se ejecute el bucle. Nosotros hemos introducido unas instrucciones para que el muñeco del jugador se gire cuando sea necesario.

Una vez creada la clase jugador, es necesario crear un objeto de la clase jugador. Introducimos este código en la inicialización de la escena.

```
var player = stage.insert(new Q.Player());
stage.add("viewport").follow(player,{x: true, y: true},{minX: 0, maxX: background.p.w
, minY: 0, maxY: background.p.h});
```

Con **viewport** hacemos que la cámara siga al jugador.

Ahora si cargamos la página veremos nuestro jugador y podremos controlarlo.

LOS ENEMIGOS

De la misma manera que hemos creado una clase jugador, tenemos que crear una clase enemigo.

```
Q.Sprite.extend("Enemy", {
        init: function(p) {
            this._super(p, {asset: "enemy.png", vx: -100, visibleOnly: true});
            this.add('2d, aiBounce');
```

```
        //to make the enemy flip when bumps with a wall
        //default direction is left
        this.on("bump.left", function(collision){
          this.p.flip = 'x';
        });
        this.on("bump.right", function(collision){
          if(this.p.direction == 'right') {
          this.p.flip = 'x';
          }
        });
    }
});
```

Con vx elegimos la velocidad de los enemigos. Los enemigos usan **aiBounce** para cambiar de dirección cuando tocan una pared.

Lo que hay en el al final es para hacer que los enemigos se giren cuando chocan con algo.

Nuevamente, tras crear la clase creamos tantos objetos de la clase como enemigos queramos dentro de la inicialización de la escena, indicando la coordenada x (en pixeles) donde queremos que aparezcan.

```
var enemy = stage.insert(new Q.Enemy({x: 600, y: 650}));
var enemy = stage.insert(new Q.Enemy({x: 1000, y: 650}));
var enemy = stage.insert(new Q.Enemy({x: 2000, y: 650}));
```

LOS DAÑOS

Ahora es el momento de añadirle propiedades a los enemigos para que cuando me toquen me maten, o para que cuando yo los pise desde arriba se mueran.

Añadimos esto dentro del constructor de los enemigos

```
//Escucha una colision por los lados o por debajo, si colisiona con un jugador, termina el juego.
        this.on("bump.left,bump.right,bump.bottom",function(collision) {
          if(collision.obj.isA("Player")) {
```

```
            Q.stageScene("endGame",1, { label: "Has muerto :( " });
            collision.obj.destroy();
            }
        });
        //Si el jugador salta encima del enemigo, desturimos el objeto enemigo y recibimos un ligero salto
        this.on("bump.top",function(collision) {
          if(collision.obj.isA("Player")) {
            this.destroy();
            collision.obj.p.vy = -300;
          }
        });
```

Lo que estamos indicando es que cuando un enemigo toque a nuestro jugador por los lados o por debajo, muestre una etiqueta que diga que estás muerto, destruya el objeto jugador y pase al estado de fin de juego, que definiremos más tarde.

Ahora bien, si el jugador salta sobre el enemigo, destruya el objeto enemigo y nos da un ligero salto.

EL FIN DEL JUEGO

Lo último es añadir qué eventos producirán un final en el juego. En principio solo dos: cuando nos matan y cuando alcanzamos nuestro objetivo (como recoger una moneda al final de la fase).

Añadimos así una escena llamada endGame:

```
// 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: "Jugar otra vez" }))
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
container.fit(20);
});
```

Para terminar del todo, hay que añadir el objetivo. En nuestro caso será una moneda de oro. Creamos la clase moneda de oro como una subclase de la clase sprite, creamos objeto de esta clase y hacemos que se lance la escena fin del juego cuando el jugador la toque.

Creacion de la clase:

```
Q.Sprite.extend("CoinGold", {
            init: function(p) {
               this._super(p, {asset: "coinGold.png"});
            }
          });
```

Creación del objeto:

```
stage.insert(new Q.CoinGold({ x: 2600, y: 595 }));
```

Modificamos las propiedades de la clase jugador para que acabe el juego cuando toque la moneda:

```
// 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 Coin Gold, you win!
                if(collision.obj.isA("CoinGold")) {
                   // Stage the endGame scene above the current stage
                   Q.stageScene("endGame",1, { label: "¡Ganaste!   " });
                   // Remove the player to prevent them from moving
                   this.destroy();
                    collision.obj.destroy();
                }
                });
```

Así también el jugador y la monea desaparecen cuando se tocan.

BONUS

Añadir sonidos

1. Cargar la librería audio **<script src='lib/quintus_audio.js'></script>**
2. Permitir sonido **Q.enableSound();**
3. Incluir audio al crear la instancia de Quintus: include("Sprites, Scenes, Input, 2D, Touch, UI, **Audio**")
4. **Crear una carpeta audio en mi directorio con los sonidos que quiera y cargar estos sonidos en load assets**

Pondremos un sonido cuando matemos a los enemigos y otro cuando alcancemos el objetivo.

Escribimos esto **Q.audio.play('hit.mp3');** dentro de:

```
this.on("bump.top",function(collision) {
                  if(collision.obj.isA("Player")) {
                     this.destroy();
                     collision.obj.p.vy = -300;
                     Q.audio.play('hit.mp3');
                  }
```

```
});
```

Cuando toquemos la moneda, escribirmos **Q.audio.play('coin.mp3'); dentro de:**

```
if(collision.obj.isA("Player")) {
                    // Stage the endGame scene above the current stage
                    Q.stageScene("endGame",1, { label: "¡Ganaste!   " });
                    // Remove the player to prevent them from moving
                    this.destroy();
                    collision.obj.destroy();
                    Q.audio.play('coin.mp3');
}
```

Añadir más fases

El juego está terminado. Si quisiéramos añadir más fases, cada vez que alcanzamos el objetivo habría que añadir una nueva escena de transición, por ejemplo, llamada toLevel2. En esa escena pondríamos un texto de “Ir al nivel 2” y un botón de aceptar, que cuando lo pulsáramos, nos llevara a la escena level2, como en la escena fin de juego, que cuando pulsamos el botón de volver a jugar, nos lleva a la escena level1.

```
Q.scene('toLevel2',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: "Go to Level 2" }))
        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('level2');

    });


    // Expand the container to visibily fit it's contents

    container.fit(20);

});
```

Bibliografía y fuentes:

- Repositorio de Quintus en GitHub: https://github.com/cykod/Quintus
- Documentación de Quintus: http://www.html5quintus.com/documentation#.VE142vl5OSo
- Tutorial clon de Super Mario con Quintus en Zenva Academy: http://www.gamedevacademy.org/create-a-html5-mario-style-platformer-game/
- Curso de Quintus en Zenva Academy: https://academy.zenva.com/lesson/course-intro-9/