Proyecto final de carrera

Flashpunk: videojuegos en Flash

Ingeniería Técnica en Informática de Sistemas
Diciembre 2011

Autor: Javier Díez Matilla
Director: Lluís Solano Albajes
Departamento del director: LSI
Centro: Facultad de informática de Barcelona (FIB)
Universidad: Universidad Politécnica de Cataluña (UPC) BarcelonaTech

Índice

		Página
1 -	Introducción	1
	Objetivos y descripción del proyecto	1
	¿Qué son Flash y ActionScript?	2
	¿Qué es XML?	3
	¿Qué es Flashpunk?	4
	Factibilidad del proyecto	6
2 -	Más sobre Flashpunk: ¿Quién y cómo lo ha hecho?	7
	Quién	7
	Cómo	8
3 -	Puesta a punto del entorno	9
4 -	Análisis: paquetes y clases	15
	net.flashpunk	15
	net.flashpunk.debug	17
	net.flashpunk.graphics	17
	net.flashpunk.masks	19
	net.flashpunk.tweens.misc	21
	net.flashpunk.tweens.motion	22
	net.flashpunk.tweens.sound	23
	net.flashpunk.utils	23
5 -	Análisis: pequeño ejemplo práctico	25
	La idea	25
	El cubo	27
	El nivel	29
	La interfaz de usuario	30
	Las casillas	32
	El jugador	33
	Resto de clases	35
	Esquema	36
6 -	Análisis económico y planificación	38
	Análisis económico	38
	Planificación	39
7 -	Posibilidades de ampliación	42
8 -	Conclusión	43
9 -	Bibliografía	45
10 -	Imágenes	46

1 – Introducción

Objetivos y descripción del proyecto

Los objetivos del proyecto son analizar y evaluar la librería **Flashpunk** para la creación de videojuegos en tecnología **Flash**, así como la creación de un pequeño ejemplo a modo de demostración de las capacidades de la librería.

En el análisis de la librería se hará un resumen a modo de revisión de las clases que ésta incorpora y se discutirá su utilidad de cara a la creación de un videojuego.

Como segunda parte del análisis, se comprobarán las capacidades de la librería mediante la realización del pequeño ejemplo antes mencionado. Se procederá a la creación de un videojuego utilizando la librería **Flashpunk**, solamente ésta y ninguna otra ya que el objetivo es comprobar lo que se puede llegar a hacer con ella con la menor ayuda externa posible.

En la creación de dicho videojuego se pondrá especial atención en la programación del mismo y se dejarán en un segundo plano los apartados gráfico y sonoro aunque se procurará cuidarlos lo máximo que el tiempo restante permita.

También se hará un breve análisis económico para así comprobar, junto con el resultado obtenido, si la calidad obtenida merece la inversión realizada.

Así pues, se dictará un veredicto sobre la viabilidad de **Flashpunk** como medio para la creación de videojuegos.

¿Qué son Flash y ActionScript?

Flash es un programa de edición multimedia desarrollado originalmente por **Macromedia** (ahora parte de **Adobe**) que utiliza principalmente gráficos vectoriales[1], pero también imágenes rasterizadas[2], sonido, código de programa, flujo de vídeo y audio bidireccional para crear proyectos multimedia. **Flash** es el entorno desarrollador y **Flash Player** es el programa (la máquina virtual) utilizado para ejecutar los archivos generados con **Flash**.

Los proyectos multimedia pueden ser desde simples animaciones hasta complejos programas pues, además de los gráficos, vídeos y sonidos, **Flash** incorpora **ActionScript**, un completo lenguaje de programación orientado a objetos que expande enormemente las posibilidades en los proyectos.

Los archivos de Flash suelen tener la extensión **.SWF** y aparecen frecuentemente en páginas web en forma de animaciones y aplicaciones. Éstas pueden permitir interacción con el usuario y además existe una alta disponibilidad de Flash ya que prácticamente todos los navegadores web incluyen el *plug-in*[3] para soportar la ejecución de animaciones **Flash**. En caso de no soportarlo, la instalación de dicho *plug-in* es sumamente sencilla e incluso a veces se hace de forma automática.

Breve historia de **Flash**:

Originalmente **Flash** no fue un desarrollo propio de **Adobe**, sino de una pequeña empresa de desarrollo de nombre **FutureWave Software** y su nombre original fue **FutureSplash Animator**. En diciembre de 1996 **Macromedia** adquiere **FutureWave Software**, y con ello su programa de animación vectorial que pasa a ser conocido como **Flash 1.0**.

En 2005 **Adobe** compra **Macromedia** y junto con ella sus productos, entre ellos **Flash**, que pasa a llamarse **Adobe Flash**.

[1] Una imagen vectorial es una imagen digital formada por objetos geométricos independientes (segmentos, polígonos, arcos, etc.), cada uno de ellos definido por distintos atributos matemáticos de forma, de posición, de color, etc. Por ejemplo un círculo de color rojo quedaría definido por la posición de su centro, su radio, el grosor de línea y su color.

[2] La rasterización es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como una pantalla de computadora, una impresora electrónica o una imagen de mapa de bits (*bitmap*).

[3] Un *plug-in* o complemento es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica.

¿Qué es XML?

XML, siglas en inglés de e**X**tensible **M**arkup **L**anguage ('lenguaje de marcas[4] extensible'), es un metalenguaje[5] extensible de etiquetas desarrollado por el **World Wide Web Consortium**[6] (W3C). Por lo tanto **XML** no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades.

XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

XML es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

Ejemplo de documento **XML**:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<libro>
        <titulo></titulo>
        <capitulo>
                <titulo></titulo>
                <seccion>
                        <titulo></titulo>
                </seccion>
        </capitulo>
</libro>
```

[4] Un lenguaje de marcado o lenguaje de marcas es una forma de codificar un documento que, junto con el texto, incorpora etiquetas o marcas que contienen información adicional acerca de la estructura del texto o su presentación. Los lenguajes de marcado suelen confundirse con lenguajes de programación. Sin embargo, no son lo mismo, ya que el lenguaje de marcado no tiene funciones aritméticas o variables, como sí poseen los lenguajes de programación.

[5] Un metalenguaje es un lenguaje que se usa para hablar acerca de otro lenguaje.

[6] El Consorcio World Wide Web (W3C) es una comunidad internacional donde las organizaciones Miembro, personal a tiempo completo y el público en general trabajan conjuntamente para desarrollar estándares Web. Liderado por el inventor de la Web Tim Berners-Lee y el Director Ejecutivo (CEO) Jeffrey Jaffe, la misión del W3C es guiar la Web hacia su máximo potencial.

¿Qué es Flashpunk?

Flashpunk es una librería gratuita de **ActionScript 3** diseñada para desarrollar videojuegos 2D en **Flash**. Dicha librería proporciona un entorno de trabajo directo, rápido, práctico y claro para desarrollar nuestros juegos **Flash** ya que el trabajo sucio ha sido realizado por nosotros para que nos podamos concentrar directamente en las etapas de diseño y test de nuestros juegos. Así pues, no tendremos que programar funciones que controlen el tiempo, la entrada mediante el teclado o las colisiones entre objetos por ejemplo, porque ya están implementadas y funcionando.

1. Logo de Flashpunk (fuente: página web de Flashpunk)

Como diferencia al desarrollo de videojuegos **Flash** en el entorno original de **Adobe Flash**, en **Flashpunk** trabajaremos con mapas de bits en lugar de gráficos vectoriales. Es decir, en vez de trabajar con vectores de **Flash** como *sprites* lo haremos con imágenes importadas de archivos PNG, JPEG o GIF.

Algunas características de **Flashpunk** son:

- Soporte para juegos que utilicen un *framerate* fijo o que sean independientes al control de *frames* por segundo.
- Sistema de colisiones por rectángulos, píxeles o mallas rápido y manejable.
- Clases intuitivas para animaciones, textos, fondos, mosaicos de imágenes (*tilemaps*), etc.
- Consola para depurar (*debug*) en tiempo real que nos dará información.
- Soporte para efectos de sonido con volumen, dirección y desvanecimiento.
- Interpolación de movimiento para movimientos lineales, curvados y basados en recorridos.
- Fácil manejo de la profundidad.
- Control simple de la entrada mediante teclado y ratón.
- Efectos y emisores de partículas rápidos y eficientes.

El entorno de trabajo de **Flashpunk** está diseñado para ser usado con **Flex**, utilizado para crear aplicaciones **Flash**. **Flex** es propiedad de **Adobe** (hasta 2005 lo fue de **Macromedia**) y es de descarga gratuita y *open source*. Con él podremos, como ya se ha

dicho, crear aplicaciones web en **Flash** con la ventaja de no necesitar el entorno de desarrollo de **Adobe Flash** cuya licencia es muy costosa.

2. Logos de Adobe Flash y Flex (fuente: Google imágenes)

Para comenzar a usar **Flashpunk** y **Flex** sólo faltaría un editor de código con el cual programar. Editores de código hay muchos pero hay unos especialmente diseñados para trabajar con **ActionScript** como **FlashDevelop** o **FlashBuilder**. En estos editores podemos importar la librería **Flashpunk** y crear juegos sin la necesidad del entorno de desarrollo oficial de **Adobe Flash**, como se ha comentado ya. El editor que usemos se ha de configurar para utilizar **Flex**, en su configuración se le ha de indicar el directorio donde están los archivos de **Flex**, necesarios para que el editor pueda crear archivos de **Flash** SWF.

3. Logo de Flashdevelop (fuente: página web de Flashdevelop)

Factibilidad del proyecto

El proyecto puede llevarse a cabo sin demasiados problemas ya que, como veremos más adelante, no requiere un despliegue específico de capital ni de personal.

Los objetivos del proyecto son el análisis de **Flashpunk** y la realización de un pequeño juego.

Para el primero basta con tener un poco de tiempo y motivación. Gracias a la documentación disponible (en línea y para descarga) nos podremos hacer una idea de qué tenemos disponible y qué podremos hacer con ello. Más importante aún será experimentar por nuestra cuenta. Podemos hacerlo con programas como por ejemplo **FlashDevelop**, en el cual podemos programar habiendo importado fácilmente la librería **Flashpunk** disponiendo así de sus clases y funciones. Mientras experimentamos programando, algo imprescindible será consultar la documentación y los recursos disponibles como el foro de la misma web de **Flashpunk** que en más de una ocasión nos serán de gran ayuda.

Para el segundo objetivo, la realización de un pequeño juego, necesitaremos algo más que tiempo y motivación (aunque es lo verdaderamente importante). Necesitaremos recursos tales como gráficos, sonidos, buenas ideas y conocimientos de programación orientada a objetos. Tales recursos pueden obtenerse gratuitamente en internet o pueden proceder de la elaboración propia si se tienen conocimientos suficientes.

El juego creado en este proyecto no pretende ser más que un simple ejemplo de algo creado con **Flashpunk** pero si tenemos la intención de crear un juego totalmente funcional y con objetivo de que lo juegue el público en general, también podemos hacerlo con **Flashpunk**.

En este caso el juego sería factible ya que económicamente es viable debido a que disponemos de recursos gratuitos; técnicamente también es viable puesto que la tecnología necesaria para su desarrollo es muy asequible, de hecho, bastaría con un sencillo ordenador sin demasiada potencia y con una conexión a internet sencilla (todo esto se detalla en el apartado de análisis económico); comercialmente tiene salida ya que en internet hay muchas páginas que ofrecen juegos del mismo estilo que los que podamos hacer con **Flashpunk** y hay un público que los juega. Nuestro juego podría estar en una web como ésas o incluso si creciera lo suficiente podría convertirse en un juego comercial y distribuirse por otros medios.

2 - Más sobre Flashpunk: ¿Quién y cómo lo ha hecho?

Quién

El padre de todo el código de **Flashpunk** es **Chevy Ray Johnston**, un joven de unos 22 años que vive en Columbia Británica, Canadá. **Chevy** afirma no haber recibido formación sobre la creación de videojuegos ni sobre programación. Se declara autodidacta y lleva varios años desarrollando videojuegos y/o colaborando en su creación ya sean para PC, iPhone o Nintendo DS. Actualmente se gana la vida realizando trabajos de programación en **Flash**.

4. Chevy Ray Johnston (fuente: Flickr)

A **Chevy** le gustan las matemáticas, la física, la astronomía y la literatura. También le gusta practicar deporte, ir en bicicleta, nadar, acampar, etc. Y sobre todo el sombrero, le encanta su sombrero.

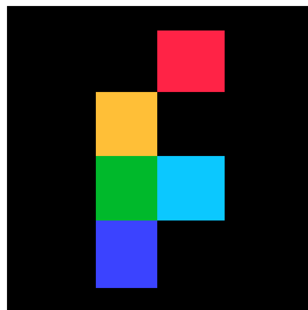
Incluso dice que le puso de nombre **Flashpunk** a su proyecto porque él es en cierta manera un poco *punk*, no le gusta seguir las reglas y hacer las cosas a su manera; también buscaba un nombre simpático, nada pretencioso y que estuviera a la moda.

Algunas webs con información sobre **Chevy** son:

- **Su página web:** http://www.chevyray.com/
- **Su página de Facebook:** http://www.facebook.com/chevyrayjohnston

Cómo

El proyecto **Flashpunk** no comenzó con la idea de ser un entorno de desarrollo como lo acabó siendo, sino que **Chevy** empezó programando una serie de códigos en **AS3** a modo de test para comprobar con qué opciones y cómo de rápido podía **Flash** tratar con mapas de bits. Fue entonces cuando descubrió **Flixel**, que es un entorno de desarrollo para videojuegos en **Flash** existente desde antes que **Flashpunk**. Pero **Flixel** no le gustó, no se sentía cómodo trabajando con ello, él tiene su manera particular de programar, de hacer las cosas; por todo ello empezó un duro y aburrido trabajo de búsqueda de información sobre **Flash** por todo internet y comenzó a programar su código, el que sería el núcleo de la librería **Flashpunk**.



5. Logo de Flixel (fuente: Flash Game Dojo)

La diferencia con **Flixel** es que éste incorpora una gran cantidad de código útil en sus clases, por ejemplo para detectar colisiones, animaciones, físicas, etc. **Flashpunk** es más básico, no incorpora tanto código mejorando el rendimiento y cumple mejor la función de ser una base a partir de la cual cada programador podrá ir añadiendo sólo lo que él necesite en cada juego. Esto puede que haga más difícil la tarea que en **Flixel**, pero se tiene más control sobre lo que se está programando y ayuda a cumplir con la filosofía de **Flashpunk**: rápido, fluido y ligero.

3 - Puesta a punto del entorno

Para empezar a desarrollar videojuegos con **Flashpunk** hay que seguir los pasos siguientes:

1. Instalar **FlashDevelop**
2. Descargar **Flex**
3. Descargar el *debug player* de **Flash**
4. Configurar **FlashDevelop**
5. Instalar **Flashpunk**

1. Para instalar **FlashDevelop** en el sistema solamente hay que ir a la web oficial del programa (http://www.flashdevelop.org) y descargar la última versión y ejecutar el instalador.
   Aunque el instalador dé la opción de instalar y configurar **Flex**, procederemos al siguiente paso ya que esta opción puede no funcionar correctamente.

2. Una vez instalado **FlashDevelop**, para que éste sea capaz de construir archivos **SWF**, necesita el código que contienen los archivos de **Flex**.
   Flex es un entorno de desarrollo que permite la creación de aplicaciones web tanto para ordenadores de escritorio como para dispositivos móviles. Para descargarlo hay que ir a esta página de **Adobe**:

   http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4

   Y descargar la última versión de la columna *Adobe Flex SDK*.

Flex 4 Downloads

Look here for a description regarding the types of SDKs available.

Look here for an explanation of the different build types found on this page.

Look here for an explanation of how to use a stable or nightly build in Flex Builder.

Latest Milestone Release Builds

Milestone	Build	Build Date	Adobe Flex SDK	Open Source Flex SDK	Adobe Add-ons
Flex 4.1 Update	4.1.0.16076	Thu June 30, 2010	Download (ZIP, 169MB)	Download (ZIP, 65MB)	Download (ZIP, 76MB)
Flex 4 Release	4.0.0.14159	Sun Mar 21 2010	Download (ZIP, 138MB)	Download (ZIP, 98MB)	Download (ZIP, 40MB)

6. Descargas de Adobe Flex (fuente: página de descarga de Adobe Flex)

Cuando se complete la descarga, es necesario descomprimir todos los archivos de **Flex** a un directorio el cual luego le indicaremos a **FlashDevelop** en su configuración.

3. El *debug player* de **Flash** es una versión especial para desarrolladores que tiene algunas funcionalidades extra, útiles para usar con **FlashDevelop**. Para descargarlo hay que ir a la página del **Flash Player** de **Adobe**:

   http://www.adobe.com/support/flashplayer/downloads.html

   Y en la sección:

   Adobe Flash Player 10.3 — Debugger (aka debug players or content debuggers) and Standalone (aka Projectors) Players for Flex and Flash developers

   Se encuentra el archivo necesario, hay que descargar el *Projector Content Debugger*.

Home / Support / Flash Player /

Adobe Flash Player Support Center

Downloads

Developers can download updated Flash Players for use with Flash from this page.

Get the latest version
Download the most recent version of Adobe Flash Player.

Updates by version:
Flash Player 10.3
Local Content Updater
Uninstallers

Get older versions
Download older versions of Adobe Flash Player.

Your rights to use any Flash player, projector, standalone player, plug-in, runtime or ActiveX control provided to you below, shall be solely as set forth in the following link, http://www.adobe.com/go/flashplayer_usage. Unless and except as provided therein, you shall have no rights to use or distribute such software.

ADOBE FLASH PLAYER 10.3 — DEBUGGER (AKA DEBUG PLAYERS OR CONTENT DEBUGGERS) AND STANDALONE (AKA PROJECTORS) PLAYERS FOR FLEX AND FLASH DEVELOPERS

08/09/2011 - Updated debugger (aka debug players or content debuggers) and standalone (aka projector) versions of Flash Player 10.3 are available for Flash Builder, Flash Catalyst, and Flash Professional users. These players contain fixes for critical vulnerabilities identified in Security Bulletin APSB11-21. All users are encouraged to update to the new players. These new players are version 10.3.183.5.

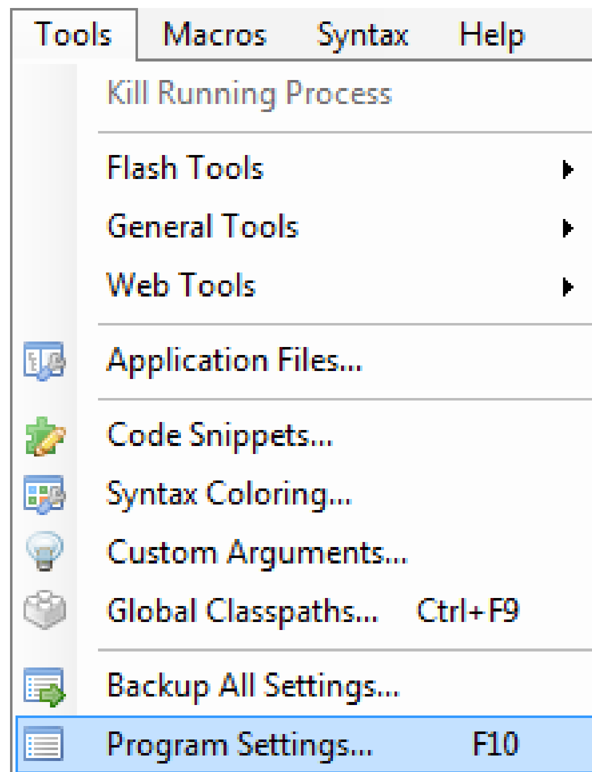
Windows

- Download the Windows Flash Player 10.3 ActiveX control content debugger (for IE) (EXE, 3.22MB)
- Download the Windows Flash Player 10.3 Plugin content debugger (for Netscape-compatible browsers) (EXE, 3.19MB)
- Download the Windows Flash Player 10.3 Projector content debugger (EXE, 6.50MB)
- Download the Windows Flash Player 10.3 Projector (EXE, 5.51MB)

7. Descargas de Adobe Flash Player (fuente: página de descarga de Adobe Flash Player)

Una vez descargado hay que poner el archivo en el directorio donde están los archivos de Flex descomprimidos. Es decir que si **Flex** fue descomprimido en *C:\Flex*, éste archivo se hallará en *C:\Flex\flashplayer_10_sa_debug.exe*.

4. Para configurar correctamente **FlashDevelop** sólo hay que indicarle en qué directorios se encuentran los archivos de **Flex** y el *debug player* de **Flash**. Para hacer esto hay que ir al menú y hacer clic en *Tools->Program settings*.

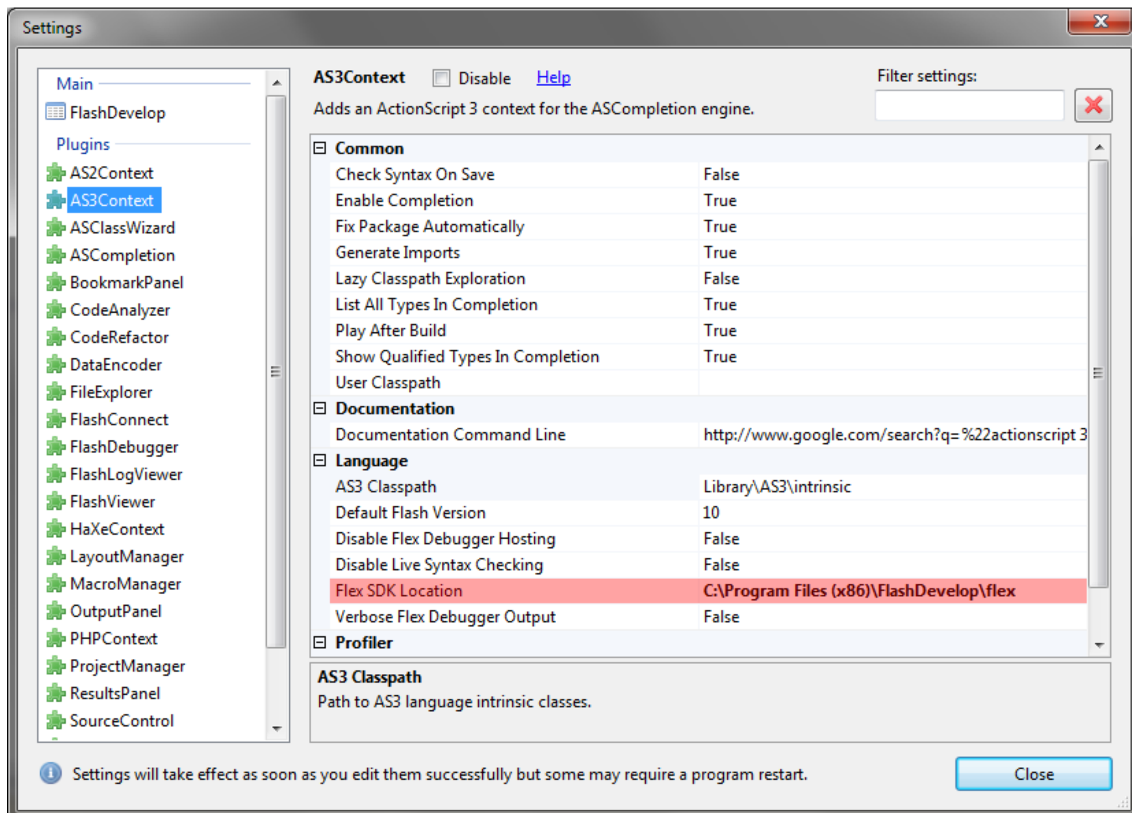


8. Opción del menú de FlashDevelop para configurarlo (fuente: propia)

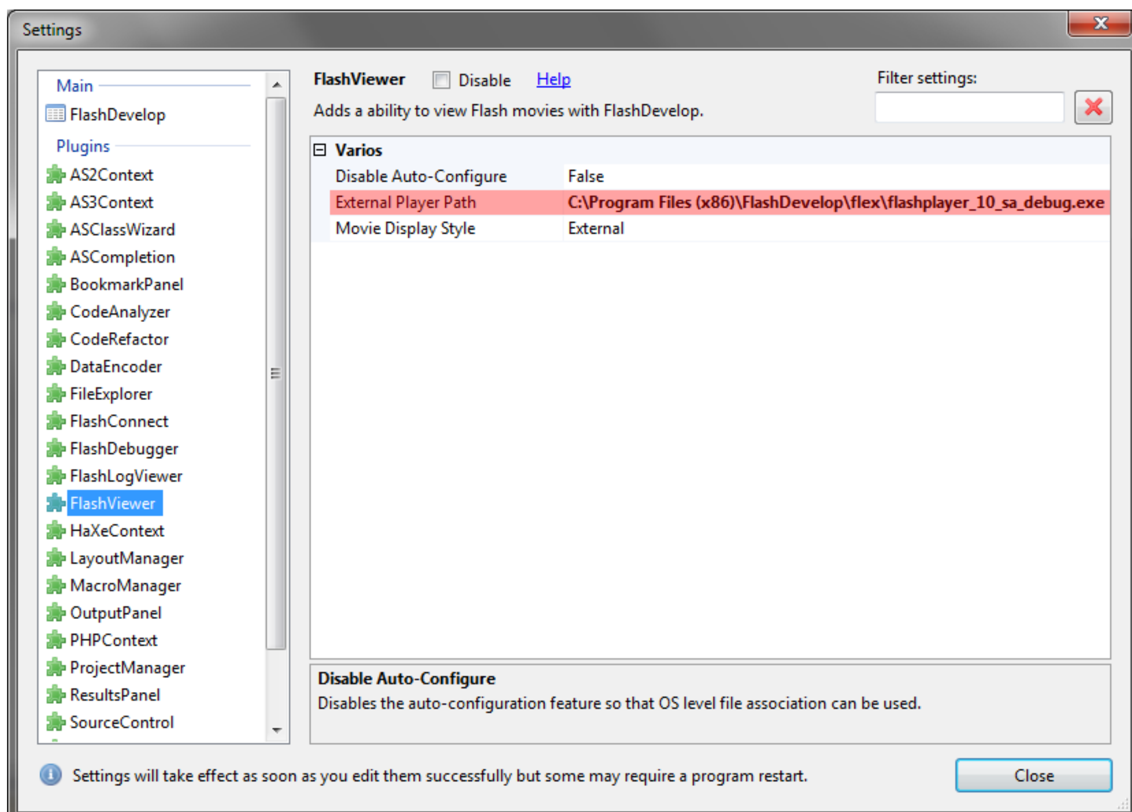
Una vez en la pantalla de configuración, se debe seleccionar la opción *AS3 Context* del menú de la izquierda para así poder indicar en el campo *Flex SDK Location* la ruta donde Flex fue descomprimido antes.

Acto seguido hay que hacer *click* en la opción *FlashViewer* del menú de la izquierda y después modificar el campo *External Player Path* indicando la ruta donde el *debug player* de **Flash** fue descargado antes.

Las siguientes imágenes ilustran estos pasos:



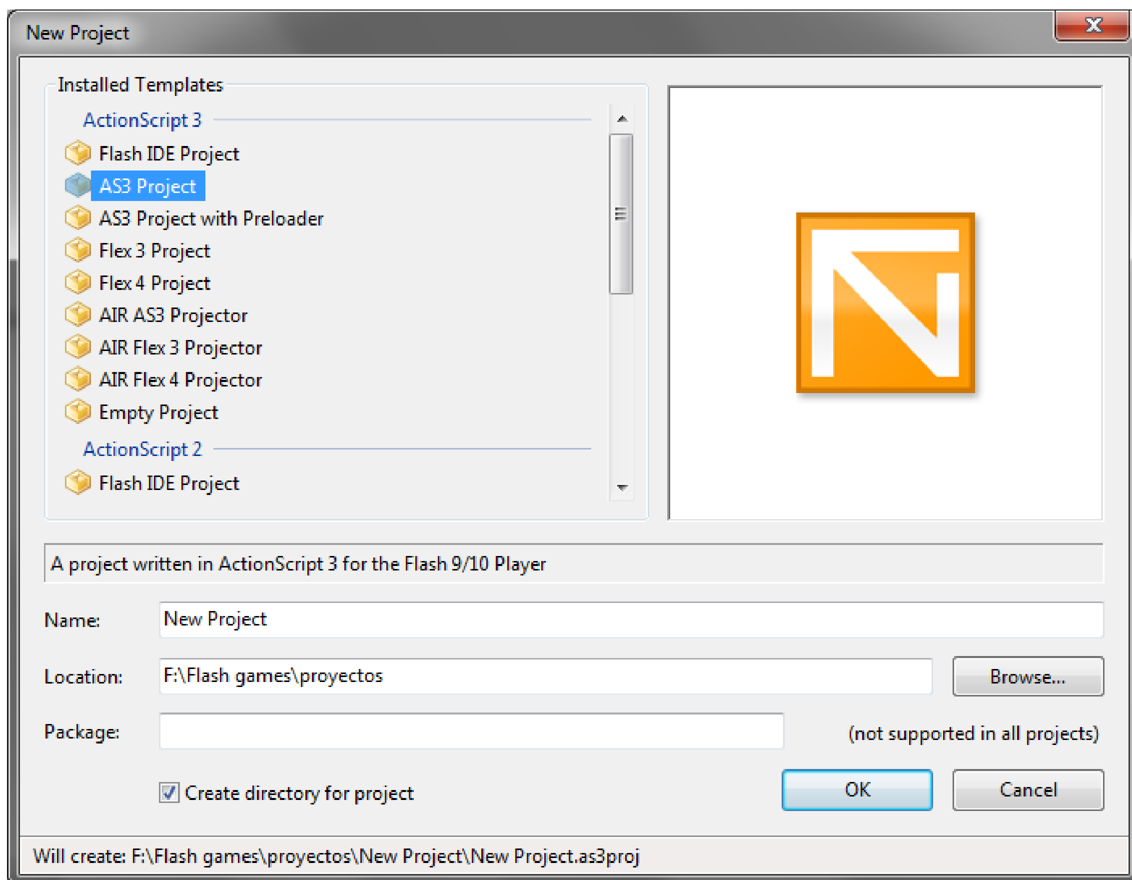
9. Pantalla de configuración de FlashDevelop (fuente: propia)



10. Pantalla de configuración de FlashDevelop (fuente: propia)

5. La instalación de **Flashpunk** es realmente sencilla. **Flashpunk** no es un programa o un software parecido, sino una colección de archivos de **ActionScript 3** que contienen el código necesario para comenzar a trabajar en algún juego **Flash**. Para utilizarlo sólo hay que descargarlo (desde la web *http://flashpunk.net*) e importarlo en el proyecto que se desee realizar.

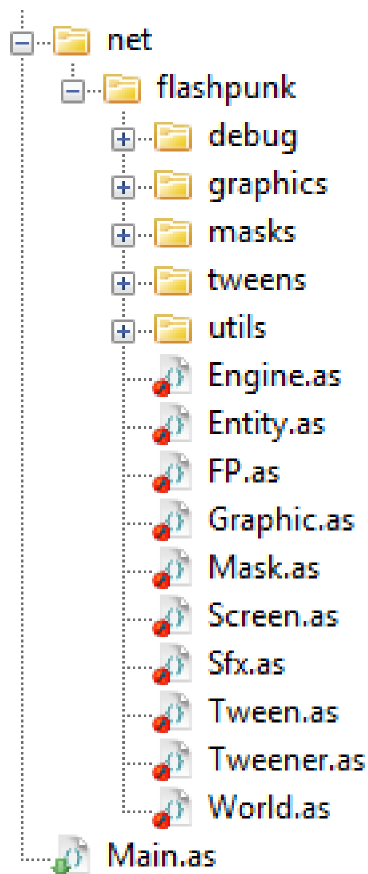
   Para importar la librería **Flashpunk** en un proyecto y poder utilizarla primero hay que comenzar un nuevo proyecto en **FlashDevelop**. Para ello basta con hacer *click* en *Project->New Project* y aparecerá una ventana como esta:



11. Pantalla de nuevo proyecto de FlashDevelop (fuente: propia)

   En esta ventana hay que seleccionar AS3 Project y poner el nombre que se quiera al proyecto, así como indicar la ruta donde éste se guardará si la que se indica por defecto no nos gusta.

   Ahora sólo queda descomprimir el *zip* que contiene la librería **Flashpunk** anteriormente descargado de tal forma que el directorio *net* de su interior (tal directorio contiene todo lo que es **Flashpunk**) se aloje en el mismo directorio que el archivo *Main.as* de nuestro proyecto de **FlashDevelop**. Quedando, por ejemplo, así:



12. Ejemplo de directorio de un proyecto en FlashDevelop (fuente: propia)

Y para importarlo usaremos los comandos *import*, y así poder empezar a utilizar sus funciones y clases. Un ejemplo de importación de **Flashpunk**:

```
1  package
2  {
3      import net.flashpunk.Engine;
4      import net.flashpunk.FP;
5      import flash.events.Event;
6  
7      public class Main extends Engine
8      {
```

13. Ejemplo comandos import en el archivo Main.as (fuente: propia)

4 - Análisis: paquetes y clases

Para tener una completa visión de todos los paquetes y clases con sus propiedades y métodos lo mejor sin duda es acudir a la *web* de **Flashpunk**. En la sección *Learn* encontraremos tutoriales, enlaces a herramientas útiles como **FlashDevelop** y la documentación, que se puede consultar *online* y también puede descargarse para consultarla desde el disco duro sin necesidad de conexión a internet.

Cabe destacar otra sección de la *web* cuya existencia puede llegar a ser de gran utilidad, el foro. En él se pueden encontrar tutoriales, ejemplos y soluciones a problemas que pueden surgir utilizando **Flashpunk** en el desarrollo de juegos.

net.flashpunk

En el paquete principal **net.flashpunk** encontramos clases imprescindibles para que funcionen los juegos que se quieran crear como la clase **Engine**. Esta clase es el tipo del cual será la clase principal de los juegos. Se compilará siempre, es necesaria. En el constructor de esta clase se deben especificar las propiedades del juego como el ancho, alto y los *frames* por segundo a los que correrá el clip de película de **Flash** que es el juego y que se está creando aquí.

Otra clase imprescindible es la clase **World** que como su nombre indica es lo que en la práctica llamamos mundo. Un mundo es un contenedor del juego que contiene todos los elementos activos del mismo. Es útil para llevar una correcta organización del juego. Así pues se podrá tener un mundo llamado “menu” para el menú del juego, otro llamado “Nivel1” para el primer nivel, etc. En esta clase se pueden controlar elementos importantes para un mundo como pueden ser la cámara, la posición del cursor del ratón, cuántos elementos hay y las capas que ocupan (cual se verá por delante de otro).

Siguiendo con las clases que todo juego necesita está la clase **Entity**. Los elementos que añadamos a un mundo serán del tipo **Entity** o derivarán de éste. Esta clase es realmente útil ya que tiene propiedades que nos permiten controlar su posición, el gráfico o imagen que tiene asociado, la capa a la que pertenece (control de profundidad) y el tipo de elemento que es entre otras.

El tipo se define en la propiedad “type” mediante una cadena de caracteres. Esto sirve para controlar las colisiones que tienen los elementos con otros elementos. Por ejemplo, si en un juego tenemos dos elementos derivados de la clase **Entity** y uno es el jugador y el otro un enemigo, lo que se podría hacer es establecer en el jugador “type

= jugador" y en el enemigo "type = enemigo"; en el código del jugador se podrá verificar si existe colisión con alguna **Entity** de tipo "enemigo" y actuar en consecuencia.

Pero la clase **Entity** aún tiene mucha utilidad. Contiene métodos que permiten añadirle gráficos o imágenes, controlar colisiones contra otras **Entities**, contra un punto, contra un rectángulo, calcular la distancia desde un punto o desde otra **Entity**, etc.

Otra clase necesaria es la clase **FP**. Esta clase es utilizada para acceder a propiedades y funciones globales del juego como por ejemplo el *framerate*, la consola (para *debug*), el tiempo, la pantalla, el mundo activo en el juego, generar números aleatorios, etc.

Hasta aquí las clases imprescindibles para que un juego funcione. Siguiendo en el mismo paquete **net.flashpunk** aún quedan otras clases por resumir cuya utilidad nos ayudará en muchas ocasiones a obtener en un juego el resultado esperado.

Una clase que seguro utilizaremos, aunque no sea directamente, es la clase **Graphic**. Esta clase es la clase base para los gráficos que puede mostrar **Entity** y que se encarga de mostrar en pantalla una imagen en concreto en una posición determinada.

Un caso similar al anterior lo encontramos con la clase **Mask**. Es la clase base para las máscaras que podamos declarar en **Entity** y controla las colisiones contra otras **Masks**.

La siguiente clase a comentar es la clase **Screen**. Con esta clase podemos modificar la pantalla y acceder a propiedades relacionadas con ella: su tamaño, el punto de origen para las transformaciones (por ejemplo rotaciones), la posición del puntero del ratón en la pantalla, color, ángulo, etc.

También encontramos la clase **Sfx** que permite reproducir sonidos. Podremos controlar la reproducción de los sonidos, su volumen, que se reproduzcan en bucle o sólo una vez, etc.

Para acabar con el paquete **net.flashpunk**, tenemos las clases **Tween** y **Tweener**. La clase **Tween** es la clase base para los objetos de tipo *Tween* (ya se comentarán más adelante) que básicamente lo que hacen es interpolar valores de variable o movimientos. Así pues con esta clase se pueden controlar esos *tweens* ejecutándolos, parándolos, consultando cuánto se han completado, su duración, etc.

La clase **Tweener** no es más que un contenedor de **Tweens**. En ella podemos añadirlos o quitarlos.

net.flashpunk.debug

El siguiente paquete es el **net.flashpunk.debug** que únicamente contiene la clase **Console** que es la consola del *debugger*. Deberemos importar esta clase si queremos tener la consola en nuestro juego y comprobar exactamente su funcionamiento. Podremos activar la consola mediante la clase **FP** del paquete principal **net.flashpunk**.

net.flashpunk.graphics

Otro paquete de indudable utilidad es el **net.flashpunk.graphics** que nos permite crear y controlar elementos que serán vistos en pantalla.

En este paquete encontramos clases como la clase **Image** que permite mostrar en pantalla de forma sencilla una imagen simple, sin animación pero a la cual se le pueden aplicar transformaciones. Se le puede rotar, cambiar de tamaño, aplicar transparencia, cambiar de color, etc.

Si la imagen que queremos mostrar es una imagen animada, podemos hacerlo con la clase **Spritemap**. Esta clase contiene muchas de las propiedades de la clase **Image** pero se añaden unas cuantas para controlar la animación.

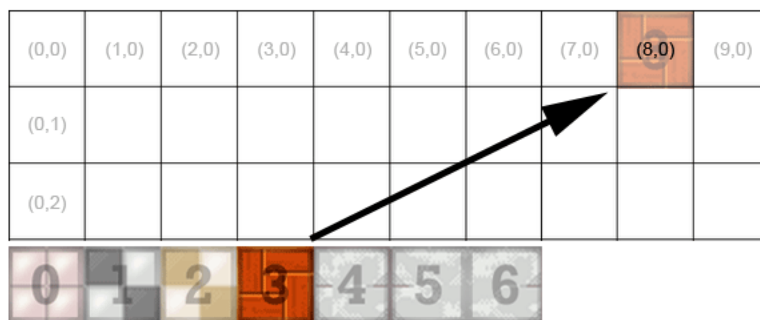
Para que funcione esta clase, hay que pasarle al constructor de la misma una imagen fuente. Esta imagen ha de consistir en un mosaico de todos los *frames* de nuestras animaciones. Luego indicamos el alto y ancho de los *frames* y la clase **Spritemap** se encargará de dividir la imagen en imágenes del alto y ancho indicado a las cuales se les asignará un índice que empieza en 0.

14. Ejemplo del índice de frames en un Spritemap (fuente: página web de Flashpunk)

También podemos asignar animaciones. Para esto sólo tendremos que indicar un nombre con el cual identificar la animación, los *frames* que componen la animación (por ejemplo: [0, 1, 2, 3]), el *framerate* (a cuantos *frames* por segundo queremos que se ejecute la animación) y si ha de reproducirse en bucle o por el contrario solamente una vez.

Teniendo claro el concepto de qué son **Image** y **Spritemap**, podemos comprender ahora la existencia de dos clases como **TiledImage** y **TiledSpritemap**. Estas clases nos permiten crear una imagen del tamaño que le indiquemos compuesta de repeticiones de la imagen fuente que también indicaremos en el constructor de las mismas. En el caso de **TiledSpritemap**, la imagen que se repetirá y formará el mosaico resultante será una imagen animada de la cual podremos controlar la animación (como en **Spritemap**).

Una clase relacionada con estos últimos conceptos es la clase **Tilemap**. Esta clase permite construir una imagen formada a partir de otras imágenes más pequeñas. Por ejemplo, si queremos construir un muro de ladrillos no será necesario dibujar el muro entero en un editor de imágenes e importarlo como una imagen; sino que con dibujar un ladrillo ya es suficiente porque podemos repetirlo tantas veces como sea necesario y formar la imagen final que es el muro.



15. Ejemplo de Tilemap (fuente: página web de Producerism)

En el constructor de la misma le indicamos qué imagen usaremos. Esta imagen ha de contener todos los *tiles* o azulejos que se usarán para formar el mosaico resultante. También debemos indicar el tamaño de la imagen mosaico y el tamaño de los *tiles*. Con todo esto ya podemos indicarle en que posiciones queremos un *tile* u otro y formar la imagen final.

Luego tenemos la clase **Stamp** que permite simplemente mostrar una imagen, sin transformaciones ni animaciones. Lo único que se debe hacer es indicarle la imagen fuente y la posición donde queremos que se vea.

La clase **Text** dibuja en pantalla el texto que queramos. Es posible importar fuentes a nuestro código y utilizarlas. A parte de la fuente también es posible controlar aspectos como la transparencia, el tamaño, el color, etc.

La clase **Backdrop** permite utilizar una imagen de fondo que puede repetirse tanto horizontal como verticalmente. Esto es útil a la hora de crear fondos con efecto de

parallax scrolling[7]. Podemos controlar su repetición y su factor de desplazamiento entre otras cosas.

La clase **Emitter** permite emitir y mostrar múltiples partículas de las cuales podremos controlar su movimiento y apariencia. Las partículas pueden ser una imagen determinada, solamente hay que indicarle la imagen fuente y el tamaño de la partícula deseado. Esta clase puede ser útil para crear explosiones y efectos parecidos.

Relacionadas con la clase **Emitter** tenemos las clases **Particle** y **ParticleType**. No nos han de preocupar ya que no tenemos por qué usarlas. Las utiliza la clase **Emitter** desde la cual tenemos control absoluto sobre las partículas (utilizando indirectamente estas clases). Así pues, **Emitter** utiliza **Particle** y **ParticleType** para realizar el seguimiento y definir el tipo de las partículas.

Luego tenemos la clase **Graphiclist** que consiste en una lista de gráficos. Es un elemento gráfico que puede contener múltiples gráficos de varios tipos. Útil si queremos que un elemento se componga de varios gráficos. Tiene métodos que permiten añadir y quitar gráficos de la lista y así su uso es realmente simple.

Para finalizar con este paquete tenemos las clases **PreRotation** y **Canvas**. Con **PreRotation** podemos generar imágenes rotadas a partir de una imagen fuente y así después, cuando queramos rotar la imagen original, poder acceder a esas imágenes ya rotadas en lugar de tener que procesar la rotación cada vez.

La clase **Canvas** la podemos entender como un lienzo sobre el cual podemos pintar. Tiene métodos que permiten copiar píxeles desde una imagen fuente, copiar gráficos, dibujar rectángulos, pintar píxeles y moverlos, etc.

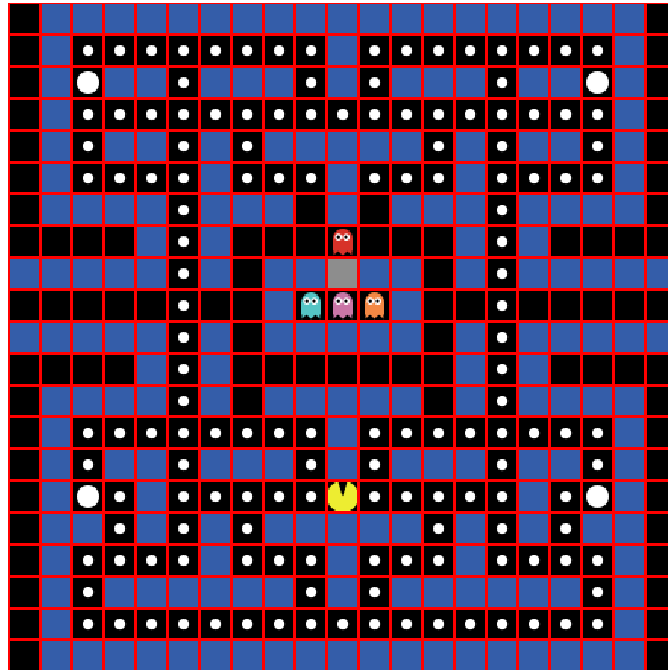
net.flashpunk.masks

El paquete **net.flashpunk.masks** permite crear máscaras para los elementos de nuestro juego. Las máscaras son los elementos que no vemos en tiempo de ejecución del juego pero que existen y son lo que nos permiten detectar colisiones con otros elementos del juego.

La clase más sencilla y más usada de este paquete es la clase **Hitbox**. No usaremos esta clase directamente porque todos los elementos que deriven de la clase **Entity** lo incorporan. Solamente hay que usar el método **setHitbox** para darle un ancho y un alto.

[7] Parallax scrolling es una técnica especial de scroll (movimiento) en la que se mueven más de un plano en lugar de sólo uno para crear efecto de profundidad con gráficos 2D.

Luego tenemos la clase **Grid** con la que podemos construir una máscara como si fuera una rejilla, o dicho de otra manera, como en un tablero dividido en casillas cuadradas en el que podemos poner máscaras (de forma cuadrada ocupando la casilla entera) en las posiciones que queramos.



16. Ejemplo de Grid (fuente: foro de MSDN)

En la imagen de ejemplo podemos ver que para que nuestro personaje no pueda circular por zonas por las que no debe se puede hacer mediante una *grid* o rejilla. En este caso asignaríamos máscara a las celdas azules de la rejilla y el personaje sólo podría moverse por el resto.

Para crear máscaras más complejas disponemos de la clase **Pixelmask**. Esta clase permite crear una máscara con una forma determinada por un gráfico que nosotros indiquemos. De esta forma, la máscara no será cuadrada y podremos detectar colisiones de forma más precisa.

Por último tenemos la clase **Masklist** que no es más que una lista de máscaras. Como lista, le podremos añadir y quitar máscaras, acceder a ellas indicando un índice y asignarlas a un elemento.

net.flashpunk.tweens.misc

En el paquete **net.flashpunk.tweens.misc** encontramos clases que nos permiten crear interpolaciones de valores para nuestras variables. Por ejemplo, si queremos mover un objeto en pantalla desde una posición hasta otra, podemos modificar manualmente su posición cambiando los valores de sus coordenadas en cada *frame* y dejar de modificarlos cuando el objeto alcance el destino, pero también podemos crear una interpolación. Cuando creamos una interpolación, basta con indicar el valor inicial, el valor final y la duración y todos los valores intermedios se generan automáticamente.

17. Ejemplo de interpolación de movimiento (fuente: propia)

Todos los paquetes que permiten crear interpolaciones son útiles para obtener un resultado más fluido y más "perfecto". Podemos mover un objeto manualmente y en teoría es lo mismo, pero si lo movemos mediante interpolación en la práctica comprobaremos como el movimiento resulta visualmente más suave y atractivo.

Las clases que pertenecen a este paquete son **Alarm**, **AngleTween**, **ColorTween**, **MultiVarTween**, **NumTween** y **VarTween**.

Alarm es un poco distinta ya que no calcula los valores intermedios de un valor a otro. Simplemente le indicamos una duración al término de la cual se ejecutará lo que queramos, por lo tanto no es más que una alarma.

Con **AngleTween**, **ColorTween**, **NumTween** y **VarTween** podemos interpolar valores ya sean ángulos, colores o valores numéricos en general.

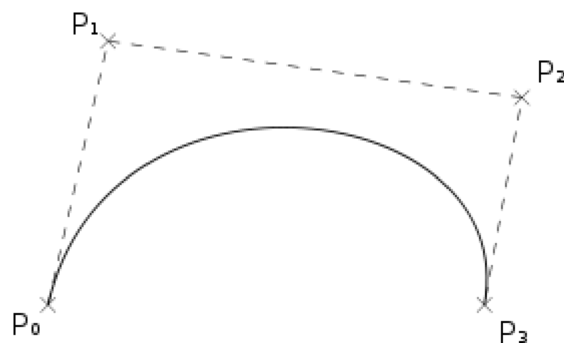
Con **MultiVarTween** también podemos interpolar valores, pero más de uno al mismo tiempo.

net.flashpunk.tweens.motion

El paquete **net.flashpunk.tweens.motion** está destinado a crear interpolación de movimiento. Las clases que pertenecen a este paquete no requieren de una explicación extra.

Así pues, encontramos la clase **CircularMotion** que permite crear un movimiento circular. Deberemos indicarle el centro, el radio de giro, el ángulo inicial, el sentido de giro y la duración.

También tenemos la clase **CubicMotion** que permite crear un movimiento a lo largo de una curva cúbica.

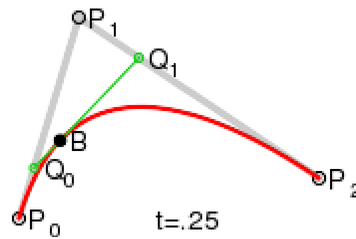


18. Curva cúbica de Bézier (fuente: Wikipedia)

La clase **LinearMotion** sirve para crear un movimiento en línea recta, de un punto a otro. Junto a ésta encontramos la clase **LinearPath** que determina movimientos lineales a lo largo de una serie de puntos.

La clase **Motion** es la clase base para las otras clases que crean interpolación y no nos preocuparemos por ella ya que no tenemos por qué usarla directamente.

Para finalizar este paquete tenemos las clases **QuadMotion** y **QuadPath**. La primera sirve para crear interpolación de movimiento a lo largo de una curva cuadrática y la segunda determina movimientos a lo largo de una serie de puntos utilizando curvas cuadráticas.



19. Construcción de un curva cuadrática de Bézier (fuente: Wikipedia)

Para más información sobre curvas cúbicas y cuadráticas es aconsejable acudir a un sitio web como **Wikipedia**[8].

net.flashpunk.tweens.sound

El siguiente paquete a tratar es **net.flashpunk.tweens.sound** que nos permite interpolar el volumen de nuestro juego de un valor hasta otro. En este paquete encontramos dos clases: **Fader** y **SfxFader**.

La clase **Fader** permite interpolar el volumen general del juego desde el valor actual hasta el valor que indiquemos con la duración que queramos.

La clase **SfxFader** permite interpolar el volumen de un efecto de sonido en concreto. Funciona de igual manera que **Fader** pero en el constructor tenemos que indicarle el efecto de sonido al que afectará. Dicho efecto de sonido pertenecerá a la clase **Sfx** tal y como hemos visto al repasar el paquete principal **net.flashpunk**. La clase **SfxFader** también permite cruzar efectos de sonido, bajando el volumen de uno y subiendo el del otro al mismo tiempo.

net.flashpunk.utils

El último paquete es **net.flashpunk.utils** que como su nombre indica incluye utilidades para nuestro juego, algunas de ellas imprescindibles si queremos que lo que estamos creando pueda llamarse juego.

Primero encontramos la clase **Data** que es usada para guardar y cargar información de las *cookies* guardadas. Tiene métodos para cargar y guardar información, para cargar y guardar booleanos, enteros y cadenas de caracteres.

Luego tenemos la clase **Draw** que nos permite acceder a funciones de dibujo. Dichas funciones no están pensadas para ser componentes gráficos de los elementos que

[8] URL de la Wikipedia: http://es.wikipedia.org

deriven de la clase **Entity** sino para realizar funciones de testeo y *debug*. Así pues, en esta clase encontramos métodos para dibujar círculos, curvas, rectángulos, líneas, rectas, etc.

También vemos la clase **Ease** que contiene funciones útiles que pueden ser utilizadas por las clases que crean interpolaciones. En esas interpolaciones establecemos por ejemplo un movimiento desde una coordenada hasta otra, pero además podemos indicar que se haga de una forma u otra, con las funciones de **Ease**. Por ejemplo podemos utilizar la función **bounceOut** para que el objeto que movamos haga unos pequeños rebotes al llegar a su destino. Podemos escoger entre varios tipos de movimientos como exponencial, cuadrático, etc.

Existen algunas páginas web que ayudan a comprender los diferentes tipos de movimientos de la clase Ease, una muy buena es:

http://www.openprocessing.org/visuals/?visualID=17112

Donde podemos seleccionar qué movimientos ver y verlos todos a la vez.

Para finalizar tenemos las clases **Input** y **Key**. La clase **Input** nos sirve para detectar pulsaciones de teclado y ratón. La clase **Key** contiene las constantes que identifican la tecla pulsada que serán usadas por la clase **Input**. Así pues utilizaremos las dos clases para saber qué tecla ha sido pulsada en un momento determinado.

Con esto hemos repasado los paquetes y las clases que contiene **Flashpunk**. En el siguiente apartado continuaremos el análisis con un ejemplo práctico.

5 - Análisis: pequeño ejemplo práctico

En este apartado se analizará **Flashpunk** desde el punto de vista práctico. Se comprobará lo práctico que es, qué límites tiene, qué problemas podemos encontrarnos, etc.

En primer lugar hay que decir que **Flashpunk** no pone los límites sino que podemos programar hasta donde **ActionScript** y **Flash** nos permitan. **Flashpunk** pone a nuestra disposición una serie de funciones que nos permiten hacer ciertas tareas sin tener que programarlas nosotros mismos. De hecho, si tuviéramos en manos un proyecto lo suficientemente grande (y los conocimientos necesarios) acabaríamos haciendo nosotros nuestra librería a medida para nuestras necesidades, otro **Flashpunk** por así decirlo pero adaptado.

Flashpunk, tal y como está, es un recurso muy bueno para crear juegos de estilo retro, usando *sprites*. Sobre todo para aquellos que no tengan un gran conocimiento sobre el entorno **Flash** y **ActionScript**.

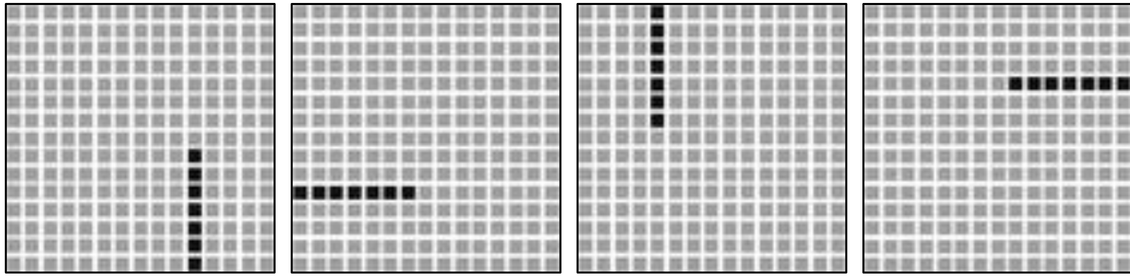
La idea

En este ejemplo vamos a tratar de hacer algo sólo con lo que **Flashpunk** nos ofrece, a ver hasta dónde podemos llegar con la idea que tenemos en mente.

Lo primero es tener una idea, saber qué hacer. Vamos a crear un juego de tipo puzle en el que lo más importante sea averiguar cómo superar los niveles. Cada nivel ha de ser más complicado que el anterior y el número de niveles ha de ser suficiente y variado, o al menos ampliable, para que la experiencia no sea monótona ni aburrida.

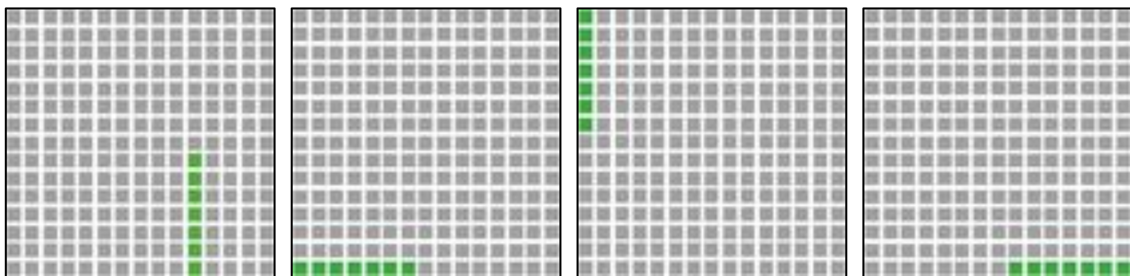
Pensaremos en unos niveles de forma cuadrada los cuales podamos girar en los dos sentidos para que nuestro personaje, ayudado de la gravedad, pueda alcanzar la salida hacia el siguiente nivel.

Los niveles estarán pensados como tableros de 15 por 15 casillas de área utilizable por el personaje. En cada posición del tablero podremos colocar una casilla del tipo que queramos. En este caso contemplaremos tres tipos de casillas: las normales que no se mueven, las verdes que se mueven en una dirección en dos sentidos y las azules que se mueven en los cuatro sentidos.



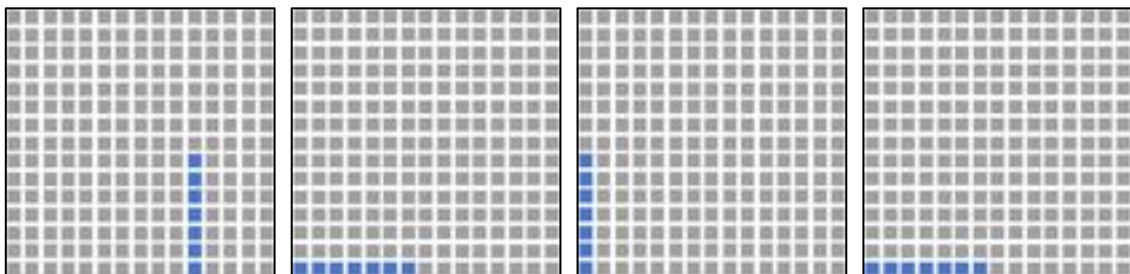
20. Ejemplo de nivel con casillas normales (fuente: propia)

En la imagen número 20 tenemos un ejemplo de nivel en el que hemos colocado una columna de casillas normales. Al girar este nivel 90º hacia la derecha observamos como las casillas también giran conservando su posición relativa en el tablero.



21. Ejemplo de nivel con casillas verdes (fuente: propia)

En la imagen número 21 tenemos un ejemplo del mismo nivel anterior pero con casillas verdes en lugar de normales. Al girar el nivel 90º hacia la derecha podemos comprobar como las casillas verdes giran al igual que las normales, pero esta vez las casillas caen hacia abajo en una sola dirección.



22. Ejemplo de nivel con casillas azules (fuente: propia)

En la imagen número 22 tenemos el mismo nivel pero con casillas azules colocadas. Estas casillas caerán siempre hacia abajo en cada rotación del nivel.

Una vez que tenemos una idea clara de lo que vamos a hacer podemos empezar a crear nuestro juego.

El cubo

Empezaremos pensando cómo se van a representar los niveles en pantalla. La idea es tener una clase encargada de colocar las casillas en sus respectivas posiciones al inicio y actualizarlas en cada rotación. Esta clase se llamará “Cubo”.

La clase “Cubo” lo primero que hará es poner el borde del cubo. Necesitamos un borde hecho de casillas normales de modo que el personaje quede encerrado dentro y no caiga nada más empezar. Este borde medirá 17 casillas de ancho por 17 de alto, quedando así un tablero de 15 por 15 en su interior, lo que será el espacio jugable.

Lo segundo que se hará es colocar un fondo para el cubo. Este fondo ha de estar centrado respecto al cuadrado de casillas que forman el borde y se ha de controlar su ángulo para que gire al mismo tiempo que el cubo.

La clase “Cubo” ha de tener una función encargada de cargar los niveles. A esta función le pasaremos como parámetro un archivo en formato **XML** que contendrá la información necesaria para construir un nivel. Los motivos por los cuales se ha optado por cargar los niveles desde archivos **XML** son la modularidad y la sencillez entre otros. Si no utilizáramos este método, tendríamos que crear una clase **world** para cada nivel que quisiéramos crear; utilizando archivos **XML** solamente necesitamos una clase **world** desde la cual vamos leyendo estos archivos.

Los archivos **XML** tienen un formato de texto plano que podemos abrir y leer desde el editor de texto que queramos. Son fáciles de interpretar, tanto es así que podemos editarlos directamente.

En la siguiente imagen podemos comprobar lo sencillo que es un nivel del juego en un archivo **XML**. En el ejemplo vemos el primer nivel que solamente consta de una columna de casillas normales. Hemos organizado las casillas en etiquetas de modo que las normales, las verdes y las azules estarán en las etiquetas de los mismos nombres. Luego para cada casilla normal sólo hay que indicar la posición en el tablero mediante los campos x e y, desde la 1x1 hasta la 15x15.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<data>
<normales>
    <casilla x="11" y="10" />
    <casilla x="11" y="11" />
    <casilla x="11" y="12" />
    <casilla x="11" y="13" />
    <casilla x="11" y="14" />
    <casilla x="11" y="15" />
</normales>
<verdes>

</verdes>
<azules>

</azules>
<entradas>
    <entrada x="4" y="15" />
</entradas>
<salidas>
    <salida x="14" y="15" />
</salidas>
<energias>
    <energia e="25" />
</energias>
<objetos>

</objetos>
</data>
```

23. Archivo XML del primer nivel (fuente: propia)

El resto de etiquetas se explican por sí solas: en “entradas” pondremos la posición en la que aparecerá el personaje, en “salidas” pondremos la posición que deberemos alcanzar para pasar al siguiente nivel, en “energías” detallaremos la energía con la que empieza el personaje (ciertas acciones como rotar el cubo requieren energía) y en “objetos” pondremos elementos distintos a los anteriores si los hubiéramos creado.

Debemos detallar un poco más la manera de describir las casillas verdes en un archivo XML. Puesto que estas casillas se desplazan en una dirección, debemos indicar de cual se trata. Lo indicaremos así:

```xml
<casilla x="3" y="4" c="1" />
<casilla x="9" y="4" c="0" />
```

Así pues, mediante el campo “c”, podemos crear una casilla verde que caerá en la primera rotación del cubo o en la segunda. Cuando este campo es igual a 0, es el valor por defecto, es decir, la casilla caerá hacia abajo en la primera rotación. Cuando este campo es igual a 1, la casilla no caerá en la primera rotación puesto que su dirección estará horizontal; en la segunda rotación la veríamos caer ya que ahora su dirección estaría vertical y las casillas siempre caen hacia abajo.

Para las casillas azules no es necesario indicar nada ya que caen siempre.

La clase “cubo”, una vez leído el nivel del archivo **XML**, deberá crear las casillas necesarias y colocarlas en sus posiciones adecuadas. Las casillas han de rotar alrededor del centro del cubo y esto se hace mediante la clase **CircularMotion2** que es una versión ligeramente modificada de la clase original de **Flashpunk CircularMotion**, de hecho, la única diferencia es haber cambiado el giro que se realiza de 360º a 90º que es lo que nos interesa en nuestro juego. Esta clase, para comenzar a girar requiere que le indiquemos el ángulo inicial de nuestro objeto. Es por eso que nuestra clase “cubo” calculará para cada casilla su radio y su ángulo inicial respecto al centro.

La clase “cubo” es también la indicada de hacer rotar las casillas y el fondo del cubo.

El nivel

Nuestro juego cargará los diferentes niveles desde esta clase, la clase “Nivel”. Cada vez que el personaje alcance la salida de un nivel, se eliminará dicho nivel y se construirá el siguiente. Esta es nuestra clase de tipo **world**, y como se ha explicado antes, con el método de cargar los niveles desde archivos de texto no necesitaremos una para cada nivel.

La clase “Nivel” también es la encargada de gestionar los cinco perfiles que podemos utilizar en el juego. Cada perfil guarda el nombre del usuario y el nivel alcanzado. En cualquier momento el usuario puede reiniciar su nivel (volver al primero) y cambiar su nombre.

Para seleccionar un perfil hemos creado una pequeña clase llamada “Perfilador” que lo único que hace es mostrar en pantalla los cinco perfiles y permitirnos seleccionar uno, haciéndolo el perfil activo y cargando su correspondiente nivel.

Pero ¿dónde se guarda la información de los perfiles? No podemos guardarla en un archivo cualquiera en un directorio que nosotros queramos. Para hacer eso tendríamos que abrir un cuadro de diálogo en el que el usuario ha de seleccionar nombre y destino para el archivo. Esto es así debido a razones de seguridad, para que una aplicación en **Flash** no tenga acceso al disco duro del usuario sin que éste se entere. No obstante, hay una forma de guardar y cargar información en el sistema del usuario, una forma controlada. Se utilizan los **Shared Objects**, que son las *cookies* de **Flash**. Estos archivos se guardan en un directorio determinado y no es posible cambiarlo. Podemos guardar un archivo distinto por cada aplicación **Flash** y no se borrará al cerrar el navegador ni el reproductor de **Flash**. Podemos guardar muchos tipos de datos en estos archivos, en nuestro caso guardamos un archivo **XML** que no es más que texto plano.

La primera vez que ejecutemos el juego no habrá en el sistema ningún archivo del tipo **Shared Object**, por eso el juego al no ser capaz de leer el archivo creará uno nuevo a partir de la información que hay en un archivo **XML** perteneciente al juego. El contenido de dicho archivo es el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<data>
<perfiles>
        <perfil num="1" niv="1" nom="Jugador 1" />
        <perfil num="2" niv="1" nom="Jugador 2" />
        <perfil num="3" niv="1" nom="Jugador 3" />
        <perfil num="4" niv="1" nom="Jugador 4" />
        <perfil num="5" niv="1" nom="Jugador 5" />
</perfiles>
</data>
```

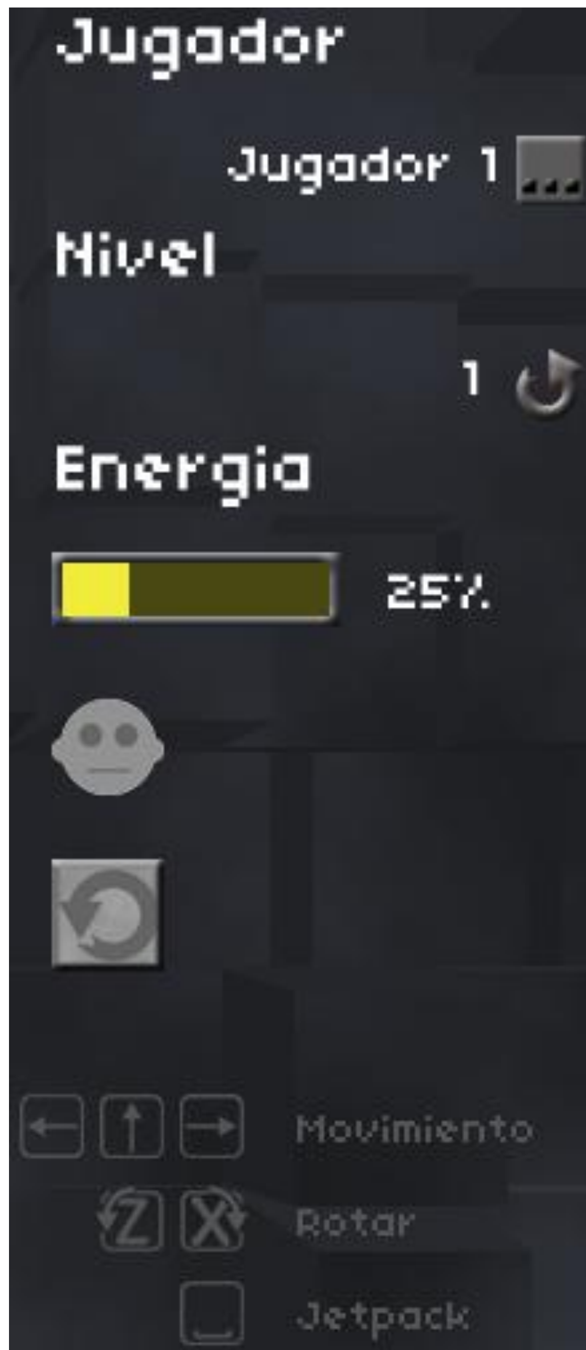
Así a cada perfil se le asigna un nombre genérico y el nivel 1.

Otro aspecto que se gestiona desde la clase “Nivel” es la música. En el juego siempre habrá música sonando de fondo pero no siempre será la misma. Hay varios archivos de música importados en el código y a través de las clases **Sfx** y **SfxFader** se reproducen y se va cambiando de uno a otro mediante un efecto de desvanecimiento para no notar un cambio brusco de melodía.

La interfaz de usuario

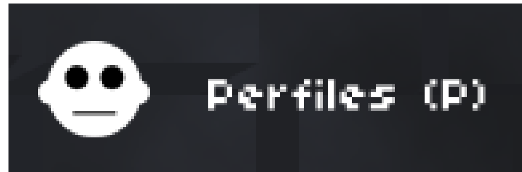
Algo que sin duda es imprescindible es la interfaz de usuario mediante la cual el usuario puede comprobar cómo va su partida, cuánta energía le queda, en qué nivel está, etc. Sin toda esta información no sería posible jugar adecuadamente al juego.

La clase encargada de todo esto es la clase “Hud”. Mediante esta clase ponemos en pantalla toda la información necesaria. Utilizamos objetos de tipo **Text** e **Image** para mostrar el nombre del jugador y su nivel así como la barra de energía y otras imágenes respectivamente.



24. El HUD de nuestro juego (fuente: propia)

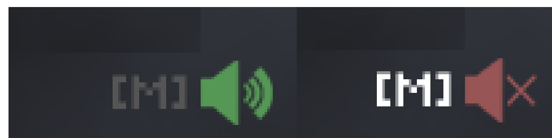
En la imagen número 24 podemos ver el "HUD" del juego. Todo lo que es texto se hace mediante objetos de tipo **Text** y algunas imágenes son del tipo **Image** y otras del tipo **Entity** ya que necesitamos de las propiedades y métodos de **Entity**. Por ejemplo, para que los botones de "perfiles" y "reiniciar nivel" funcionen como es debido, necesitamos detectar la colisión del puntero del ratón con las imágenes de dichos botones. La clase **Image** no tiene métodos para detectar colisiones mientras que la clase **Entity** sí los tiene, por lo tanto estos botones serán **Entities** cuyo atributo gráfico será una **Image**.



25. Funcionamiento del botón “perfiles” (fuente: propia)

En la imagen 25 vemos el funcionamiento del botón “perfiles”. Cuando pasamos el ratón por encima éste se ilumina y además aparece un texto de ayuda. Todo esto sucede cuando la colisión del puntero del ratón con el botón es detectada.

Un último elemento de la interfaz de usuario que queda por describir es el icono de silencio. Pulsando la tecla “M” pondremos a 0 el volumen global del juego si no está silenciado o lo habilitaremos si está silenciado. Dicho icono se encuentra en la esquina inferior derecha de la pantalla del juego, como se aprecia en la imagen número 26.



26. Funcionamiento de la tecla de silencio (fuente: propia)

Las casillas

La clase “Casilla” se ha creado para dotar de modularidad y capacidad de ampliación al juego. Si construimos los niveles a base colocar casillas, para crear más niveles solamente tendremos que indicar dónde colocar las casillas; si por el contrario utilizamos una gran imagen, para cada nivel tendremos que crear una imagen distinta. Además, colocando casilla a casilla podemos tener control absoluto e independiente de cada una de ellas y dotarlas de comportamientos distintos si quisiéramos.

27. Los tres tipos de casillas (fuente: propia)

En la clase “Casilla” se controla la posición que éstas deben tener en todo momento. Para ello se utilizan las clases **CircularMotion2** y **AngleTween** para colocar las casillas en la posición correcta respecto al centro de rotación y para girar adecuadamente la imagen de cada una de ellas respectivamente.

También tenemos la clase “CasillaVerde” que añade algunas características. Con esta clase creamos las casillas verdes y azules. Con una variable booleana controlamos, después de cada rotación, si la casilla ha de caer hacia abajo o no. En el caso de tener que caer, el movimiento se hace mediante la clase **NumTween** que nos servirá para realizar la interpolación de movimiento. Además se utiliza la clase **Ease** con su función **bounceOut** para añadir un efecto de rebote en las casillas al llegar al suelo.

También añadimos una función para calcular el ángulo que tienen las casillas respecto al centro, ya que al moverse estas casillas antes de la siguiente rotación varían su radio y su ángulo y así realizar correctamente la rotación.

El jugador

Para crear a nuestro personaje hemos creado una clase llamada “Player”. En esta clase se controlan las acciones del jugador mediante el teclado. Utilizando las clases **Input** y **Key** haremos que el personaje se mueva al pulsar las flechas de dirección, rote el cubo al pulsar *z* y *x* y utilice sus propulsores al pulsar *espacio*.

También controlaremos que el personaje no camine si está en contacto con una casilla que le impide el paso.

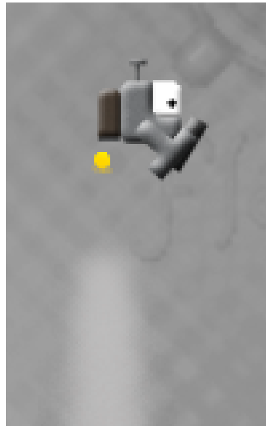
Otro aspecto a tratar es la imagen que mostraremos para nuestro jugador. En este caso utilizaremos un gráfico de la clase **SpriteMap** y así creamos las animaciones para el personaje.

28. Parte del gráfico del personaje (fuente: propia)

Así vemos en la imagen 28 un fragmento de la imagen utilizada para el personaje en la cual tenemos las animaciones. Para la animación del personaje en estado quieto

indicamos los nueve primeros *frames*, es decir, del 0 al 8. Los siguientes *frames* corresponden a la animación de correr.

Para el efecto de los propulsores se ha utilizado un emisor de partículas, es decir, la clase **Emitter**. Ha quedado configurado para que las partículas sean expulsadas hacia abajo y cambien su color desde el amarillo al naranja.



29. Los propulsores del personaje (fuente: propia)

En la imagen también se observa que el personaje al utilizar los propulsores va dejando un rastro de humo. Para lograr este efecto se ha creado una clase llamada "Humo" que lo único que hace es colocar en pantalla un gráfico de humo, redimensionarlo y aplicarle transparencia y al cabo de un determinado tiempo es eliminado del mundo. Cada objeto de tipo "Humo" es colocado en la posición del jugador en cada momento y por eso se consigue el efecto de estela o cola que podemos ver.

Otro efecto que se ha añadido al personaje es un efecto de ondas que salen de su antena.

30. Efecto de la antena (fuente: propia)

Para lograr este efecto se ha creado la clase "AntenaFX" cuyo contenido es muy similar al de la clase "Humo" pero en esta ocasión el gráfico sí sigue al personaje, variando su posición junto a éste.

Un detalle añadido para complementar el personaje es la animación de descanso. Cuando no se ha pulsado ninguna tecla durante al menos 10 segundos, el robot entra en un modo de *standby* como si durmiera.

31. Animación de descanso (fuente: propia)

Para completar el personaje se han añadido unos efectos de audio que se pueden apreciar utilizando los propulsores, corriendo, rotando el cubo o descansando.

Resto de clases

El resto de clases que no se han explicado son las clases "Titulo", Bot y "Main". La clase "Bot" se ha creado en un principio solamente para dotar de algo de vida a la pantalla de bienvenida del juego, que es la clase "Titulo". Así pues, "Bot" es un personaje en apariencia exactamente igual a "Player" pero no está controlado por el usuario. Su comportamiento es muy simple: se escoge aleatoriamente un sentido (izquierda o derecha) y el personaje corre en ese sentido hasta colisionar con una casilla o con otro "Bot". Cuando dos "Bot" chocan, saltan hacia atrás y escogen nuevamente un sentido en el que seguir corriendo.

La clase "Titulo", como se ha dicho antes, es simplemente la pantalla de bienvenida al juego. Cuando el usuario pulse una tecla cualquiera incluido el botón del ratón, se cargará el primer nivel o el nivel actual del usuario con el primer perfil.

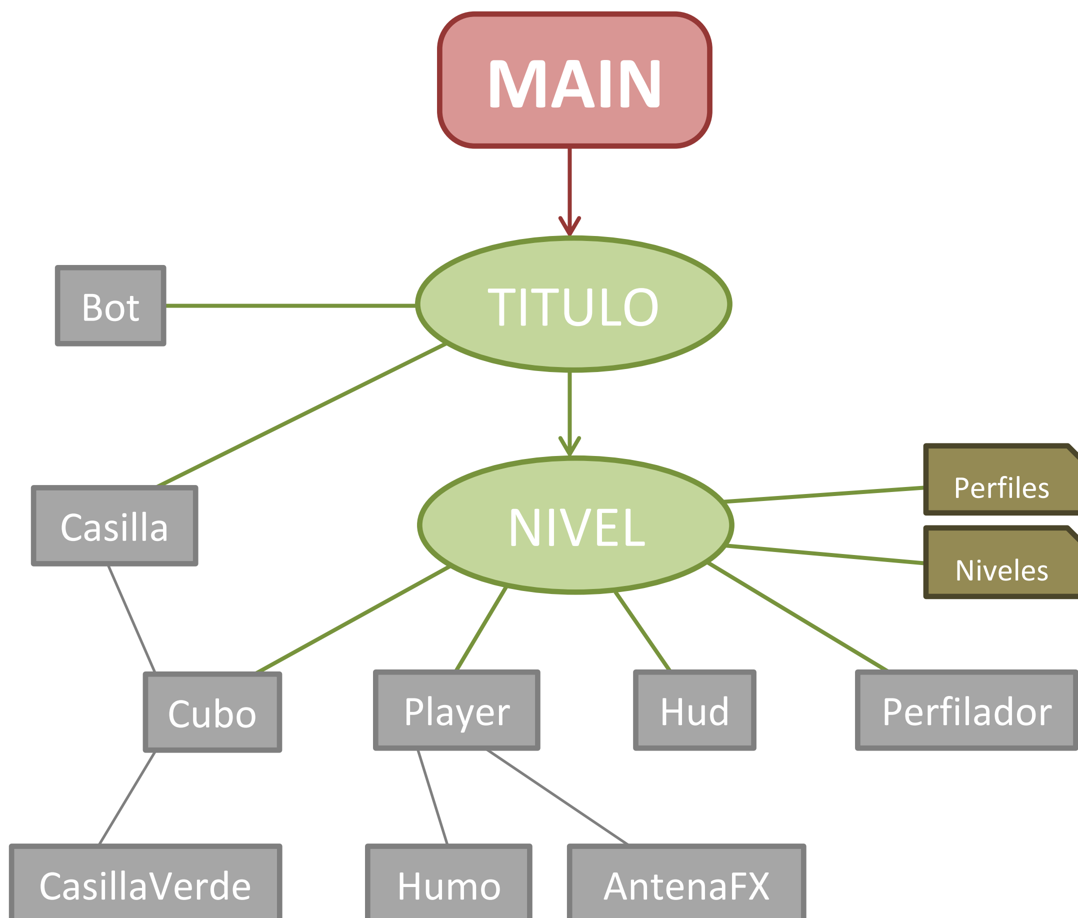
32. Los "Bots" en el "Titulo" (fuente: propia)

Por último, la clase “Main” se encarga de “construir” el juego llamando al constructor de la clase **Engine**, en el cual se indica el tamaño de la pantalla y el *framerate* que tendrá el juego. Acto seguido se establece el mundo activo, de la clase **World**, que será el primer mundo que será cargado.

La clase “Main” es necesaria, se compilará siempre, ya que es la que “crea” el juego. Podríamos cambiarle el nombre si quisiéramos, pero su contenido debe ser el mismo.

Esquema

A continuación veremos de forma sencilla mediante un esquema cómo se relacionan las clases que componen el juego.



Cuando se ejecuta el juego la clase “Main” es llamada. Ésta carga el primer mundo activo que es la clase “Titulo”.

La clase “Titulo” utiliza la clase “Bot” para crear en su mundo un número determinado de robots que actúan según su programación y sin más propósito que moverse por la pantalla. “Titulo” también utiliza la clase “Casilla” para crear las casillas que podemos ver en pantalla. Dicha clase “Titulo” establece el siguiente mundo activo del juego que es la clase “Nivel”.

La clase “Nivel” es la última de tipo **world** que cargamos, es decir que a partir de ahora todo lo que suceda es controlado por esta clase. Desde ella se accede a los archivos de los perfiles y los niveles y así poder cargarlos. También se accede a las clases “Perfilador”, para gestionar y modificar la información de los perfiles; “Hud”, para mostrar por pantalla toda la información relevante de la partida; “Player”, para crear el personaje que el usuario puede controlar; “Cubo”, para colocar en pantalla y dotar de características a las casillas que formarán los niveles del juego.

La clase “Player” utiliza las clases “Humo” y “AntenaFX” que simplemente añaden ciertos efectos en pantalla para eliminarlos posteriormente cuando ya no se necesiten.

La clase “Cubo” se sirve de las clases “Casilla” y “CasillaVerde” para construir los niveles.

6 - Análisis económico y planificación

Análisis económico

Para hacer una estimación del coste real de llevar a cabo el proyecto habría que tener en cuenta muchos factores, algunos difíciles de calcular como por ejemplo la electricidad consumida o el ancho de banda utilizado y por eso se centrará el análisis en aspectos más cercanos.

Así pues, la realización del proyecto ha tenido un coste nulo ya que puede realizarse completamente con software gratuito y ya se disponía de un ordenador capaz de llevar a cabo las tareas necesarias. También cabe destacar que los recursos tales como gráficos y sonidos pueden obtenerse gratuitamente en internet o pueden crearse con software gratuito.

Elemento	Coste
Flashpunk	0€
Flex	0€
FlashDevelop	0€
Archivos gráficos	0€
Archivos sonoros	0€
Total	**0€**

Pero esto es válido solamente en el caso de que el proyecto se realice como actividad individual desde casa de uno mismo.

La cosa puede cambiar si se trabaja para otra persona o empresa que deba pagar nuestras horas de trabajo. A continuación se hace una estimación del coste del proyecto para un caso hipotético.

Elemento	Coste
Flashpunk	0€
Flex	0€
FlashDevelop	0€
Archivos gráficos	900€
Archivos sonoros	900€
Sueldo ingeniero	3600€
Ordenador	400€
Total	**5800€**

En este caso, la persona que quiere hacer un juego se beneficia del coste nulo de **Flashpunk**, **FlashDevelop** y **Flex**. Pero quiere un mínimo de calidad en el resultado y contrata a varias personas para ello.

Suponiendo que el sueldo de los ingenieros es de 30€ por hora y éstos trabajan 30 horas semanales, al no ser un proyecto grande los ingenieros encargados de crear los gráficos y los sonidos hacen su trabajo en una semana. Esto nos da 900€ por cada uno. Luego supondremos que el ingeniero encargado de programar el juego lo hace en un mes a 30 horas semanales. No necesitará más tiempo ya que conoce **Flashpunk**, tiene experiencia y sabe lo que hace. Esto nos da 3600€ para él (30€ × 120h).

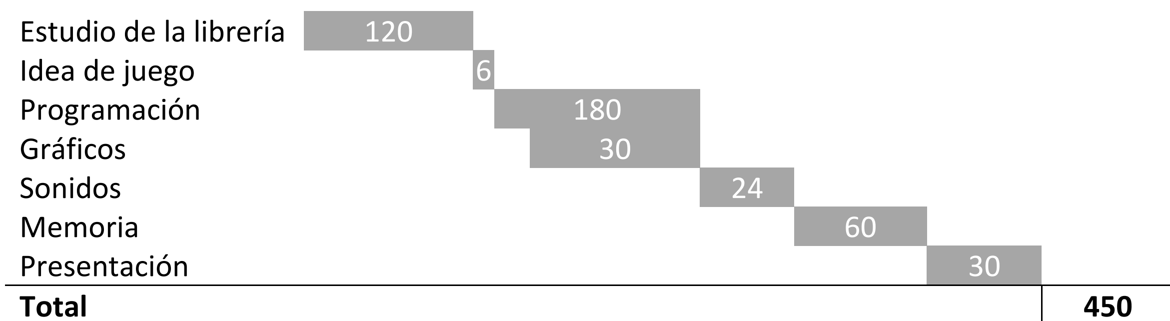
También puede ser posible que sea necesario un ordenador, y como para hacer un juego de estas características no hace falta un ordenador potente podemos estimar su coste en 400€.

Todo esto sumado nos da un coste estimado de 5800€.

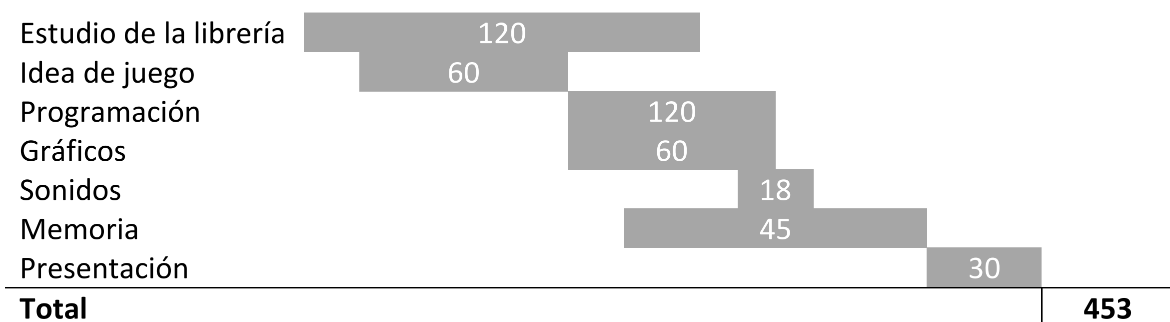
Planificación

A continuación analizaremos la planificación del proyecto. Compararemos el tiempo dedicado inicialmente pensado con el que ha resultado ser finalmente.

Tiempo dedicado inicialmente pensado:



Tiempo dedicado final:



Inicialmente se habían dividido las **450** horas disponibles en:

- Un mes (120 horas) para el estudio de la librería.
- Un día (6 horas) para pensar la idea sobre la que hacer el juego.
- Un mes y medio (180 horas) para programar.
- Una semana (30 horas) para obtener los gráficos necesarios.
- Cuatro días (24 horas) para obtener los sonidos.
- Dos semanas (60 horas) para redactar la memoria del proyecto.
- Una semana (30 horas) para preparar la presentación.

Finalmente se han empleado **453** horas de la siguiente manera:

- Un mes (120 horas) para el estudio de la librería.

  Este tiempo no ha variado pero se ha compaginado sobre todo con la programación ya que a medida que se iban utilizando las clases de **Flashpunk** iban surgiendo nuevos problemas y dudas que se han solucionado consultando la documentación y la comunidad.

- Dos semanas (60 horas) para pensar la idea sobre la que hacer el juego.

  Este tiempo se ha visto dramáticamente aumentado debido a que varias pruebas iniciales no obtenían el resultado deseado y esto ha retrasado la etapa de programación.

- Un mes (120 horas) para programar.

  La elaboración del código, una vez clara la idea y una vez claro el funcionamiento de **Flashpunk**, no ha sido en ningún caso excesivamente difícil y este periodo se ha visto acortado.

- Dos semanas (60 horas) para obtener los gráficos necesarios.

  Esta etapa ha requerido más tiempo puesto que los gráficos hallados en internet no se adecuaban a los requisitos del juego y se han creado con software de edición gráfica. No obstante, se ha compaginado con la etapa de programación.

- Tres días (18 horas) para obtener los sonidos.

  Esta etapa se ha podido reducir en tiempo debido a que en internet hay recursos realmente buenos que no han requerido mucho esfuerzo para obtener sonidos que cumplieran con lo deseado.

- Semana y media (45 horas) para redactar la memoria del proyecto.

  En un poco menos de lo esperado se ha podido redactar la memoria ya que se ha compaginado con otras áreas del proyecto y se ha ido completando a rachas.

- Una semana (30 horas) para preparar la presentación.

  Este periodo no se ha visto alterado puesto que se ha realizado cuando ya se habían completado todas las demás áreas del proyecto y se tenía claro qué hacer.

En definitiva, podemos comprobar que prácticamente se ha cumplido el objetivo de las 450 horas. Se han obtenido tan sólo 3 horas más y si no se hubiera gastado tanto tiempo en la etapa de obtener una idea se hubiera podido ajustar completamente el tiempo total.

7 - Posibilidades de ampliación

El estado final del proyecto deja la puerta abierta para diversas ampliaciones. Siempre hay aspectos a mejorar. Podría revisarse el código en busca de posibles optimizaciones, podrían reemplazarse los gráficos por unos mejor elaborados y lo mismo con los sonidos.

Respecto a ampliar el juego, una opción podría ser añadir más tipos de casillas, por ejemplo unas rompibles o unas que el jugador no pudiera tocar. Otra opción sería añadir ítems y así aumentar la variedad de niveles ya que si por ejemplo ponemos objetos que dieran energía extra al personaje esto daría más juego a la hora de pensar niveles.

Otra opción complementaria y bastante interesante sería crear un editor para el juego. Dicho editor constaría de una vista parecida a la del juego, el nivel a un lado y la información con las casillas y objetos al otro. Añadiríamos los objetos deseados al nivel, ya sean casillas, posiciones de entrada o de salida. Una vez tuviéramos el nivel a nuestro gusto, el editor debería ser capaz de guardarlo en un archivo **XML** para posteriormente poder probarlo en el juego o incluso en el mismo editor.

8 - Conclusión

Crear un juego en **Flash** puede llegar a ser algo complicado. Son necesarios conocimientos del entorno **Flash** y de **ActionScript**. Pero con **Flashpunk** no necesitamos tanto conocimiento, únicamente se necesitan habilidades de programación y aunque es cierto que se utiliza **ActionScript**, si se ha programado antes no es difícil adaptarse a la sintaxis de dicho lenguaje.

Flashpunk pone a nuestra disposición una serie de clases útiles que nos facilitarán enormemente la tarea de programar un juego. En ningún caso supone una limitación respecto a lo que se puede conseguir con **ActionScript** y **Flash**. Se puede hacer uso de cualquier clase existente, simplemente son una serie de clases y métodos extra ya programados, nadie nos obliga a usarlos.

Flashpunk es adecuado para crear juegos de estilo retro ya que la forma de crear personajes animados, mediante mapas de *sprites*, es la manera en cómo se hacían antes. Por lo tanto, si nuestro objetivo es crear un juego de este estilo, Flashpunk es ideal para ello. Pero si por el contrario pretendemos crear algo más actual, más vistoso, **Flashpunk** no nos sirve. Acabaríamos por no usarlo, programando nuestras propias clases o usando el entorno **Flash**.

En resumidas cuentas, **Flashpunk** es ideal para crear juegos 2D sencillos y mucha gente que no posee conocimientos sobre **Flash** o **ActionScript** puede animarse a crear algún juego que de otra manera no sabrían cómo empezar. Aquí lo importante es tener una idea y ganas de llevarla a cabo.

Además, la comunidad de **Flashpunk** pone a nuestra disposición gran cantidad de tutoriales, ejemplos y resolución de dudas que serán una valiosísima ayuda cuando tengamos problemas, tanto si estamos empezando y tenemos dudas de novato como si somos usuarios más experimentados y nuestras dudas sean más serias.

Por lo tanto, siempre que nuestro objetivo esté acorde con la filosofía de **Flashpunk**, éste nos servirá perfectamente para conseguirlo y se recomienda 100%.

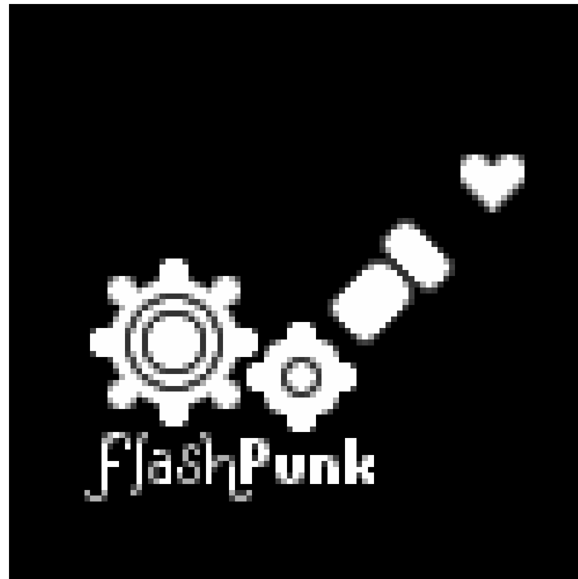
Si se quiere ir más allá, se tiene que recurrir a otras librerías. **Flashpunk** no incorpora ningún motor de físicas ni código de red para jugar online o comunicarse con algún servidor. Por ejemplo, un motor de físicas para **Flash** bastante popular es **Box2D**, cuya web es:

http://www.box2dflash.org/

Para el juego online puede servirnos **PlayerIO**, que proporciona el código y el servidor. Su web es:

http://playerio.com/

Hasta aquí la experiencia con **Flashpunk**, la cual ha sido positiva y recomendable si se tiene interés en la creación de videojuegos.



33. Logo de Flashpunk (fuente: página web de Flashpunk)

9 - Bibliografía

Página web: http://flashpunk.net

Página web: http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html

Página web: http://www.adobe.com/products/flex/

Página web: http://digitaltools.node3000.com/interview/2303-interview-chevy-ray-johnston-flashpunk

Página web: http://www.alegsa.com.ar

Página web: http://es.wikipedia.org

Página web: http://www.w3c.es

Página web: http://www.freesound.org

Página web: http://www.brighthub.com

Página web: http://www.emanueleferonato.com/

Página web: http://flashgamedojo.com/

10 - Imágenes

Imagen 1, 33: http://flashpunk.net
Imagen 2: http://www.xinterface.net/tag/flex/
Imagen 3: http://www.flashdevelop.org
Imagen 4: http://www.flickr.com/photos/navaboo/5495863837/
Imagen 5: http://flashgamedojo.com/go/
Imagen 6: http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4
Imagen 7: http://www.adobe.com/support/flashplayer/downloads.html
Imagen 14: http://flashpunk.net/2011/05/animated-sprites/
Imagen 15: http://producerism.com/blog/flashpunk-dame-and-lua-tutorial-part-6/
Imagen 16: http://forums.create.msdn.com/forums/t/4934.aspx
Imagen 18, 19: http://es.wikipedia.org/wiki/Curva_de_B%C3%A9zier