Memoria Trabajo Fin de Grado

Desarrollo de videojuegos en 2D
usando Simple DirectMedia Layer, SDL

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

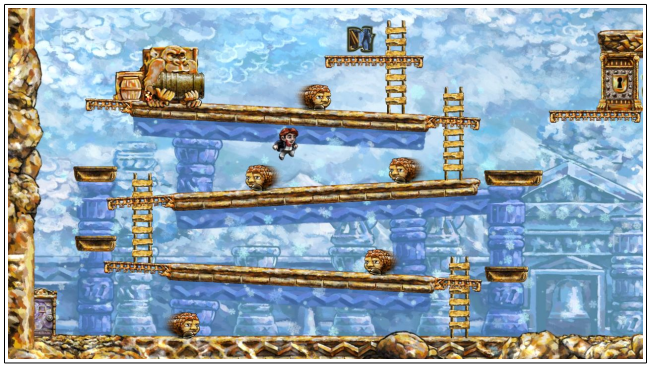
Autor: Jorge Santiago Buffa Casalis
Tutor: Agustín Rafael Trujillo Pino
Departamento de Informática y Sistemas
Las Palmas de Gran Canaria, Julio 2013

Índice de contenido

1. Introducción

Los Indie games, o videojuegos independientes, son aplicaciones creadas por individuos o pequeños grupos sin apoyo financiero de distribuidores. A menudo se centran en la innovación, y se basan en la distribución digital. En los últimos años han visto un gran aumento principalmente debido a; nuevos métodos de distribución en línea (Steam Greenlight[1], Xbox Live[2], Playstation Store[3], Android Market, Apple Store), nuevas formas de financiación como Kickstarter[4] y potentes herramientas gratuitas para el desarrollo.

Algunos de estos juegos se han convertido en un gran éxito financiero, entre ellos podemos encontrar a:



"Braid"[5], creado por el desarrollador independiente Jonathan Blow para el servicio Xbox Live Arcade de Xbox 360, en la actualidad se ha portado a Windows, Mac, PlayStation 3 y Linux. El creador a llegado a comentar que durante el desarrollo que duró unos tres años y cinco meses y gastó unos 180.000$ de su propio bolsillo para financiarlo. Fue comprado por más de 55,000 personas durante su primera semana a la venta.

"World of Goo" [6], producido por 2D Boy, una desarrolladora de videojuegos independiente compuesta por Kyle Gabler y Ron Carmel, ambos ex empleados de Electronic Arts. Publicado para la consola Wii, y para los tres principales sistemas operativos de PC: Windows, Mac OS X y Linux. En su desarrollo se emplearon un buen número de tecnologías de código abierto como Simple DirectMedia Layer, Open Dynamics Engine y TinyXML. Subversion y Mantis Bug Tracker se utilizaron para el proceso de coordinación. El juego fue creado por un equipo muy reducido, con tan solo tres integrantes. Llegaron al millón de descargas en la App Store de iOS y Mac en los primeros trece meses[7].

1 http://www.eurogamer.es/articles/2012-09-01-un-repaso-a-steam-greenlight-que-es-y-que-juegos-hay
2 http://xbox.create.msdn.com/en-US/
3 http://www.zonared.com/noticias/anade-apartado-indie-playstation-store/
4 http://www.kickstarter.com/blog/over-100-million-pledged-to-games
5 http://es.wikipedia.org/wiki/Braid_(videojuego)
6 http://es.wikipedia.org/wiki/World_of_Goo
7 http://www.gamasutra.com/view/news/39557/World_of_Goo_reaches_1_million_App_Store_downloads.php

"Super Meat Boy" [8],desarrollado por Team Meat y sucesor del juego en Flash para navegador Meat Boy, de Edmund McMillen y Tommy Refenes, se lanzó para Xbox Live Arcade el 20 de octubre de 2010, y para Windows el 30 de noviembre de 2010, y en Mac OS X un año después, en Noviembre de 2011. Según los desarrolladores, el juego supone "un gran retorno a muchos clásicos difíciles de NES como *Ghosts'n Goblins*, *Mega Man (videojuego)*, y la versión japonesa de *Super Mario Bros. 2"*[9]. El 3 de Enero de 2012 el "Team Meat", desarrolladores, anunciaron por su Twitter oficial que había superado el millón de ventas.

Este proyecto se puede considerar como uno de los primeros pasos en el desarrollo de videojuegos independientes o Indie. Usando SDL como base se pretende diseñar y desarrollar dos prototipos de videojuego, el primero será un clon del conocido Tetris, aprovechando la poca complejidad de las mecánicas del juego para tener un primer contacto con las herramientas. El segundo, de mayor complejidad, se centrará en desarrollar las principales características de un juego tipo plataformas en 2D, del estilo Super Mario, Sonic o los anteriormente mencionados Super Meat Boy y Braid.

La Simple DirectMedia Layer (SDL) es un conjunto de bibliotecas desarrolladas en el lenguaje de programación C, que proporcionan funciones básicas para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, además de carga y gestión de imágenes. Fueron desarrolladas inicialmente por Sam Lantinga en 1998, en este proyecto se ha usado la versión 1.2.15 y se espera que este año se termine la versión 2.0, el uno de Junio de este año ha alcanzado el estado "Release Candidate".

Una de sus grandes virtudes es que se trata de una biblioteca multiplataforma, siendo compatible oficialmente con los sistemas Microsoft Windows, GNU/Linux, Mac OS y QNX, además de otras arquitecturas y sistemas como Sega Dreamcast, GP32, GP2X, etc. La biblioteca se distribuye bajo la licencia LGPL, lo que ha facilitado su avance y evolución.

8 http://supermeatboy.com/about/
9 http://en.wikipedia.org/wiki/Super_Meat_Boy

1.1 Objetivos

Los principales objetivos planteados para este trabajo fin de grado son los siguientes:

-Resolver los problemas más comunes que se presentan en el desarrollo de juegos en 2D.

Este es el objetivo principal del proyecto, se pretende conseguir mediante el desarrollo de dos prototipos de videojuego a través de los cuales se enfrenten los problemas característicos del desarrollo de videojuegos en 2D.

El principal problema a resolver es de que manera nos comunicaremos con el S.O. para tener acceso a los diferentes subsistemas(video, audio, E/S, etc) con los que necesitaremos interactuar para desarrollar la aplicación, para ello se utilizará la SDL.

Centrados en el desarrollo de videojuegos encontramos problemas comunes como son; el pintado de la escena, la gestión de los controles y el sonido, la construcción del bucle principal y la gestión de los diferentes elementos que forman el juego. En cuanto a videojuegos en 2D algunos de los problemas mas típicos son; representación del mapa, colisiones en 2D y manejo de sprites.

Todos estos problemas se podrían haber atenuado si se hubiese optado por usar algún entorno para la creación de videojuegos como Unity[10] o Microsoft XNA[11]. Esto se comentó en las reuniones iniciales del proyecto que se mantuvieron con el tutor, en un principio no se descartó y se dedicó tiempo a buscar alternativas. Este estudio y la justificación de usar SDL se encuentran más adelante en esta memoria, en los capítulo “Estado actual” y “SDL”.

-Mejorar las aptitudes del estudiante para el desarrollo de videojuegos.

Es mediante la puesta en práctica de los conocimientos como se adquiere experiencia, teniendo en cuenta la situación laboral en la que nos encontramos y habiendo terminado ya los estudios, el desarrollo de pequeños proyectos como este, es una buena manera de seguir formándose y mantenerse al día. Durante los estudios se cursó la asignatura Informática Gráfica Aplicada, que en cierta manera fue el primer contacto con el desarrollo de videojuegos que se tuvo, fue una buena experiencia y desde entonces se ha querido seguir con el tema, por eso se optó por un proyecto de este tipo.

-Uso, estudio y familiarización con herramientas libres relacionadas con el tema(SDL, Eclipse, Gimp, Tiled, etc).

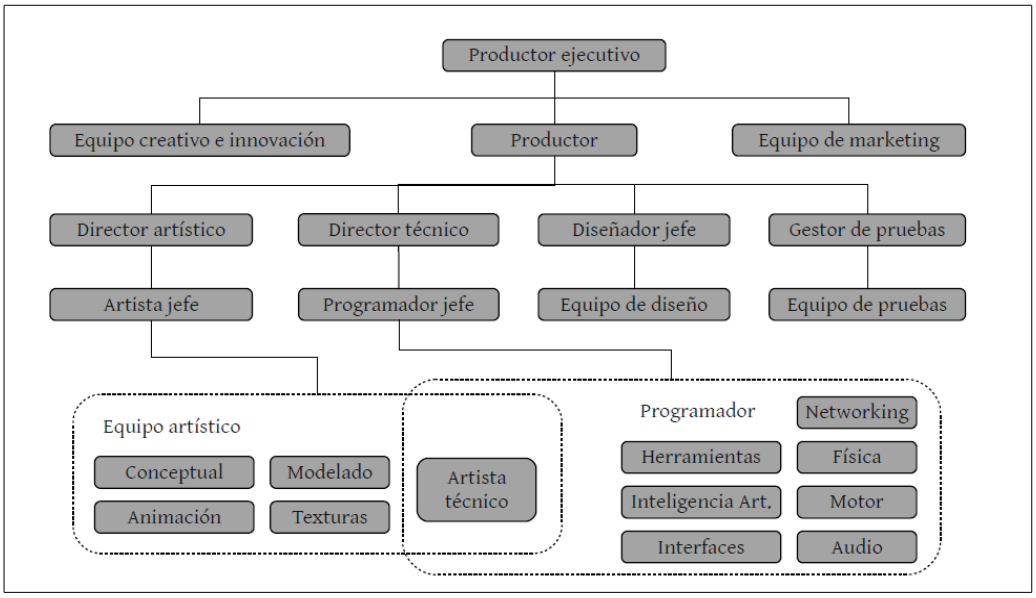
Un factor fundamental que ha permitido la aparición de videojuegos independientes, es el acceso a potentes herramientas de desarrollo sin coste, desde librerías multipropósito como SDL hasta motores de videojuegos profesionales como Unity. Entre las principales herramientas usadas están; Eclipse, IDE de trabajo, que ya se había usado en varias asignaturas, Gimp, programa de edición de imágenes, del que se tenían conocimientos básicos antes del proyecto y SDL, de la que no se conocía nada antes del proyecto.

10 http://spanish.unity3d.com/
11 http://msdn.microsoft.com/es-es/library/bb203894%28v=xnagamestudio.40%29.aspx

-Conseguir tener una visión global de todo el proceso y de los diferentes roles implicados en el desarrollo de videojuegos.

La creación de este tipo de aplicaciones es un proceso complejo en el que participan profesionales con conocimientos muy especializados en diferentes disciplinas, desde ciencias formales hasta ciencias sociales que van más allá del típico proyecto de software e implican al mismo tiempo creatividad e imaginación. Una clasificación por grupos podría ser:

- Diseñadores: deciden en qué consiste el juego y cómo se juega, establecen las reglas (ej: "si un enemigo toca al jugador este pierde una vida", "si el jugador cae por un precipicio el jugador muere") y mecánicas (ej: "el jugador puede moverse hacia los lados, saltar, disparar"). Son aquellas personas que concibieron la idea original y tienen una visión global del juego. Dentro de este grupo encontramos; diseñador de niveles, diseñador artístico, guionistas
- Programadores: escriben el software que implementa la funcionalidad pedida en el diseño (reglas y mecánicas), la tecnología necesaria para mostrar los contenidos y las herramientas para construirlos (motor de juego, motores gráficos, sistemas de juego, etc.) . Dentro de este grupo encontramos diferentes perfiles; inteligencia artificial, físicas, gráficos, interfaz de usuario, etc..
- Gráficos: Construyen los elementos visuales, los escenarios y los personajes. Realizan el diseño de modelado 2D o 3D, texturas, animaciones, maquetación, renderizado, cinemáticas, etc.
- Sonido: Diseñan y crean el ambiente sonoro, la música y melodías que escucharemos a lo largo del juego y que componen la banda sonora, las voces y efectos de sonido.



VALLEJO FERNÁNDEZ D. y MARTÍN ANGELINA C. (2012). *Arquitectura del Motor de Videojuegos.*

La competencia que impone el modelo económico actual, la postura de la sociedad y el lugar en el que se encuentra la industria actualmente, han abierto el mercado de los videojuegos de una forma espectacular, teniendo por un lado empresas importantes con equipos de desarrollo de 25 o más miembros para lograr publicar un solo videojuego comercial, pudiendo durar un proyecto más

de 3 años con presupuestos de millones de dólares[12], compitiendo con juegos hechos por pequeños grupos, incluso de hasta sólo una persona[13].

1.2 Competencias que hay que cubrir

CII01 – *Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.*

Esta competencia queda cubierta a través de todo el trabajo, concretamente en los capítulos en los que se diseñan y desarrollan los dos prototipos de juego en 2D, aplicaciones fiables, seguras y conformes a principios éticos y a la legislación y normativas vigentes.

CII02 – *Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.*

La capacidad de planificar es fundamental en cualquier actividad compleja, necesario en cualquier proyecto de ingeniería. Igual de importante es concebir la idea de proyecto, una visión global, a la vez que se descartan otras por los motivos que se hayan encontrado. En cuanto a la dirección del proyecto, se llevaron a cabo múltiples reuniones con el tutor en que se analizaba el trabajo realizado, mediante la prueba de prototipos en el laboratorio, y se proponían nuevos puntos a desarrollar. Además en la sección *"Aportaciones"* se valora el impacto económico y social.

CII04 – *Capacidad para elaborar el pliego de condiciones técnicas de una instalación informática que cumpla los estándares y normativas vigentes.*

Esta capacidad se desarrolla en el capítulo "Pliego de condiciones técnicas" de esta memoria.

CII18 – *Conocimiento de la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional.*

El capítulo "Normativa y Legislación" de esta memoria cubre esta competencia, analizando la normativa y la regulación de la informática en los ámbitos nacional, europeo e internacional.

TFG01 – *Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.*

La realización de esta memoria como consecución del trabajo fin de grado presentado, "Desarrollo de videojuegos en 2D usando Simple DirectMedia Layer, SDL" cubre esta competencia.

12 http://leviathyn.com/games/opinion/2013/04/12/the-costs-of-aaa-games-development/
13 http://en.wikipedia.org/wiki/Fez_%28video_game%29

1.3 Aportaciones

El desarrollo de este trabajo de fin de Grado, ha servido para mejorar a mis conocimientos en la planificación, dirección y realización de proyectos. Se pretende conseguir una visión global tanto del desarrollo de videojuegos como de la realización de un proyecto, desde la concepción de la idea hasta la redacción de la memoria. Aportará al estudiante nuevos conocimientos relacionados con el tema principal, gracias por un lado al estudiado de diferentes tecnologías para el desarrollo y por otro gracias a toda la documentación consultada en libros, blogs, videos y demás soportes, sobre diferentes técnicas para la solución de los problemas que se generaban.

El juego tradicional ha sido siempre una importante herramienta de aprendizaje de conductas, actitudes sociales y prácticas, es uno de los métodos de transmisión de aprendizaje más eficaces. Esta consideración ha ido cambiando a lo largo del tiempo y lo ha relegado a un mero objeto de ocio. El juego durante el aprendizaje ofrece la ventaja de asimilar conceptos a través de la experiencia, de las propias acciones y de las de los otros. Los videojuegos aportan la ventaja añadida de desarrollar habilidades y destrezas propias de esta época.

Son además un instrumento tecnológico que están plenamente integrados en la sociedad, son un vehículo de cultura. En la actualidad gran parte de la sociedad tiene acceso a un dispositivo donde se puedan ejecutar videojuegos, ya sea una consola, un teléfono inteligente o un ordenador personal.

La industria del videojuego ha mantenido en los últimos años su posición como principal industria de ocio audiovisual e interactivo, con una cuota de mercado muy superior a la del cine y la música. A pesar del contexto económico y de su efecto en el consumo, el sector del videojuego ha mantenido su apuesta por la innovación, adaptándose a las nuevas tendencias de consumo y encontrando nuevos segmentos de mercado para favorecer el crecimiento de negocio.

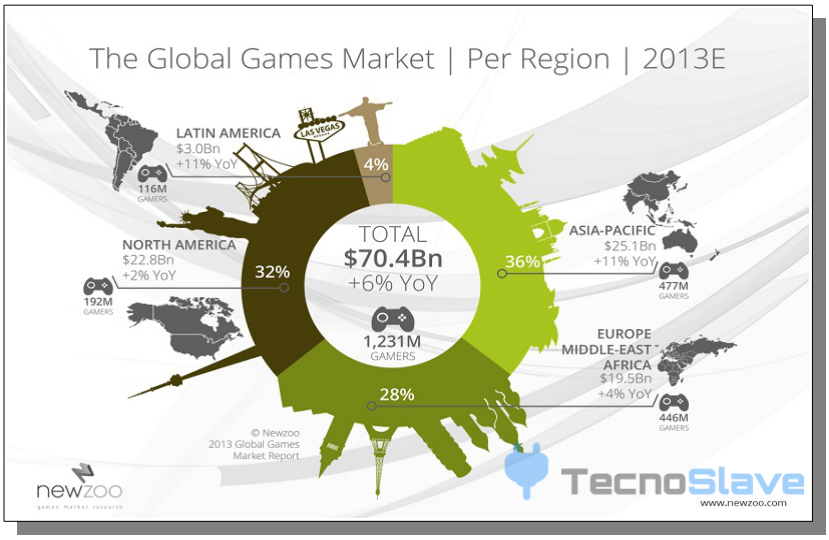
Por este motivo, los principales estudios de mercado la sitúan como la industria tecnológica con mayor proyección de crecimiento. Los analistas aseguran que el sector será uno de los protagonistas en la denominada "economía de la innovación" gracias principalmente al desarrollo del canal online de ventas y la multiplicación de las funcionalidades del videojuego, que comienza ya a alcanzar nuevos ámbitos más allá del ocio.

El Strategic Research Center de EAE Business School presenta el estudio "El gasto en videojuegos en España 2012"[14], uno de los muchos estudios de la situación del mercado de videojuegos en España, tanto del total del país como a nivel autonómico, y a escala internacional, observando además la tendencia de este mercado de ocio para el periodo 2013 – 2016. En él se pueden analizar muchos de los números que genera este mercado, a continuación se comentan los más interesantes.

El tamaño del mercado mundial de videojuegos en el año 2012 es de 31.909 millones de euros. Desde 2007 ha crecido todos los años (excepto en 2009, cuando cayó más de un 8%), mostrando porcentajes de crecimiento entorno al 10% en 2010, 2011 y 2012. Las previsiones para el periodo 2013 – 2016 muestran que el mercado global de videojuegos pasará de los 31.909 millones de euros de 2012 a los 47.024 millones en 2016, un crecimiento del 47% en apenas cuatro años y un crecimiento medio anual superior al 10%. Según el estudio de EAE, en 2016 los principales

14 http://www.eae.es/news/2013/02/18/el-mercado-del-videojuego-en-espana-crecera-un-20-los-proximos-cuatro-anos

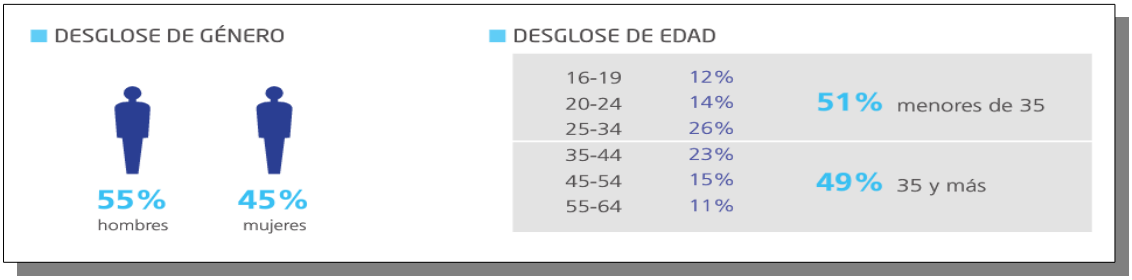
mercados serán EEUU, Japón y Alemania con 15.000, 4.200 y 3.300 millones de euros de gasto respectivamente



Mercado global por regiones.

En España, el 91,7% de los videojuegos que se venden se han desarrollado para videoconsolas, siendo el restante 8,3% del mercado para videojuegos desarrollados para ordenador, bien sea PC o para Mac. En lo que se refiere a la distribución de estos productos, el estudio de EAE pone de manifiesto que se encuentra muy concentrada en tres canales: distribuidores de electrónica (con un 56,4% del mercado), distribuidores de música, video y libros (con un 19,3%) y supermercados e hipermercados (con un 13,8%). Se prevé que el mercado español de videojuegos para los años 2013 – 2016 sea de 824 millones de euros, un crecimiento del 20,27% respecto la cifra actual.

Nos encontramos ante un mercado en el que, en estos momentos y a medio plazo, existen grandes oportunidades. Además los videojuegos como productos, en estos años, generan un gran interés en la sociedad no sólo en los jóvenes sino a todas edades y género.



El Anuario del Videojuego 2012, Asociación Española de Distribuidores y Editores de Software de Entretenimiento.

Teniendo en cuenta la cantidad de potentes herramientas gratuitas de desarrollo, nuevas formas de distribución y opciones de financiación colectiva[15], nos encontramos ante una opción profesional interesante a tener en cuenta.

15 Estos tres temas se tratan durante la memoria, las herramientas en el capítulo sobre SDL, y los nuevos canales de distribución y financiación en el capítulo "Estado actual", en el apartado "Actualidad".

2. Estado Actual

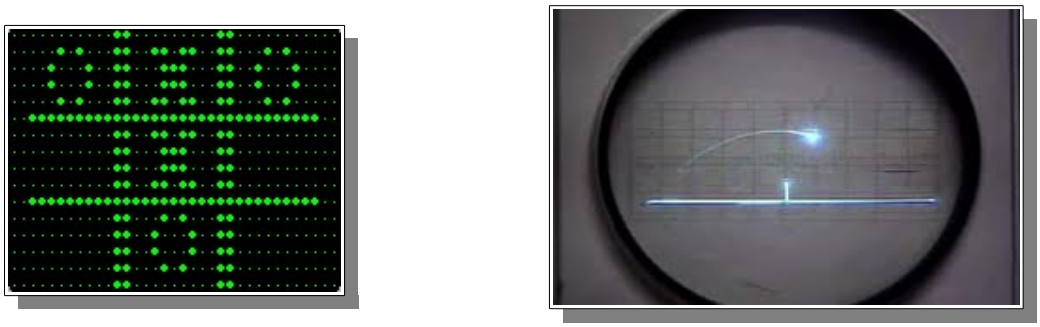
2.1 Historia de los videojuegos

La historia de los videojuegos tiene su origen en la década de 1940 cuando, tras el fin de la Segunda Guerra Mundial, las potencias vencedoras construyeron las primeras supercomputadora programables como el ENIAC, de 1946. Los primeros intentos por implementar programas de carácter lúdico (inicialmente programas de ajedrez) no tardaron en aparecer, y se fueron repitiendo durante las siguientes décadas. Los primeros videojuegos modernos aparecieron en la década de los 60, y desde entonces el mundo de los videojuegos no ha cesado de crecer y desarrollarse con el único límite que le ha impuesto la creatividad de los desarrolladores y la evolución de la tecnología.

En los últimos años, se asiste a una era de progreso tecnológico dominada por una industria que promueve un modelo de consumo rápido donde las nuevas superproducciones quedan obsoletas en pocos meses, pero donde a la vez un grupo de personas e instituciones -conscientes del papel que los programas pioneros, las compañías que definieron el mercado y los grandes visionarios tuvieron en el desarrollo de dicha industria- han iniciado el estudio formal de la historia de los videojuegos.

El más inmediato reflejo de la popularidad que ha alcanzado el mundo de los videojuegos en las sociedades contemporáneas lo constituye una industria que genera unos beneficios multimillonarios que se incrementan año tras año como se ha visto en el capítulo de "Aportaciones". El impacto que supuso la aparición del mundo de los videojuegos significó una revolución cuyas implicaciones sociales, psicológicas y culturales constituyen el objeto de estudio de toda una nueva generación de investigadores sociales que están abordando el nuevo fenómeno desde una perspectiva interdisciplinar, haciendo uso de metodologías de investigación tan diversas como las específicas de la antropología cultural, la inteligencia artificial, la teoría de la comunicación, la economía o la estética, entre otras. Al igual que ocurriera con el cine y la televisión, el videojuego ha logrado alcanzar en apenas medio siglo de historia el estatus de medio artístico, y semejante logro no ha tenido lugar sin una transformación y evolución constante del concepto mismo de videojuego y de su aceptación.

Los primeros videojuegos eran mayoritariamente experimentos y pruebas académicas por parte de científicos y físicos, no sería hasta la llegada de la década de los 70 cuando los videojuegos empezarían a tener un carácter más comercial, marcando el inicio de una imparable transformación y evolución del concepto mismo del videojuego que acabaría por convertirlo en el fenómeno cultural de masas que hoy conocemos.



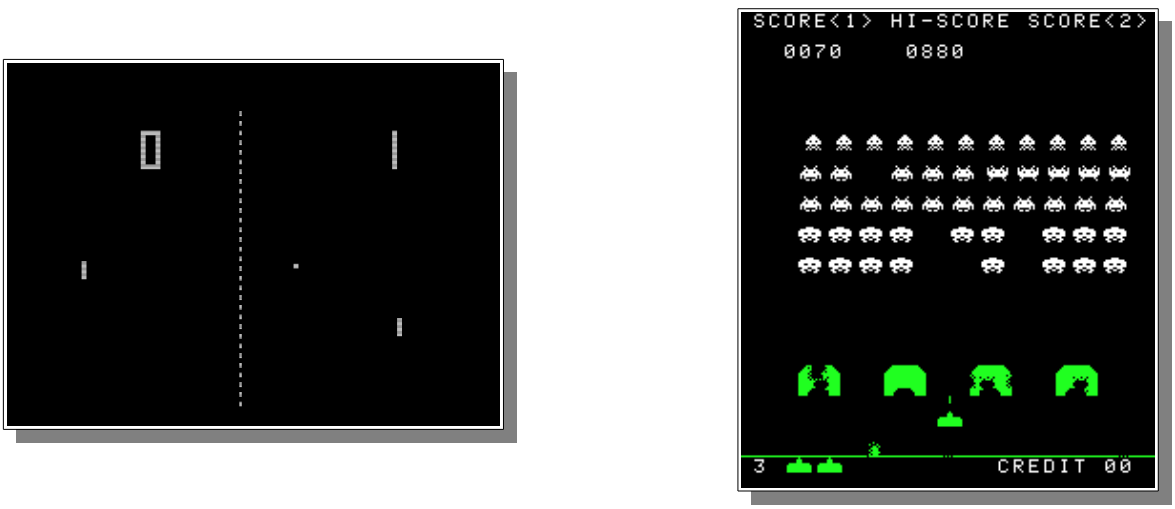
Izquierda "OXO" y derecha "Tennis for two", considerados unos de los primeros videojuegos de la historia.

En 1966 Ralph Baer empezó a desarrollar junto a Albert Maricon y Ted Dabney un proyecto de videojuego llamado Fox and Hounds dando inicio al videojuego doméstico. Este proyecto evolucionaría hasta convertirse en la Magnavox Odyssey, el primer sistema doméstico de videojuegos lanzado en 1972 que se conectaba a la televisión y que permitía jugar a varios juegos pregrabados.

Magnavox Odyssey, primero videoconsola de la historia.

Un hito importante en el inicio de los videojuegos tuvo lugar en 1971 cuando NolanBushnell comenzó a comercializar Computer Space, una versión de Space War, en Estados Unidos, aunque es posible que se le adelantara Galaxy War otra versión recreativa de SpaceWar aparecida a principios de los 70 en el campus de la universidad de Standford.

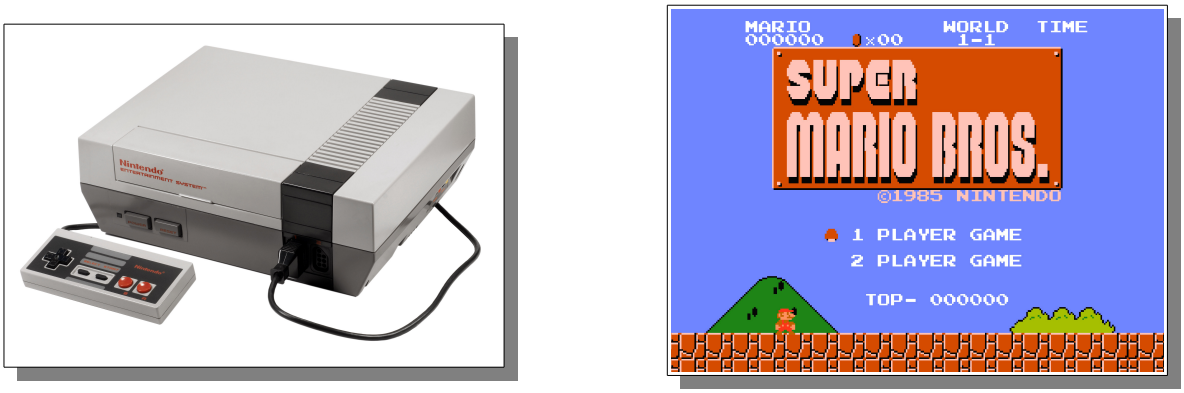
La ascensión de los videojuegos llegó con la máquina recreativa Pong, muy similar al Tennis for Two pero utilizada en lugares públicos: bares, salones recreativos, aeropuertos, etc. El sistema fue diseñado por Al Alcorn para Nolan Bushnell en la recién fundada Atari. Space Invaders se presentó en 1972 y fue la piedra angular del videojuego como industria. Durante los años siguientes se implantaron numerosos avances técnicos en los videojuegos (destacando los microprocesadores y los chips de memoria), aparecieron en los salones recreativos juegos como Space Invaders (Taito) o Asteroids (Atari) y sistemas domésticos como el Atari 2600.



Izquierda "Pong" y derecha "Space Invaders".

En el verano de 1982 la fiebre por los videojuegos aumentó considerablemente. Desde que *Space Invaders* irrumpió en el mercado los ingresos generados por la industria habían pasado de los 454 millones de dólares de ese año hasta los 5 313 millones de 1982, es decir, estaba incrementando

sus beneficios un 5% mensualmente. El interés del gran público por los videojuegos parecía no tener fin y las máquinas recreativas se encontraban por doquier, desde los bares y restaurantes hasta los hoteles y supermercados. Durante la década de los 80 Atari, Mattel, Nintendo y Sega protagonizaron una guerra que generó consolas como la Mattel Intellivision 1980, la Atari 5200, la Coleco Colecovision, la Sega SG-1000, la Nintendo NES, la Sega Master System, la Atari 7800 y la NEC Turbografx-16. De estas consolas la más exitosa fue la Nintendo Entertainment System de 8 bits, que gracias a su juego Mario Bros. alcanzó el liderazgo.



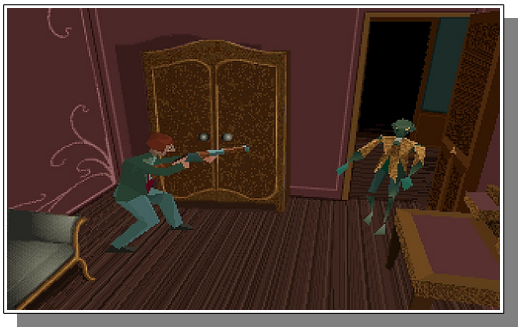
Izquierda consola "N.E.S." de Nintendo, derecha el juego "Super Mario Bros." para esa consola.

El negocio asociado a esta nueva industria alcanzó en poco tiempo grandes cotas. Sin embargo, en 1983 comenzó la que se ha dado por llamar crisis del videojuego, la cual afectó principalmente a Estados Unidos y Canadá, y que no llegaría a su fin hasta 1985.

En el resto del mundo se produjo una polarización dentro de los sistemas de videojuegos. Japón apostó por el mundo de las consolas domésticas con el éxito de la NES (Nintendo Entertainment System), mientras que Europa se decantaba por los microordenadores como el Commodore 64 o el Spectrum.

A principios de los años 90 las videoconsolas dieron un importante salto técnico gracias a la competición de la llamada "generación de 16 bits" compuesta por la Mega Drive, la SuperFamicom de Nintendo conocida en occidente como Super Nintendo Entertainmet System "SNES", la PC Engine de NEC, conocida como Turbografx en occidente y la CPS Changer de Capcom.

Mientras tanto diversas compañías habían comenzado a trabajar en videojuegos con entornos tridimensionales, principalmente en el campo de los PC, obteniendo diferentes resultados desde las "2D y media" de Doom, 3D completas de 4D Boxing a las 3D sobre entornos pre-renderizados de Alone in the Dark.



Izquierda "Doom", derecha "Alone in the dark".

Referente a las ya antiguas consolas de 16 bits, su mayor y último logro se produciría en la SNES mediante la tecnología 3-D de pre-renderizados de SGI, siendo su máxima expresión juegos como Donkey Kong Country y Killer Instinct. También surgió el primero juego poligonal en consola, la competencia de la SNES, Mega-Drive, lanzó el Virtua Racing, que tuvo un gran éxito ya que marco un antes y un después en los juegos 3D en consola.

Izquierda "Killer Instinct" y derecha "Virtua Racing".

Rápidamente los videojuegos en 3D fueron ocupando un importante lugar en el mercado, principalmente gracias a la llamada "generación de 32 bits" en las videoconsolas: Sony PlayStation, Sega Saturn (que tuvo discretos resultados fuera de Japón); y la "generación de 64 bits" en las videoconsolas: Nintendo 64y Atari jaguar. En cuanto a los PC, se crearon las aceleradoras 3D que permitían un gran salto en la capacidad gráfica de los juegos.

"Mario 64" para la consola Nintendo 64,

Super Mario 64, el primer juego de plataformas en 3D de la saga de *Mario*, estableció un nuevo arquetipo para el género (gracias a su forma de juego, totalmente libre, y el formato de sus gráficos 3D poligonales mezclados con sprites bidimensionales) tal como *Super Mario Bros.* lo hizo para los juegos de plataformas en dos dimensiones. El juego ha sido llamado «revolucionario» y no sólo ha influenciado a muchos grandes juegos en su género, sino que también ha tenido un duradero impacto en los juegos 3D en general.

Desde mediados de la década de 1990 la industria de las máquinas recreativas estaba en crisis. Este tipo de aparatos siempre había basado su éxito en el hecho de disponer de una tecnología y una potencia audiovisual muy por encima de las capacidades de los microordenadores personales y de las consolas domésticas. La Neo-Geo de SNK permitía al jugador disfrutar en su casa de la misma tecnología que exhibían las máquinas de los salones recreativos, pero era una consola muy cara sólo al alcance de unos pocos. El éxito de la Playstation y las consolas de su generación (Nintendo 64 y Sega Saturn) desequilibró definitivamente esta situación, pues las nuevas máquinas domésticas igualaban e incluso superaban tecnológicamente a la mayoría de recreativas. Por otro lado el rápido crecimiento del sector de la telefonía móvil, con aparatos cada vez más potentes que permitían ejecutar videojuegos de creciente complejidad, sentenció definitivamente las máquinas que habían protagonizado el inicio de la revolución de los videojuegos obligando a los dueños de los salones recreativos a reorientar su modelo de negocio.

Por su parte los videoconsolas portátiles, producto de las nuevas tecnologías más potentes, comenzaron su verdadero auge, uniéndose a la Game Boy (Nintendo) máquinas como la Game Gear (Sega), la Lynx(Atari) o la Neo Geo Pocket (SNK), aunque ninguna de ellas pudo hacerle frente a la popularidad de la Game Boy, siendo esta y sus descendientes (Game Boy Pocket, Game Boy Color, Game Boy Advance, Game Boy Advance SP, Game Boy Micro) las dominadoras del mercado.

Game Boy, de Nintendo.

En 2000 Sony lanzó la anticipada PlayStation 2 , un aparato de 128 bits que se convertiría en la videoconsola más vendida de la historia, mientras que Microsoft hizo su entrada en la industria un año más tarde con su X-Box, una máquina de características similares que sin embargo no logró igualar su éxito. De 2001 fue también la GameCube, una nueva apuesta de Nintendo que no consiguió atraer al público adulto y cuyo fracaso comercial supuso un replanteamiento de la estrategia comercial de la compañía nipona. Mientras tanto, el mercado de los PC seguía dominado por esquemas de juego que ya habían hecho su aparición con anterioridad. Triunfaban los videojuegos de estrategia en tiempo real (Warcraft, Age of Empires) y los juegos de acción en línea (Call of Duty, Battlefield 1942).

El final de 2005 vio el lanzamiento de la Xbox 360, la primera de la séptima generación de consolas de videojuegos. El años 2006 marca la continuación de lanzamientos de la nueva generación en la forma de dos nuevas consolas. Sony con su PlayStation 3 y Nintendo con la Wii.

2.2 Actualidad

En la actualidad nos encontramos con un mercado mucho más competitivo por varios factores. Por un lado el número de dispositivos capaces de ejecutar videojuegos es inmenso, cada uno de ellos con su propia arquitectura. Tenemos por un lado la “nueva” generación de consolas, Sony con PlayStation 4, Microsoft con XBOX ONE y Nintendo con Wii U. Por otro los teléfonos inteligentes y tabletas, tanto de Android como de Apple y Microsoft, que se están convirtiendo en una opción muy popular en cuanto a dispositivos portátiles.

Izquierda XBOX ONE, derecha PlayStation 4.

Tan grande ha sido la popularidad de los sistemas Android en este mercado, reflejado en las ventas de videojuegos para smartphones y tablets en los diferentes markets, que varias empresas

han empezado a comercializar pequeñas y económicas consolas que usen este sistema. Entre ellas encontramos; OUYA, videoconsola de código abierto que funciona con el SO Android 4.1, Nvidia Shield, M.O.J.O. de la empresa Mad Catz, GameStick, GamePop en la que aparte de juegos Android se podrán ejecutar de iOS gracias a su herramienta de virtualización conocida como Looking Glass, y muchas más.

Consola Android, OUYA.

Izquierda Nvidia Shield, derecha GameStick.

Por último pero no menos importante tenemos al PC como plataforma de ocio, y sus tres sistemas operativos principales Windows, Linux y Mac. En la actualidad las arquitecturas de las consolas potentes son ya muy parecidas a la de un PC, gracias a esto nos encontramos casi con los mismos juegos en los dos sistemas, quitando algunas exclusividades para consolas, cosa que sucedía en el pasado. Estos detalles sumados a que un PC es una herramienta con muchísimas más posibilidades ha conseguido que todavía se mantenga como una de las principales plataformas.

Otro factor influyente en el uso de los PC frente a consolas es el auge en los últimos años del modelo de comercialización “Free to Play” o juegos gratuitos. Este modelo comenzó a usarse en los videojuegos multijugador masivo en línea más conocidos como MMO buscando atraer la atención de los casual gamers (jugadores casuales), antes de que esta fuera ocupada por algún otro juego con mayor peso. Podemos encontrar también en los diferentes markets de teléfonos inteligentes y tabletas muchos juegos gratuitos, algunos de ellos de gran calidad. No es exactamente el mismo modelo que nos encontramos en PC ya que la mayoría de los juegos gratuitos de los markets incluyen publicidad durante el juego o no son la versión completa. En PC existen varias formas por las que la empresa consigue beneficios a través de este tipo de juego, siendo la más habitual las llamadas “microtransacciones”, mediante dinero real compramos un tipo de moneda especial dentro del juego que sirve para comprar cierta clase de objetos.

En un principio eran juegos de baja calidad gráfica, comparada con sus competidores, pero en la actualidad existen grandes empresas que publican juegos AAA bajo este sistema, por ejemplo Sony con PlanetSide 2 o Piranha Games con MechWarrior Online.

PlanetSide 2.

Mech Warrior Online.

Nos encontramos por tanto en un momento en el que el número de plataformas para las que se pueden desarrollar juegos es un factor importante, siendo ya algo común que para juegos que han tenido éxito se desarrollen versiones para la mayoría de sistemas, por ejemplo Braid, que se lanzó para Xbox Live Arcade 6 de agosto de 2008, para PC el 10 de abril de 2009, Mac 20 de mayo de 2009, PSN 17 de diciembre de 2009 y Linux 14 de diciembre de 2010. También podemos encontrar nuevos lanzamientos que estarán disponibles para varios sistemas desde el primer día, normalmente

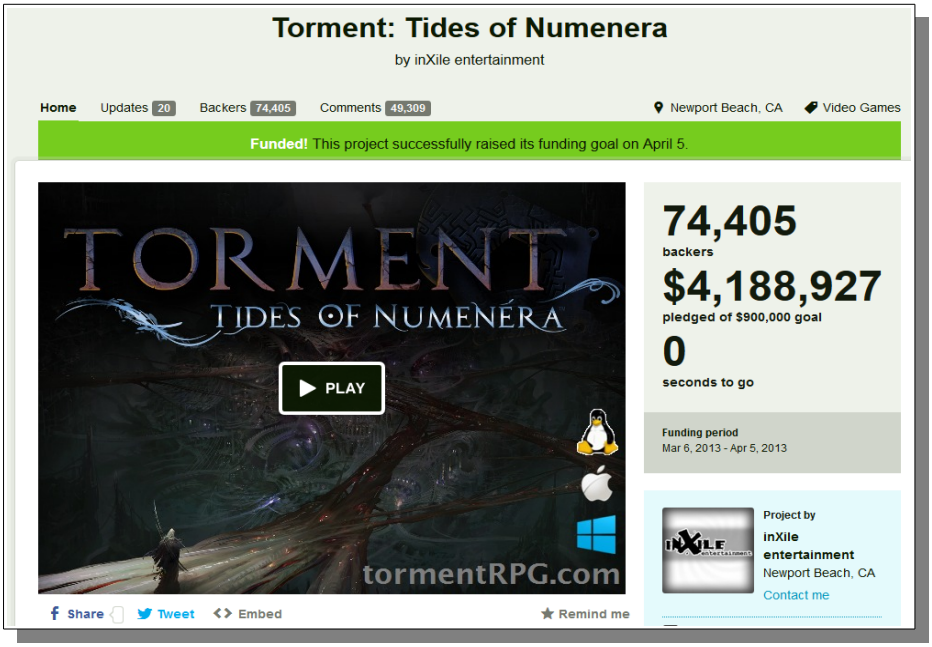
de empresas medianas-pequeñas, por ejemplo la empresa inXile Entertainment que ha conseguido financiación a través de Kickstarter para desarrollar dos videojuegos que estarán disponibles para Windows, Linux y Mac.

Otro factor que está marcando estos tiempos es lo que se ha llamado "Financiación Colectiva" o "crowdfunding", es el proceso de recaudación de "pequeñas" cantidades de dinero por un procedimiento colaborativo en red, sea a través de una web especializada o bien de modo directo. Puede ser usado para muchos propósitos, desde artistas buscando apoyo de sus seguidores, campañas políticas, financiación del nacimiento de compañías, pequeños negocios o videojuegos.

Kickstarter es un ejemplo, es un sitio web para la financiación colectiva de proyectos creativos, ha financiado una amplia gama de esfuerzos, que van desde películas independientes, música y cómics a periodismo, videojuegos y proyectos relacionados con la comida. Fue fundada en 2008 por Perry Chen, Yancey Strickler y Charles Adler, en 2013 se ha recaudado a través de su página más de 100 millones de dólares para 1.476 proyectos de videojuegos[16]. En los últimos años la cantidad de dinero recaudada para juegos ha evolucionando de manera impresionante:

- 2009: 60,601 $
- 2010: 546,362 $
- 2011: 3,855,692 $
- 2012: 83,144,565 $
- 2013: 22,423,264 $ hasta la fecha del artículo.

Gracias a este tipo de páginas los desarrolladores pueden presentar su idea ante el gran público para conseguir financiar el proyecto, normalmente en un tiempo limitado, en Kickstarter son 30 días.



Captura de la página principal de un proyecto en Kickstarter.

Este tipo de páginas sirve tanto como para pequeños desarrolladores indie, como por

16 http://www.kickstarter.com/blog/over-100-million-pledged-to-games

ejemplo el primer Kickstarter español[17] que recaudó 101,564 en los 30 días para el desarrollo de un videojuego para Pc, Mac OS, iOS, Android y Linux. Empresas de tamaño medio como inXile Entertainment con campañas en las que consiguieron asombroso números; para el desarrollo de Wasteland 2 secuela del juego Wasteland de Interplay, pedían 900.000$ consiguieron 2,933,252$[18], y para Torment: Tides of Numenera pidiendo lo mismo (900.000$) consiguieron 4,188,927$[19].

Gran parte del éxito de estos proyectos se centra en la comunicación directa de los desarrolladores con los futuros usuarios, hay que tener en cuenta que financiar un proyecto de este tipo significa comprar el juego antes incluso de que se haga, teniendo que esperar en ocasiones años hasta que este terminado. Se establece una relación muy estrecha entre los desarrolladores y las personas que han colaborado en la financiación, los primeros mantienen informados a los usuarios de los avances que se van produciendo mediante video, fotos y artículos, y los segundos proporcionan feedback a los desarrolladores a través de los foros, y llegado el momento serán los testers en las fases alpha y beta del juego.

Este sistema se ha vuelto tan popular en usuarios como en desarrolladores que incluso grandes proyectos han conseguido financiarse de esta manera. El más claro ejemplo en este sentido es la campaña llevada a cabo por el prometedor videojuego de simulación y estrategia espacial, Star Citizen[20], se ha convertido en el videojuego que más dinero ha recaudado mediante técnicas de financiación colectiva en la historia de este tipo de métodos. Empezó la financiación en su propia página web y luego creó un Kickstarter que recaudó 2,134,374$, a día de hoy todavía sigue en marcha en su web y lleva recaudados en total 11,961,129$.

Otro factor determinante en la actualidad son los medios de distribución de videojuegos. Antes de mediados de los años 1990, la distribución de videojuegos comerciales era controlada por grandes distribuidoras y minoristas, y los desarrolladores de videojuegos independientes fueron forzados a o bien formar su propia empresa distribuidora, o encontrar una dispuesta a distribuir sus videojuegos, o distribuirlos en alguna forma de shareware. En estos momentos gracias a las diferentes tiendas virtuales ya comentadas (Android Market, PS Store, ... en las que se pueden publicar creaciones sin coste o con un coste muy bajo), plataformas de distribución digital como Steam, o simplemente vendiéndolo desde su página web, los desarrolladores no dependen de grandes distribuidoras que impongan reglas, teniendo de esta forma más libertad en el desarrollo. Steam es utilizada tanto por pequeños desarrolladores independientes como grandes corporaciones de software para la distribución de videojuegos y material multimedia relacionado. Ha creado recientemente una sección llamada “Steam Greenlight” mediante la cual se facilita la entrada a la plataforma de desarrollos independientes. Incluso Amazon ha creado recientemente un canal exclusivo para videojuegos independientes, *“The Amazon Indie Games Store helps independent game developers reach more customers, and helps customers learn about more creative games and the people and processes behind building those games. “*[21].

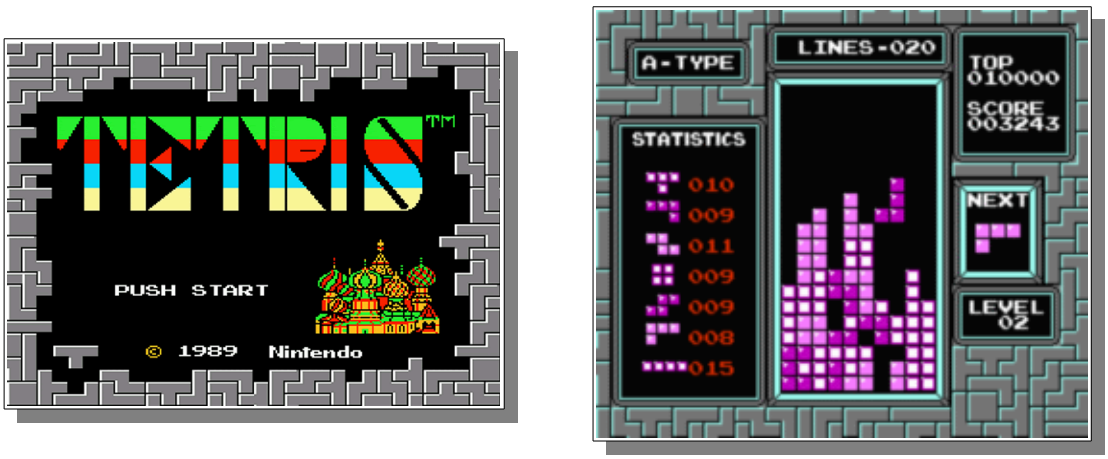
Todos estos factores; múltiples plataformas, nuevos canales de distribución, nuevos canales de financiación, sumado a potentes herramientas de desarrollo gratuitas, explican en gran parte el fenómeno indie que vivimos en estos momentos[22] y su éxito.

17 http://www.mundoreturn.com/ar-k-el-primer-juego-espanol-en-kickstarter
18 http://www.kickstarter.com/projects/inxile/wasteland-2/
19 http://www.kickstarter.com/projects/inxile/torment-tides-of-numenera
20 https://robertsspaceindustries.com/
21 http://www.amazon.com/b?ie=UTF8&node=6923534011
22 http://www.periferia.cat/2012/el-nuevo-paradigma-del-videojuego/

2.3 Tetris

Tetris es un videojuego tipo puzzle originalmente diseñado y programado por Alekséi Pázhitnov en la Unión Soviética. Fue lanzado el 06 de junio de 1984, mientras trabajaba para el Centro de Computación Dorodnitsyn de la Academia de Ciencias de la Unión Soviética en Moscú, RSFS de Rusia. El juego deriva su nombre del prefijo numérico griego tetra- (todas las piezas del juego, conocidas como Tetrominós, contienen cuatro segmentos) y del tenis, el deporte favorito de Pázhitnov.

El juego (o una de sus muchas variantes) está disponible para casi cada consola de videojuegos y sistemas operativos de PC, así como en dispositivos tales como las calculadoras gráficas, teléfonos móviles, reproductores de multimedia portátiles, PDAs y reproductores de música .



Capturas de la versión para la consola NES.

Mientras que las versiones de Tetris se vendieron para una amplia gama de plataformas de los años 1980, fue la exitosa versión portátil para el Game Boy lanzada en 1989, la que estableció al juego como uno de los más populares de todos los tiempos. La edición número 100 del Electronic Gaming Monthly tuvo al Tetris en el primer lugar como el "mejor juego de todos los tiempos". En 2007, Tetris ocupó el segundo lugar en los "100 mejores videojuegos de todos los tiempos" para la el sitio web IGN. Ha vendido más de 70 millones de copias. En enero de 2010, se anunció que el Tetris ha vendido más de 100 millones de copias para teléfonos móviles sólo desde el año 2005.

En la actualidad existe la compañía llamada “The Tetris Company, LLC”, con base en Hawaii y propiedad de Henk Rogers and Alexey Pajitnov, que tiene registrada la patente del juego[23]. Esta patente consiste en por un lado derechos sobre el nombre “Tetris” y por el otro las mecánicas del juego o “gameplay”.

23 http://www.faqs.org/patents/app/20100144424

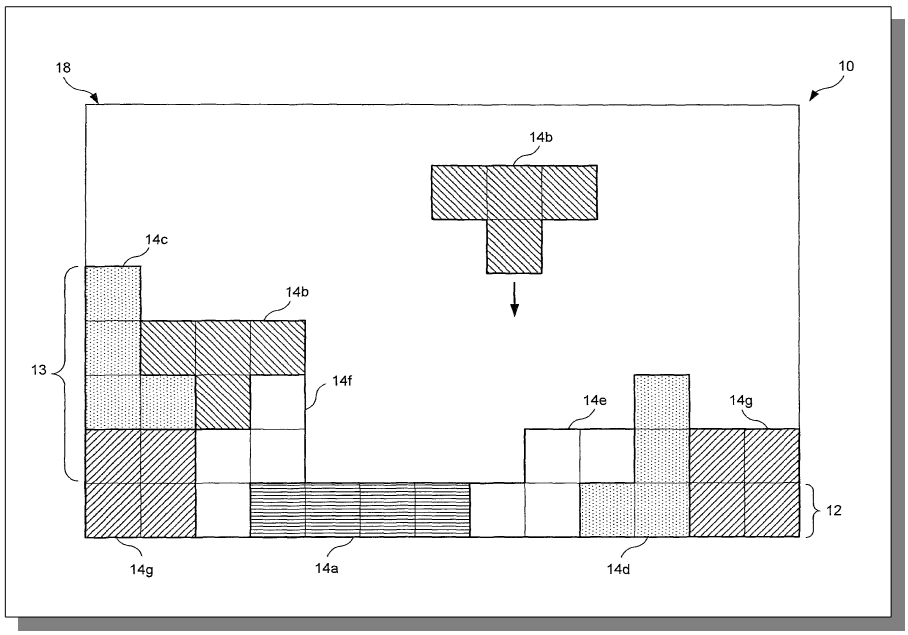


Imagen extraída del texto original de la patente[24] donde se observan las diferentes piezas.

En los últimos años han perseguido aquellas compañías que comercializaban copias que no contaban con su permiso, entre ellas se encuentran Apple que en 2008 tuvo que retirar de su App Store el juego "Tris"[25], en 2010 mantuvieron conversaciones con Google sobre varios clones que se alojaban en su tienda online, a lo que Google respondió retirando los 35 juegos a los que acusaban de infringir su patente[26]. En 2011[27] y 2012[28] la compañía a seguido demandando por estas mismas razones.

Como curiosidad, en 2008 en el MIT[29] se estableció que el Tetris pertenece a la "familia" de problemas NP-completos. Esto significa que no se conoce una manera "eficiente" (polinomial) de calcular la posición de las piezas que nos daría la puntuación máxima. Ni aun conociendo qué piezas y en qué orden nos las van a dar. La única solución es calcular todas las posibilidades.

2.4 Plataformas

Los videojuegos del tipo plataformas, o simplemente plataformas, son todos aquellos juegos que comparten unas características comunes que consisten principalmente en; caminar, correr, saltar o escalar sobre una serie de plataformas, evitar caer en precipicios, esquivar o matar enemigos y recoger objetos para poder completar los diferentes niveles del juego, además suelen usar vistas con desplazamiento horizontal y vertical para seguir los avances del jugador.

Es un género que desde sus comienzos en la década de 1980 ha sido muy popular y que hoy en día sigue teniendo numerosos seguidores. Nace en las máquinas recreativas con clásicos como Donkey Kong o Pitfall!, se consolida con Super Mario Bros y la consola NES de Nintendo en 1985.

24 http://ipforests.com/patents/video-game-systems-and-methods-for-providing-software-based-skill-adjustment-mechanisms-for-video-game-systems--2
25 http://kotaku.com/5041686/iphone-tetris-clone-tris-pulled-from-app-store
26 http://es.scribd.com/doc/32052396/Tetris-LLC-pulled-out-Clones-from-Android-Market
27 http://www.knowyourmobile.com/games/12031/tetris-company-forces-tetrada-out-wp7-marketplace
28 http://www.wired.co.uk/news/archive/2012-06/20/tetris-clone-ruling
29 http://arxiv.org/PS_cache/cs/pdf/0210/0210020v1.pdf

Durante el reinado de la NES en la segunda mitad de los ochenta el género de las plataformas en 2D vivió sus mejores momentos. Por una parte la popularización de la fórmula del Mario en las máquinas arcade trajo grandísimos títulos como New Zealand Story o Bubble Bobble, de Taito, o Adventure Island de Hudson. Por otro lado las posibilidades de una consola casera permitieron la aparición de juegos mucho más largos, dando origen a los plataformas de exploración como Metroid (1986) o Castlevania III (1989). Este subgénero tenía generalmente una historia y una ambientación más desarrollada, y un estilo de juego más orientado a exploración de los escenarios que al juego lineal y rápido de Super Mario Bros o los títulos provenientes de máquina arcade. Títulos como Ghosts'n Goblins, Mega Man, Contra (conocido en Europa como Probotector) o más tarde Prince of Persia (1989) fueron aportando frescura y popularidad al género durante la segunda mitad de los 80.

En la primera mitad de los 90 el género se estandarizó por completo tanto en las consolas de 16 bits (SNES y Sega Mega Drive) como en las consolas portátiles (Game Boy y Sega Game Gear). En los 16 bits salieron grandes títulos como Sonic the Hedgehog en la Mega Drive o Super Mario World en la SNES. Entre la marea de juegos de plataformas que se lanzaron en esta época, destacaban de vez en cuando títulos con un buen diseño como Earthworm Jim, Super Metroid o Another World. La última gran revolución en las plataformas en los 16 bits vino con Donkey Kong Country de Rare, que utilizando un avanzado sistema de pre-renderización de los gráficos, explotó por completo las capacidades de la SNES.

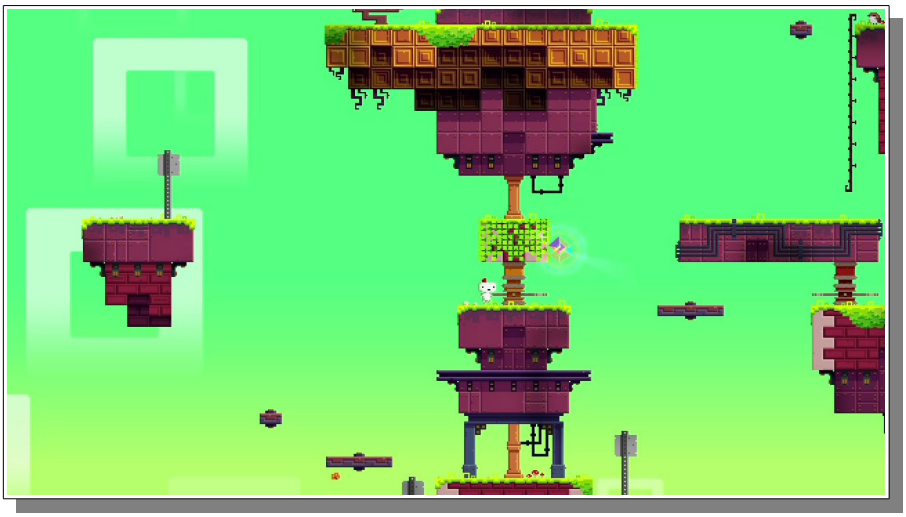
Izquierda "Super Metroid", derecha "Donkey Kong Country", los dos para Super Nintendo.

Con la aparición de las 3D y la nueva generación de consolas se experimentó la conversión del género a la tercera dimensión. Los primeros títulos como Clockwork Knight para Sega Saturn

presentaban gráficos tridimensionales pero un esquema de juego lineal, similar a los plataformas de toda la vida. Este tipo de juegos se consideran "2.5D". También se debe mencionar puntualmente el Jumping Flash, un juego para PlayStation que adaptaba los elementos de los juegos de plataformas con la jugabilidad en primera persona al estilo Doom.

No fue hasta la aparición de Super Mario 64 cuando se consiguió la fórmula correcta para adaptar el género a las 3D. Super Mario 64 introdujo escenarios completamente tridimensionales donde la libertad y la cantidad de movimientos en el escenario por fin consiguieron trasladar la jugabilidad, característica esencial de los plataformas, a la tercera dimensión. La genial fórmula de Super Mario 64, uno de los mejores y más influyentes títulos de la historia, tocó techo nada más estrenar el estilo en las 3D, y ningún título aportó nada excesivamente importante al género durante los años siguientes.

A comienzos de ésta última década se produce un auge en los plataformas 2D, llegando a convertir a los plataformas 3D en un subgénero minoritario. Muchas franquicias que habían dado el salto a los 3D vuelven a las 2 dimensiones debido entre otras cosas a, menores costes de desarrollo, la posibilidad de ofrecer un apartado artístico más elaborado y los problemas de cámara que sufren muchos plataformas 3D y que no se conseguían arreglar. Ejemplos de juegos que volvieron a las 2 dimensiones son The Impossible Game, Rayman Origins, Donkey Kong Country Returns y Sonic Generations. Sin embargo, y sobre todo en los cada vez más populares videojuegos independientes, también se crearon plataformas 2D completamente nuevos, como Super Meat Boy y Fez.



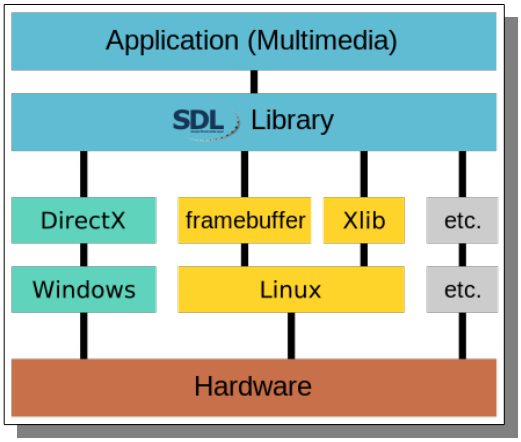
“*Fez” por el grupo de desarrollo independiente Polytron Corporation.*

3. SDL

Creada por Sam Latinga y su grupo, programadores de Loki Entertaiment Software, para portar juegos a Linux principalmente y lanzada en 1998, la Simple DirectMedia Layer, o SDL, es un conjunto de bibliotecas multimedia multi-plataforma que proporcionan funciones básicas para realizar operaciones de dibujo en dos dimensiones mediante framebuffer, en tres dimensiones a través de OpenGL, gestión de efectos de sonido y música, además de carga y gestión de imágenes. La biblioteca se distribuye bajo la licencia LGPL , es una librería muy utilizada en diferentes tipos de aplicaciones entre ellas se encuentra software para reproducir video MPG, emuladores y un gran número juegos y adaptaciones de juegos entre plataformas[30].

Juegos que utilizan SDL, izquierda "OpenTTD" derecha "Civilization: Call To Power".

Las plataformas soportadas por la versión usada en el proyecto (SDL 1.2.15) son Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX y QNX. El código contiene además soporte para AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS y OS/2, pero no son oficiales.



Capas de abstracción para diferentes plataformas.

30 http://en.wikipedia.org/wiki/List_of_games_using_SDL

SDL está escrita en C y funciona en C++ de manera nativa, además se puede usar con un gran número de lenguajes diferentes incluyendo Ada, C#, D, Eiffel, Erlang, Euphoria, Go, Guile, Haskell, Java, Lisp, Lua, ML, Oberon/Component Pascal, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, Smalltalk y Tcl.

La SDL está dividida en varios subsistemas; video, audio, manejo de eventos, CD-ROM, joystick, timer y threads. Además se han desarrollado una serie de bibliotecas adicionales que complementan las funcionalidades y capacidades de la biblioteca base, se encuentran en la página oficial de la biblioteca y su documentación se presenta junto a la oficial.

- SDL_image : soporta múltiples formatos de imagen.
- SDL_mixer : facilita el trabajo con el audio.
- SDL_net : soporte para trabajo en red.
- SDL_ttf : para el pintado de fuentes TrueType[31].
- SDL_rtf : para el pintado de formato RichText[32].

Durante el desarrollo del proyecto existían tres versiones de la biblioteca, la 1.2 que en ese momento era la oficial y estable, la 1.3 cuya principal novedad era la inclusión de IO´s y Android como plataformas soportadas y la 2.0 que se encontraba en desarrollo y se espera que se termine durante el 2013, substituta oficial de la 1.2. Se escogió la versión 1.2 por ser la recomendad en la página oficial ya que la 1.3 aunque tuviese novedades interesantes no estaba completamente terminada. Las últimas noticias sobre la versión 2.0 es que se encuentra en estado Release Candidate[33] .

La 2.0 es una actualización importante, entre lo más destacado se encuentra; inclusión de IO ´s y Android como plataformas soportadas, API force-feedback para joysticks, soporte para capturar audio, multi-threading mejorado, gráficos 2D acelerados por hardware, soporte multi-ventana, mejorado el soporte para pantalla completa, etc... .Además esta distribuida con licencia zlib a diferencia de la 1.2, lo que implica que se podrá incluir en proyectos comerciales de código cerrado.

3.1 Trabajando con SDL

Inicializando

Antes de poder usar cualquiera de los subsistemas antes mencionados deben ser inicializados. Para ello se tienen las funciones *SDL_Init* o *SDL_InitSubSystem*. *SDL_Init* debe ser llamado antes que cualquier otra función de la SDL. Automáticamente inicializa los subsistemas de eventos, I/O archivos y threads, y por el parámetro podemos especificar que otros subsistemas queremos iniciar. Los posibles parámetros de estas funciones son:

- *SDL_INIT_VIDEO*
- *SDL_INIT_AUDIO*
- *SDL_INIT_TIMER*
- *SDL_INIT_CDROM*

31 http://en.wikipedia.org/wiki/TrueType
32 http://en.wikipedia.org/wiki/Rich_Text_Format
33 http://www.phoronix.com/scan.php?page=news_item&px=MTM4MzU

- *SDL_INIT_JOYSTICK*
- *SDL_INIT_EVERYTHING*

Si lo queremos activar todo, pasaremos como único parámetro *SDL_INIT_EVERYTHING*. Una vez inicializado SDL, si necesitamos inicializar otro subsistema podemos hacerlo con la función *SDL_InitSubSystem.* En el siguiente ejemplo se inicializa la SDL y los subsistemas de video y audio además de los automáticos.

```
#include <SDL/SDL.h>     // Biblioteca principal
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

    cout << "Inicializando SDL." << endl;

    // Inicializar subsistemas automáticos, Video y Audio
    if((SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO)==-1)) {

        cerr << "No se pudo inicializar SDL: " << SDL_GetError() << endl;
        return(-1);
    }

    cout << "SDL inicializada." << endl;
.
.
.
    cout << "Cerrando SDL." << endl;
    // Cerrando todos los subsistemas
    SDL_Quit();
    return(0);
}
```

La función *SDL_GetError* devuelve el último error interno de SDL en formato cadena. Cuando una aplicación SDL es inicializada, se crean dos archivos, *stderr.txt* y *stdout.txt.* Durante la ejecución del programa, cualquier información que se escriba en la salida de error estándar se escribe en el archivo *stderr.txt*. Igualmente, cualquier información que se escriba en la salida estándar se guardará en el archivo *stdout.txt*. Una vez finalizada la ejecución, si estos archivos están vacíos son borrados automáticamente, si no, estos permanecen intactos.

Este comportamiento se puede cambiar por el normal, y de hecho se cambió para el desarrollo del proyecto. Existen dos formas de cambiarlo, la fácil sería añadir una serie de líneas después de la llamada a *SDL_Init* y la difícil consistiría en recompilar el código fuente de la SDL con una configuración diferente para conseguir un *libSDLmain.a* nuevo a substituir por el que viene con el código de la página oficial. Las instrucciones para esto se encuentran en la documentación oficial de la SDL[34].

Al igual que inicializamos SDL, cuando hemos terminado el trabajo hemos de cerrarla. La función encargada de esta tarea es *SDL_Quit*. En el caso de que queramos finalizar sólo un

34 http://www.libsdl.org/cgi/docwiki.fcg/FrontPage

subsistema de SDL usaremos la función *SDL_QuitSubSystem*.

3.1.1 Subsistemas

Video

Para poder utilizar el subsistema de vídeo debemos inicializarlo. Para ello, como ya se ha visto, debemos pasar como parámetro a la función *SDL_Init*, la correspondiente macro del subsistema, en este caso *SDL_INIT_VIDEO*. Una vez inicializado debemos establecer el modo de video mediante la función *SDL_SetVideoMode*, la cual establece las propiedades de la superficie principal de nuestra aplicación.

```
SDL_Surface* SDL_SetVideoMode(int width, int height, int bitsperpixel, Uint32 flags);
```

width [in]

- El ancho deseado para el modo de video en pixeles.

height [in]

- El alto deseado para el modo de video en pixeles.

bitsperpixel [in]

- El número de bits por pixel para el modo de video. Siendo los más comunes 24 para 3 bytes por pixel y 32 para 4 bytes por pixel.

flags [in]

- Posibles valores para los flags que se pueden combinar. Son los mismos que usa *SDL_Surface*.

SDL_SWSURFACE	Crea la superficie de vídeo en la memoria del sistema
SDL_HWSURFACE	Crea la superficie de vídeo en la memoria de video
SDL_ASYNCBLIT	Activa el uso de actualizaciones asíncronas de la superficie. Esto puede bajar el rendimiento en maquinas con un sólo CPU, pero mejorarlo en sistemas SMP.
SDL_ANYFORMAT	Si los *bitsperpixel* solicitados no están disponibles, SDL los emulará mediante una superficie sombra. Pasando esta flag lo evitamos.
SDL_HWPALETTE	Da a SDL acceso exclusivo a la paleta, sin esta flag no existe la certeza de que siempre se consiga el color que se desea al usar *SDL_SetColors* o *SDL_SetPalette*.
SDL_DOUBLEBUF	Activa el uso de doble buffer por hardware, sólo válido con *SDL_HWSURFACE*. Llamando a *SDL_Flip* se intercambiaran los buffers y

	actualizará la pantalla. El pintado se realiza en la superficie que no se está mostrando en ese momento.
SDL_FULLSCREEN	La SDL intentará usar el modo pantalla completa. Si no es posible un cambio de resolución hardware (por la razón que sea), se usará la siguiente resolución hacia arriba con el pintado centrado sobre un fondo negro.
SDL_OPENGL	Crea un contexto de pintado para OpenGL.
SDL_OPENGLBLIT	Esta opción se mantiene por compatibilidad y será eliminada en la siguiente versión, no es recomendada para código nuevo.
SDL_RESIZABLE	Crea una ventana a la que se le puede cambiar el tamaño.
SDL_NOFRAME	Si es posible, causa que la SDL cree una ventana sin título ni decoración de ningún tipo. El modo pantalla completa tiene esta flag activada automáticamente.

```
// Inicializar SDL
SDL_Init(SDL_INIT_VIDEO);

// Iniciar modo video
SDL_Surface* myVideoSurface=SDL_SetVideoMode(640,480,24,SDL_DOUBLEBUF|SDL_FULLSCREEN);

// Mostrar información sobre le modo de video
if (myVideoSurface != NULL)
   cout << "Bits per pixel en el modo de video actual: " << (int)myVideoSurface-
>format->BitsPerPixel << endl;

else
   cerr << "Error iniciando modo de video: " << SDL_GetError() << endl;

// Cerrar SDL y subsistemas
SDL_Quit();
```

Le hemos pedido a SDL que establezca un modo de video con una resolución de 640x480 píxeles y 24 bits de color (true color). Además le solicitamos que use una estrategia de doble buffer para el repintado y el modo pantalla completa.

Si se usa doble buffer, la siguiente función los intercambia, es decir, vuelca el contenido del buffer secundario al principal. Esta función se usa cuando la nueva escena esta completamente dibujada en el buffer secundario. El parámetro debe ser la superficie principal, si tiene éxito devuelve 0, si no, devuelve –1. De esta manera evitamos que la imagen de la sensación de parpadeo.

```
int SDL_Flip(SDL_Surface* screen);
```

En SDL las “surface” o superficies son el elemento básico que se utiliza para construir los gráficos, representa un área de memoria gráfica sobre la que se puede dibujar. Su estructura es la siguiente:

```
typedef struct SDL_Surface {
    Uint32 flags;                           /* Sólo lectura */
    SDL_PixelFormat *format;                /* Sólo lectura */
    int w, h;                               /* Sólo lectura */
    Uint16 pitch;                           /* Sólo lectura */
    void *pixels;                           /* Lectura-Escritura */
    SDL_Rect clip_rect;                     /* Sólo lectura */
    int refcount;                           /* Lectura principalmente */

  /* Esta estructura posee atributos privados que se no se muestran aquí */
} SDL_Surface;
```

La superficie que nos devuelve SDL_SetVideoMode es la única visible. Sin embargo necesitaremos crear y utilizar superficies intermedias para almacenar gráficos, construir escenas, etc... Estas superficies no pueden ser directamente mostradas por pantalla, pero sí copiadas en la superficie principal (visible), a este proceso lo llamamos bit blitting y se realiza mediante la función *SDL_BlitSurface*. Esta función copia zonas rectangulares de una superficie a otra.

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect
*dstrect);
```

Si observamos los parámetros, hay dos de tipo *SDL_Surface* y otros dos del tipo *SDL_Rect*. El primer tipo es el que acabamos de comentar, en concreto son los parámetros *src* y *dst*, que son la superficie fuente y la de destino respectivamente. El parámetro *srcrect* define la zona o porción rectangular de la superficie desde la que queremos copiar. *Dstrect* es la zona rectangular de la superficie de destino en la que vamos a copiar el gráfico. El tipo *SDL_Rect* tiene la siguiente forma:

```
typedef struct{
   Sint16 x, y;
   Uint16 w, h;
} SDL_Rect;
```

Existen varias formas de crear superficies nuevas, con la función *SDL_CreateRGBSurface* creamos una superficie vacía y lista para usar.

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height,
            int bitsPerPixel, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

Esta función al igual que hacía la función *SDL_SetVideoMode*, nos devuelve un puntero a una superficie. Los parámetros *width* y *height* tienen el mismo significado que en *SDL_SetVideoMode*, y *depth* es lo mismo que *bpp*. Los posibles valores para *flags* son:

- *SDL_SWSURFACE* : Crea la superficie de vídeo en la memoria principal, mejora el rendimiento en el trabajo a nivel de pixeles, sin embargo no se podrán aprovechar algunos

tipos de hardware bliting.

- *SDL_HWSURFACE* : La SDL intentará crear la superficie en la memoria de vídeo.
- *SDL_SRCCOLORKEY* : Permite el uso de transparencias (color key).
- *SDL_SRCALPHA* : Activa el alpha-blending.

Otra función mediante la cual se crean superficies es *SDL_LoadBMP*, la cual tiene como parámetro la dirección de un fichero gráfico con formato *.bmp* (bitmap). Si la función tiene éxito nos devuelve una superficie del tamaño del gráfico que acabamos de cargar, además hay que tener en cuenta que cuando carga un archivo Windows BMP de 24 bit, la información de los pixels no es rojo, verde, azul (RGB) sino azul, verde, rojo.

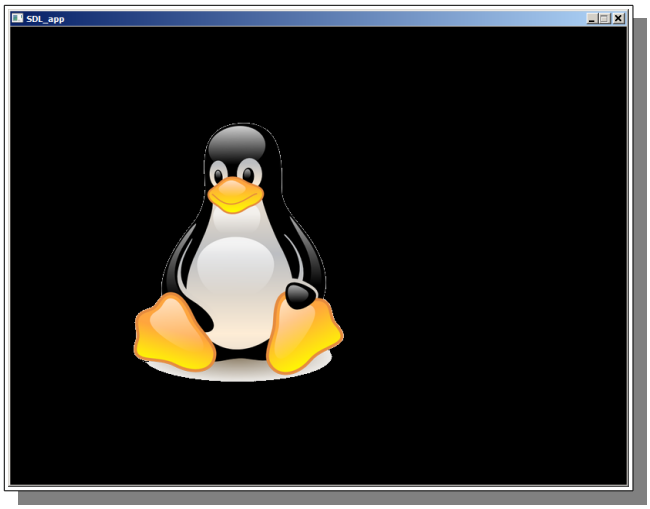
```
SDL_Surface *SDL_LoadBMP(const char *file);
```

En el siguiente ejemplo se ponen en práctica las funciones comentadas y se introducen nuevas. El código a continuación comienza declarando las variables que se usaran, inicializa la SDL y comprueba el modo de video antes de establecerlo. Se utiliza la función *atexit* para que se cierre la SDL al terminar el programa. Se carga una imagen y se establece su "color key", el cual es un color que en el momento de hacer blit se eliminará. Se copia la imagen sobre la superficie principal y se realiza el intercambio de buffers. Una vez se ha terminado con el uso de una superficie se debe liberar usando la función *SDL_FreeSurface*. El código acaba con un bucle de espera que finaliza cuando se detecta que se ha cerrado la ventana.

```
#include <stdio.h>
#include <SDL/SDL.h>

#include <iostream>
using namespace std;

// Vamos a cargar una imagen en una superficie
// y vamos a mostrarla por pantalla

int main(int argc, char *argv[]) {

// Declaramos los punteros para la pantalla y la imagen a cargar
	SDL_Surface *pantalla, *imagen;
	SDL_Event evento;

	// Variable auxiliar dónde almacenaremos la posición dónde colocaremos
	// la imagen dentro de la superficie principal
	SDL_Rect destino;

	// Iniciamos el subsistema de video de SDL
	if(SDL_Init(SDL_INIT_VIDEO) < 0) {

		cerr << "No se pudo iniciar SDL: " << SDL_GetError();
		exit(1);
	}
	atexit(SDL_Quit);

	// Comprobamos que sea compatible el modo de video
	if(SDL_VideoModeOK(800, 600, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
```

```
		cerr << "Modo no soportado: " << SDL_GetError();
		exit(1);
	}

	// Establecemos el modo de video y asignamos la superficie
	// principal a la variable pantalla
	pantalla = SDL_SetVideoMode(800, 600, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);

	if(pantalla == NULL) {

		cerr <<"No se puede inicializar el modo gráfico: " << SDL_GetError();
		exit(1);
	}

	// Cargamos un bmp en una nueva superficie para realizar las pruebas
	imagen = SDL_LoadBMP("../Imagenes/software_libre.bmp");

	if(imagen == NULL) {

		cerr <<"No se puede cargar la imagen: " << SDL_GetError();
		exit(1);
	}

	//Establecemos el color key
	if(SDL_SetColorKey(imagen, SDL_SRCCOLORKEY|SDL_RLEACCEL,SDL_MapRGB(imagen-
>format, 0, 255, 0)) == -1){

		cerr << "No se pudo establecer el ColorKey: " << SDL_GetError();
	}

	// Inicializamos la variable de posición y tamaño de destino
	destino.x = 150; // Posición horizontal con respecto a la esquina sup right
	destino.y = 120; // Posición vertical con respecto a la esq sup right
	destino.w = imagen->w;  // Longitud del ancho de la imagen
	destino.h = imagen->h;  // Longitud del alto de la imagen

	// Copiamos la superficie creada rectángulo en la anterior pantalla
	SDL_BlitSurface(imagen, NULL, pantalla, &destino);

	// Mostramos la pantalla, intercambio de buffers
	SDL_Flip(pantalla);

	// Ya podemos liberar el rectángulo, una vez copiado y mostrado
	SDL_FreeSurface(imagen);

	// Ahora mantenemos el resultado en pantalla
	// hasta cerrar la ventana
	for(;;) {

		// Consultamos los eventos

		while(SDL_PollEvent(&evento)) {

			if(evento.type == SDL_QUIT) // Si es de salida
				return 0;
```

```
            }
        }
        return 0;
}
```



A la izquierda la imagen utilizada en el ejemplo. Derecha, captura del ejemplo en marcha.

Existen además un grupo de funciones relacionadas con la gestión de la ventana, con las que podemos cambiar el nombre o el icono de la aplicación, cambiar de modo ventana a pantalla completa o recoger información especifica sobre el gestor de ventanas entre otras.

```
//Cambiar a pantalla completa
int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
//Establecer nombre de la ventana e icono
void SDL_WM_SetCaption(const char *title, const char *icon);
//Recoger información del window-manager
int SDL_GetWMInfo(SDL_SysWMinfo *info);
```

Eventos

La gestión de eventos de SDL es realmente cómoda y fácil de usar. El subsistema de eventos de SDL se inicializa junto al subsistema de video, por lo que no hay que inicializarlo expresamente. Internamente, la SDL almacena todos los eventos en espera a ser procesados en un cola. Usando las funciones *SDL_PollEvent*, *SDL_PeepEvents* y *SDL_WaitEvent* se pueden observar y procesar estos eventos en espera.

La clave de la gestión de eventos en SDL es la unión SDL_Event, que contiene en su estructura una unión por cada tipo de evento. Según la documentación oficial es probable que sea la segunda estructura más importante en SDL después de *SDL_Surface*.

```
typedef union{
  Uint8 type;
  SDL_ActiveEvent active;
  SDL_KeyboardEvent key;
  SDL_MouseMotionEvent motion;
  SDL_MouseButtonEvent button;
```

```
  SDL_JoyAxisEvent jaxis;
  SDL_JoyBallEvent jball;
  SDL_JoyHatEvent jhat;
  SDL_JoyButtonEvent jbutton;
  SDL_ResizeEvent resize;
  SDL_ExposeEvent expose;
  SDL_QuitEvent quit;
  SDL_UserEvent user;
  SDL_SysWMEvent syswm;
} SDL_Event;
```

Como se puede observar existen varios tipos de eventos en SDL; teclado, ratón, joystick, y otros relacionados con el sistema operativo.

- type: Indica el tipo de evento que se ha producido. Existen varios tipos de eventos, desde los producidos por una entrada directa del usuario por teclado hasta los eventos del sistema. Como por ejemplo *SDL_Quit* que se uso en el anterior ejemplo para detectar que la aplicación de había cerrado.

type del evento	Estructura del evento
SDL_ACTIVEEVENT	SDL_ActiveEvent
SDL_KEYDOWN/UP	SDL_KeyboardEvent
SDL_MOUSEMOTION	SDL_MouseMotionEvent
SDL_MOUSEBUTTONDOWN/UP	SDL_MouseButtonEvent
SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent
SDL_JOYBUTTONDOWN/UP	SDL_JoyButtonEvent
SDL_VIDEORESIZE	SDL_ResizeEvent
SDL_VIDEOEXPOSE	SDL_ExposeEvent
SDL_QUIT	SDL_QuitEvent
SDL_USEREVENT	SDL_UserEvent
SDL_SYSWMEVENT	SDL_SysWMEvent

- active: Evento de activación. Este evento informa acerca de la situación del foco sobre nuestra ventana, es considerado un evento de sistema.
- key: Es un evento producido por una acción en el teclado.
- motion: Evento producido por el movimiento del ratón.
- button: Evento producido por una acción sobre el botón del ratón.
- jaxis: Evento de movimiento del eje del joystick.
- jball: Evento de movimiento del trackball del joystick.
- jhat: Evento producido por el movimiento del minijoystick o hat del dispositivo de juego.

- jbutton: Evento activado por una acción en algún botón del joystick.
- resize: Evento provocado por el redimensionado de la ventana que contiene la aplicación.
- quit: Evento producido al cerrar la aplicación.
- user: Evento definido por el usuario.
- syswm: Evento indefinido producido por el gestor de ventanas.

Para este proyecto no se ha usado control sobre el ratón ni joystick´s, solamente el teclado, por lo que se estudiará este tipo en particular. Cada vez que se produce un evento de teclado (pulsar o soltar una tecla) este se almacena en la estructura *SDL_KeyboardEvent* que tiene la siguiente forma:

```
typedef struct{
  Uint8 type;
  Uint8 state;
  SDL_keysym keysym;
} SDL_KeyboardEvent;

typedef struct{
  Uint8 scancode;
  SDLKey sym;
  SDLMod mod;
  Uint16 unicode;
} SDL_keysym;

// Ejemplo de uso
...
while(SDL_PollEvent(&event)){

  switch(event.type){

    case SDL_KEYDOWN:

      if(event.key.keysym.sym==SDLK_LEFT) move_left();
      break;
    ...
  }
}
...
```

El *type* y *state* de *SDL_KeyboardEvent* guardan la misma información, la diferencia está en que usan diferentes valores. Los eventos de teclado se generan cuando se presiona una tecla (*type* = *SDL_KEYDOWN* o *state* = *SDL_PRESSED*) o se suelta (*type* = *SDL_KEYUP* o *state* = *SDL_RELEASED*). La información sobre que tecla se ha pulsado se encuentra en *keysym*, concretamente en *sym* del tipo *SDLKey*, como se puede observar en el ejemplo. *SDLKey* es un enumerado que contiene todas los tipos de teclas controladas por SDL, empezando todas por *SDLK_* y luego el nombre de la tecla.

SDLKey	**Valor ASCII**
SDLK_BACKSPACE	'\b'

SDLK_TAB	'\t'
SDLK_CLEAR	
SDLK_RETURN	'\r'
SDLK_PAUSE	
SDLK_ESCAPE	'^['
SDLK_SPACE	' '
…	...

Para la captura de eventos SDL nos ofrece tres opciones diferentes, podemos usar *SDL_PollEvent* que extrae el primer evento de la cola si hay alguno, *SDL_PeepEvents* que chequea la cola en busca de eventos y de forma opcional (configurable a través de los parámetros) los extrae y por último *SDL_WaitEvent* el cual espera indefinidamente hasta el próximo evento disponible.

```
int SDL_PollEvent(SDL_Event *event);

int SDL_PeepEvents(SDL_Event *events, int numevents, SDL_eventaction action, Uint32
mask);

int SDL_WaitEvent(SDL_Event *event);
```

Para finalizar con este apartado, dos funciones a tener en cuenta. Por un lado *SDL_EnableKeyRepeat* nos permite controlar la repetición de teclas cuando mantenemos una apretada, pudiendo establecer el tiempo que se ha de mantener hasta que empiece la repetición o deshabilitar esta opción. Y por otro *SDL_PumpEvents*, que se utiliza para forzar la actualización de la cola de eventos. Su uso es sólo necesario cuando se necesita tener acceso a la cola pero no se va a llamar a ninguna de las funciones antes mencionadas para la gestión de eventos, ya que éstas lo hacen de forma implícita, por ejemplo para un filtrado previo. Solamente se puede llamar a esta última función desde el thread que estableció el modo de video.

Sonido

El subsistema de audio de SDL, en comparación con el resto de los subsistemas, es uno de los aspectos menos versátiles de los que proporciona esta biblioteca. Comparado con otras partes de SDL, el subsistema de audio es de muy bajo nivel debido a que debe de mantener la compatibilidad con todas las plataformas que soporta.

Afortunadamente, existe la biblioteca oficial llamada *SDL_mixer* que facilita el trabajo con el sonido y está más en línea con el estilo de la SDL. Para usar esta biblioteca tendremos que inicializar el subsistema de audio en la inicialización de la SDL, pero esto se verá en el apartado de

"Bibliotecas adicionales".

Timming

En este apartado se tratará el control de tiempo. Dejamos sin ver todo lo referido a multitarea y programación de hilos, porque no se ha usado para el proyecto. El control del tiempo es una tarea esencial, ya que no todos los ordenadores funcionan a la misma velocidad. Si no lo controlásemos, nuestro juego funcionaría bien en algunos ordenadores, pero en otros (los más rápidos) iría a gran velocidad.

Para llevar a cabo esta tarea la SDL nos proporciona dos funciones. La primera devuelve el número de milisegundos transcurridos desde que se inicializó SDL, Gracias a ella podemos controlar el tiempo transcurrido entre dos instantes dados dentro del programa.

```
void SDL_Delay(Uint32 ms);
```

La siguiente función nos permite detener la ejecución del juego durante un tiempo determinado. Como parámetro pasamos el número de milisegundos que queremos esperar, ésta como mínimo esperará el tiempo indicado, pero probablemente más debido a la planificación del sistema operativo. La precisión es de 10 ms, algo común.

```
void SDL_Delay(Uint32 ms);
```

Mencionar también que la SDL proporciona dos funciones para el trabajo con temporizadores o timers, una para crear uno nuevo y otra para eliminar uno ya creado. Funcionan como cabría esperar que funcionasen, cuando se crea uno nuevo se le pasa como parámetros al constructor el intervalo de tiempo que ha de esperar para ejecutarse, la función a ejecutar y los parámetros que necesite la función a ejecutar.

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback callback, void* param);
SDL_bool SDL_RemoveTimer(SDL_TimerID id);
```

3.1.2 SDL y OpenGL

OpenGL es una biblioteca gráfica disponible para un gran número de plataformas. Sin embargo requiere código especifico para conseguir que funcione en cada una de ellas, por lo que no suele ser una solución ideal para aplicaciones multiplataforma. SDL soluciona este problema, permite crear programas que usen OpenGL sin perder las propiedades multiplataforma, aprovechando los demás subsistemas de SDL, audio, gestión de eventos, threads y timming, dejando el apartado gráfico en manos de la SDL.

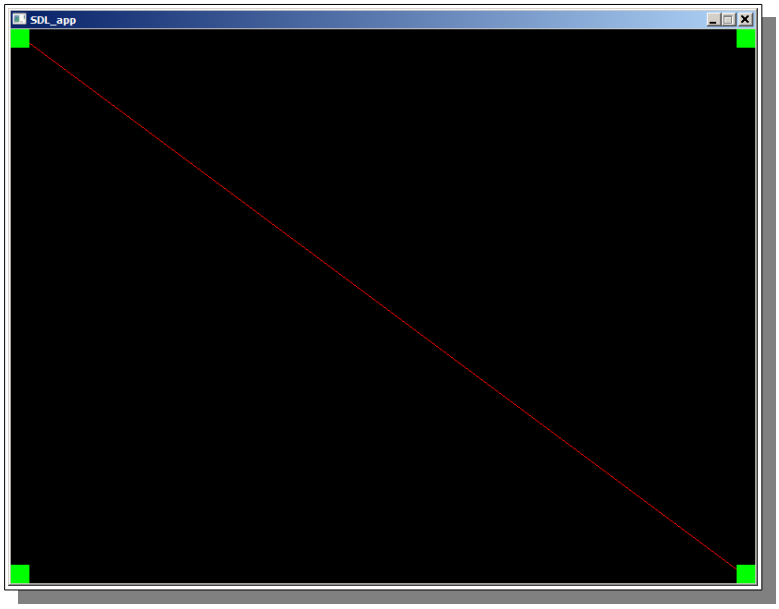
El uso de OpenGL dentro de SDL es el mismo que con cualquier otra API, por ejemplo GLUT. Se utilizan las mismas llamadas a funciones y tipos de datos. La diferencia está en la forma de inicializar OpenGL. No se entrará en detalles sobre OpenGL porque no se llegó a utilizar en el proyecto, aunque durante el estudio y prueba de SDL no se descartó, una de esas pruebas es el código siguiente en el que se trata la antes comentada inicialización.

```
#include <stdio.h>
```

```
#include <SDL/SDL.h>

//Bibliotecas Opengl
#include <gl/gl.h>
#include <gl/glu.h>

#include <iostream>
using namespace std;

//Inicializa SDL y OpenGL
//Pinta una linea y unos cuadrados con OpenGL

void pintarPrueba(void);
void inicializarOpenGL(void);

int main(int argc, char *argv[]) {

	SDL_Surface *pantalla;
	SDL_Event evento;

	// Iniciamos el subsistema de video de SDL
	if(SDL_Init(SDL_INIT_VIDEO) < 0) {

		cerr << "No se pudo iniciar SDL: " << SDL_GetError();
		exit(1);
	}
	atexit(SDL_Quit);

	//dejamos el uso de double buffer a OpenGL
	SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

   // Comprobamos que el modo de video sea compatible con la flag especifica de OpenGL
	if(SDL_VideoModeOK(800, 600, 24, SDL_OPENGL) == 0) {

		cerr << "Modo no soportado: " << SDL_GetError();
		exit(1);
	}

	// Establecemos el modo de video y asignamos la superficie principal
	pantalla = SDL_SetVideoMode(800, 600, 24, SDL_OPENGL);

	if(pantalla == NULL) {

		cerr <<"No se puede inicializar el modo gráfico: " << SDL_GetError();
		exit(1);
	}
	//inicialización de OpenGL
	inicializarOpenGL();

	for(;;) {

		// Consultamos los eventos
		while(SDL_PollEvent(&evento)) {

			if(evento.type == SDL_QUIT) // Salida
				return 0;
```

```
            }

            //Pintamos algo de prueba
            pintarPrueba();

            //Un descanso
            SDL_Delay(15);
      }
      SDL_Quit( );
      return 0;
}

void inicializarOpenGL(void){

      //Establecemos el color que se usará para limpiar el buffer
      glClearColor(0, 0, 0, 0);

      //Establecemos el viewport
      glViewport(0, 0, 800, 600);

      //Inicializamos la matriz de proyección
      glMatrixMode(GL_PROJECTION);
      //reseteamos la matriz
      glLoadIdentity();

      //Establecemos una perspectiva 2D
      glOrtho(0, 800, 600, 0, -1, 1);

      //Inicializamos la matriz ModelView
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}

void pintarPrueba(void){

      //Limpiamos los buffers
      glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

      glMatrixMode( GL_MODELVIEW );
      glLoadIdentity();

      //cambia color a rojo
      glColor3f(1.0f, 0.0f, 0.0f);

      //Dibuja linea
      glBegin(GL_LINES);
            glVertex3f(0,0,0); // origen
            glVertex3f(800,600,0); // final
      glEnd( );

      //Cambia color y dibuja un cuadrado en cada esquina
      glColor3f(0.0f, 1.0f, 0.0f);
      glBegin(GL_QUADS);
          glVertex3f(0, 0, 0);
          glVertex3f(20, 0, 0);
          glVertex3f(20, 20, 0);
```

```
            glVertex3f(0, 20, 0);
        glEnd();

        glBegin(GL_QUADS);
                glVertex3f(780, 0, 0);
                glVertex3f(800, 0, 0);
                glVertex3f(800, 20, 0);
                glVertex3f(780, 20, 0);
        glEnd();

        glBegin(GL_QUADS);
                glVertex3f(780, 580, 0);
                glVertex3f(800, 580, 0);
                glVertex3f(800, 600, 0);
                glVertex3f(780, 600, 0);
        glEnd();

        glBegin(GL_QUADS);
                glVertex3f(0, 580, 0);
                glVertex3f(20, 580, 0);
                glVertex3f(20, 600, 0);
                glVertex3f(0, 600, 0);
        glEnd();

        //Hace las funciones de SDL_Flip(pantalla), intercambia los buffers
        SDL_GL_SwapBuffers();
}
```



Captura del ejemplo.

Lo más destacado del ejemplo es la manera de comunicar a la SDL que se usará OpenGL. Se inicializa el subsistema de vídeo y se establecen unos atributos especiales SDL/OpenGL con la función *SDL_GL_SetAttribute*, el más importante es el que se usa en el ejemplo. De esa forma le comunicamos a OpenGL que deseamos usar doble buffer, ya que la SDL no se encargará de esa tarea. Otro detalle importante se encuentra en la forma de establecer el modo de vídeo, se le debe pasar la flag *SDL_OPENGL*. Dentro del ejemplo la función *inicializarOpenGL* inicializa las

propiedades de la cámara y la perspectiva, acciones comunes en cualquier aplicación que use OpenGL. Por último comentar que para hacer el intercambio de buffers no se puede llamar a *SDL__Flip* como cuando la SDL controlaba esta tarea, ahora se deberá llamar a *SDL_GL_SwapBuffers* para realizar esta tarea, esto se puede observar en el código anterior al final de la función de pintado.

3.1.3 Bibliotecas adicionales

En este apartado se trataran las bibliotecas adicionales a SDL que amplían sus funcionalidades. Estas bibliotecas son oficiales, siendo Sam Latinga uno de los autores en todas ellas. Se puede encontrar una completa documentación de éstas junto con la oficial de SDL. La instalación de estas bibliotecas se deja para el apartado de "Preparación del entorno". Como se ha comentado antes existen cinco de éstas bibliotecas, aunque para el proyecto solamente se han usado dos de ellas y probado una tercera las cuales serán las que se comenten a continuación.

SDL_ttf

SDL_ttf es una biblioteca para el renderizado de fuentes TrueType[35] y depende de freetype2[36] para manejar los datos de las fuentes. Permite al programador usar múltiples fuentes TrueType sin tener que codificar una rutina de pintado. Gracias a las fuentes vectoriales, las outline fonts[37] y el uso de antialiasing[38] se puede conseguir texto de gran calidad sin mucho esfuerzo.

Esta biblioteca nos permite dibujar el texto que queramos en una superficie SDL utilizando el tipo de letra que deseemos. Para utilizar esta nueva librería, e igualmente que con otros aspectos de SDL, debemos de inicializarla y cerrarla una vez hayamos terminado de usarla. Las siguientes funciones realizan éstas tareas.

```
int TTF_Init()
void TTF_Quit()
int TTF_WasInit()
```

La inicialización debe ser la primera función de la biblioteca en usarse, exceptuando *TTF_WasInit* que comprueba si ya se ha inicializado.

Una vez inicializada se pueden crear y destruir fuentes con las que escribir texto sobre superficies. La estructura que mantiene la información de esta biblioteca auxiliar es la estructura *TTF_font*. Los detalles de implementación de esta estructura están totalmente ocultos y no son necesarios para trabajar con la biblioteca. Sólo necesitamos trabajar con punteros a este tipo de estructura pasándolo como parámetro en distintas funciones.

```
#include "SDL_ttf.h"

...
   font = TTF_OpenFont("../font/NewShape.ttf",14);
```

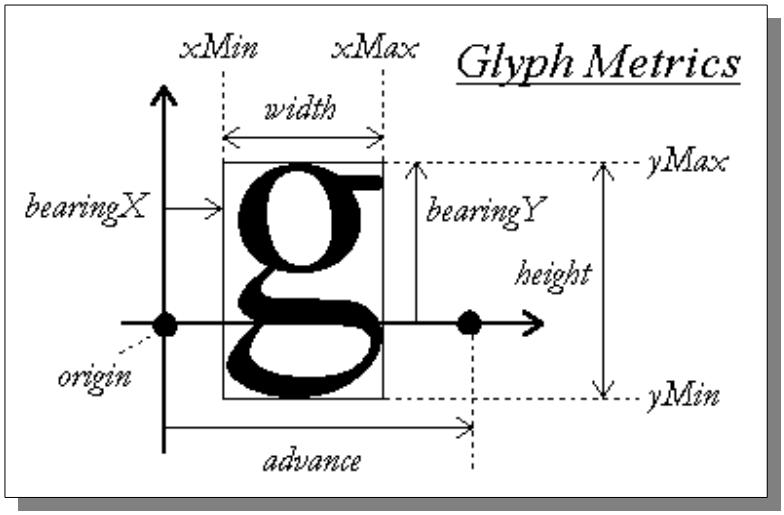
35 http://es.wikipedia.org/wiki/TrueType
36 http://www.cs.sunysb.edu/documentation/freetype-2.1.9/docs/ft2faq.html#general-what
37 http://en.wikipedia.org/wiki/Computer_font#Outline_fonts
38 http://es.wikipedia.org/wiki/Antialiasing

```
    if(font == NULL) {

        cerr << "No se pudo abrir la fuente con tamano 14: " << SDL_GetError() << endl;
        exit(-1);
    }
...
```

En el extracto de código anterior se puede ver como se crea una fuente, se pasa como parámetro la dirección de la fuente TrueType a cargar y el tamaño que deseamos. Para liberar las fuentes se pasa el puntero como parámetro a la función *TTF_CloseFont*.

```
void TTF_CloseFont(TTF_Font *font)
```

Existe también un grupo de funciones para tratar los atributos de las fuentes, mostrados en la siguiente imagen, esta parte no se ha estudiado por desviarse un poco del tema.



Atributos modificables.

Por último se repasará las funciones envueltas en el pintado de texto. SDL nos proporciona tres modos para esta tarea:

- *Solid* : "rápido y sucio", crea una superficie de 8-bit y renderiza el texto dado con el color especificado. Es el modo más rápido de los tres.
- *Shaded* : "lento y bueno, pero con caja solida", crea una superficie de 8-bit y renderiza el texto dado con gran calidad con el color especificado y una caja que rodea el texto cuyo color también hay que especificar. Se usa antialiasing para el pintado del texto.
- *Blended* : "lento, lento, lento, lento, pero altísima calidad", crea una superficie de 32-bit ARGB y renderiza el texto con el color dado, usando "alpha blending"[39] para evitar la caja del método anterior. También se usa antialiasing.

39 http://en.wikipedia.org/wiki/Alpha_compositing#Alpha_blending

Captura donde se aprecia el acabado de cada método.

A continuación el código completo de la prueba anterior donde se puede observar el uso de los tres modos.

```
#include <iostream>
#include <SDL/SDL.h>
#include <SDL/SDL_ttf.h>

using namespace std;

//Puntero para la pantalla
SDL_Surface *pantalla = NULL;
//Puntero para la fuente
TTF_Font *font;

enum Modos_Pintado {Solid, Shaded, Blended};

SDL_Surface* inicializarSDL(){

	// Iniciamos el subsistema de video de SDL
	if(SDL_Init(SDL_INIT_VIDEO) < 0) {

		cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
		exit(1);
	}

	atexit(SDL_Quit);

	SDL_WM_SetCaption("Prueba SDL_ttf", NULL);

	// Comprobamos que sea compatible el modo de video
	if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {

		cerr << "Modo no soportado: " << SDL_GetError();
		exit(1);
```

```
	}

	// Establecemos el modo de video
	return SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
}

void pintarCadena(int x, int y, string text, Modos_Pintado modo){

	SDL_Rect pos;
	pos.x = x;
	pos.y = y;

	SDL_Color rojo = {255,0,0};
	SDL_Color amarillo = {0,255,0};
	SDL_Surface *text_surface;

	switch (modo){

		case Solid:

		   text_surface = TTF_RenderText_Solid(font, text.c_str(), rojo);
		   break;

		case Shaded:

		   text_surface =TTF_RenderText_Shaded(font,text.c_str(),rojo, amarillo);
		   break;

		case Blended:

		   text_surface = TTF_RenderText_Blended(font, text.c_str(), rojo);
		   break;
	}

	if (text_surface != NULL){

		SDL_BlitSurface(text_surface, NULL, pantalla, &pos);
		SDL_FreeSurface(text_surface);

	}
	else {
		 //error
		cerr << "No se pudo escribir en pantalla: " << SDL_GetError() << endl;
	}
}

int main(int argc, char *argv[]) {

	// Variable para la gestión del evento de salida
	SDL_Event evento;

	pantalla = inicializarSDL();
	if(pantalla == NULL) {

		cerr <<"No se pudo establecer el modo de video: "<<SDL_GetError()<< endl;
		exit(1);
```

```
	}

	//Inicializando biblioteca ttf
	if(TTF_Init() == -1) {

		cerr << "No se pudo iniciar TTF: " << SDL_GetError() << endl;
		exit(1);
	}

	// Abrir fuente y establecer tamaño
	font = TTF_OpenFont("../font/NewShape.ttf",30);

	if(font == NULL) {

		cerr << "No se pudo abrir la fuente: " << SDL_GetError() << endl;
		exit(1);
	}

	pintarCadena(100, 50, "Modo Solid", Solid);
	pintarCadena(100, 125, "Modo Shaded", Shaded);
	pintarCadena(100, 200, "Modo Blended", Blended);

	// Mostramos la pantalla
	SDL_Flip(pantalla);

	//Cerramos la fuente
	TTF_CloseFont(font);

	bool gogogo = true;

	while(gogogo) {

		SDL_WaitEvent(&evento);
		if (evento.type == SDL_QUIT)
			gogogo = false;
	}

	SDL_FreeSurface(pantalla);
	//cerrar SDL_ttf
	TTF_Quit();

	return 0;
}
```

SDL_mixer

Esta biblioteca adicional simplifica el trabajo con el audio cuando usemos SDL. Nos ofrece funciones para cargar y reproducir música y efectos de sonido. Una de las mayores ventajas de SDL_mixer es que se encarga de realizar la mezcla de los canales de audio de forma automática, aunque también se puede especificar otra función propia para la mezcla. SDL_mixer distingue entre la reproducción de sonidos y la reproducción de la música del juego (para la que reserva un canal exclusivo).

Para poder utilizar esta librería el subsistema de audio debe estar inicializado. Soporta los

siguientes formatos:

- WAVE/RIFF (.wav) .
- AIFF (.aiff) .
- VOC (.voc) .
- MOD (.mod .xm .s3m .669 .it .med y más).
- MIDI (.mid) .
- OggVorbis (.ogg) requiere las bibliotecas ogg/vorbis en el sistema.
- MP3 (.mp3) requiere las bibliotecas SMPEG en el sistema.

Su funcionamiento es sencillo, consta de dos tipos principales:

- *Mix_Chunk* : en los videojuegos existen gran cantidad de efectos sonoros, todos estos sonidos se abstraen individualmente como un chunk. Los chunks pueden ser cargados de ficheros de disco o desde la memoria. Normalmente se almacenan en un fichero de formato WAV o VOC
- *Mix_Music* : La música en se maneja de forma parecida a un efecto de sonido, aunque existe una diferencia importante: sólo puede reproducirse una canción al mismo tiempo.

Como es habitual en la SDL lo primero que se debe hacer es inicializar la biblioteca, después de haber inicializado el subsistema de audio en la inicialización de la SDL. Y cerrarla adecuadamente una vez se ha terminado de usarla. Para ello se utilizan las siguientes funciones, incluyendo su propio archivo cabecera.

```
#include "SDL_mixer.h"

int Mix_OpenAudio(int frequency, Uint16 format, int channels, int chunksize);
void Mix_CloseAudio();
```

- *frequency* : especifica la frecuencia (en Hertzios) de reproducción del sample.
- *format* : especifica el formato del sample (bits y tipo).
- *channels* : número de canales para la salida, 1 mono 2 estéreo.
- *chunksize* : es el tamaño de chunk que queremos especificar en nuestra aplicación. La documentación aconseja usar un valor de 4096.

Una vez inicializada la biblioteca podemos empezar a cargar los diferentes archivos de audio que se usaran en la aplicación. Le pasamos a la función como parámetro la dirección del fichero que queremos cargar y esta se encarga de devolvernos un puntero a la estructura Mix_Chunk o Mix_Music con el que trabajar. Si ocurre algún error la función devolverá el valor NULL.

```
...
        Mix_Music *musicaFondo;
        Mix_Chunk *salto, *impacto;
...
        Mix_AllocateChannels(6);
```

```
        // Cargamos música
        musicaFondo = Mix_LoadMUS("../sonidos/musicaFondo.ogg");
        if ( musicaFondo == NULL ) {

                cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
                exit(-1);
        }

       // Cargamos efectos
        salto = Mix_LoadWAV("../sonidos/salto.wav");
        if ( salto == NULL ) {

                cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
                exit(-1);
        }

        impacto = Mix_LoadWAV("../sonidos/impacto_bala.ogg");
        if ( impacto == NULL ) {

                cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
                exit(-1);
        }
...
```

Aunque la biblioteca se encarga de la mezcla, es conveniente informar a SDL_mixer de cuántos canales queremos utilizar con la función *MIX_AllocateChannels*, deberemos pasarle un número que representa cuantos sonidos simultáneos queramos poder reproducir

Una vez utilizado, y si ya no es necesaria su presencia en memoria principal, podemos liberar el chunk mediante la siguiente función.

```
...
        // liberamos recursos
        Mix_FreeMusic(musicaFondo);
        Mix_FreeChunk(salto);
        Mix_FreeChunk(impacto);
...
```

Una vez que hemos cargado los sonidos y la música contamos con funciones para comenzar a reproducirlos, pausar y reanudar la reproducción. Para iniciar la reproducción hay que diferenciar entre música y efecto, para el primero se usa *Mix_PlayMusic*, cuyos parámetros son el puntero a la música previamente cargada y el número de veces que queremos que se reproduzca, siendo indefinido si introducimos un -1 y una sola vez con 0. Para reproducir chunks se usa la función *Mix_PlayChannel*, cuyos dos segundos parámetros son iguales que para la anterior función, salvo que ahora se le pasa un chunk en vez de una música. El primer parámetro es el canal en el que queremos que reproduzca el sonido, si le pasamos -1 lo hará la biblioteca automáticamente. Estas dos funciones devuelven un entero que representa el canal en el que se reproducirá el sonido o música.

```
...
     canalMusicaFondo = Mix_PlayMusic(musicaFondo, -1);
```

```
...
     Mix_PlayChannel(-1, salto, 0);
...
```

SDL_net

Esta biblioteca no se usa en el proyecto, sin embargo durante la fase de estudio y prueba de la SDL no se descartó, por lo tanto se hará un breve repaso a las principales características. Esta biblioteca proporciona funciones para el trabajo en red de una manera sencilla, permite iniciar y controlar sockets de tipo TCP y UDP. Como en los casos anteriores tiene funciones para la inicialización y cierre de la biblioteca.

```
...
	//Inicializando SDL
	if(SDL_Init(0) == -1){

		cerr << "SDL_Init: " << SDL_GetError() << endl;
		exit(1);
	}

	//Inicializando SDL SDL_net
	if(SDLNet_Init() == -1){

		cerr << "SDLNet_Init: " << SDLNet_GetError() << endl;
		exit(1);
	}
.
.
.
	//Cerrando SDL_net
	SDLNet_Quit();

	//Cerrando SDL
	SDL_Quit();
...
```

No se va a explicar el trabajo con sockets en esta memoria, ya que entra dentro de los objetivos, pero sí se realizará un breve exposición de como establecer una conexión básica entre un cliente y un servidor TCP con esta biblioteca. El servidor por su lado tiene las siguientes tareas:

- Abrir un socket en un puerto específico.
- Comprobar si algún cliente quiere conectar.
- Gestionar mensajes de los clientes.

```
...
	//variables
	IPaddress ip;
	TCPsocket server = NULL;
	Uint16 port;
```

```
        //establecer puerto
        port = 5555;

        //Rellenamos la variable ip para su posterior uso
        if(SDLNet_ResolveHost(&ip, NULL, port) == -1){

                cerr << "SDLNet_ResolveHost: " << SDLNet_GetError() << endl;
                exit(1);
        }

        //Abrimos un socket con los datos generados antes
        server = SDLNet_TCP_Open(&ip);
        if(!server){

                cerr << "SDLNet_TCP_Open: " << SDLNet_GetError() << endl;
                exit(1);
        }
...
```

Mediante la función *SDLNet_ResolveHost* se construye la dirección ip, que es el parámetro que necesita la siguiente función *SDLNet_TCP_Open* para abrir un socket por donde establecer las conexiones. El primer parámetro es la variable donde guardar la dirección que se genere, el segundo es el *host*, que al ponerlo a NULL indica que actuaremos como servidor. Y el tercero es el puerto que se usará para la comunicación. La segunda función es más sencilla, usando la dirección creada crea un socket de tipo TCP, que en este caso por tener el *host* a *NULL* aceptará conexiones.

```
...
    TCPsocket client = NULL;
    Ipaddress *remoteip;
    Uint32 ipaddr;
    unsigned int result;
    char buffer[512];
...
    //Comprobar si hay clientes intentando conectar
    client = SDLNet_TCP_Accept(server);

    if(client){
        //conexión aceptada
        //Dirección del cliente
        remoteip = SDLNet_TCP_GetPeerAddress(client);
        if(!remoteip){

                cerr << "SDLNet_TCP_GetPeerAddress: " << SDLNet_GetError() << endl;
                break;
        }
        /Mostramos información del cliente, ip y puerto
        ipaddr = SDL_SwapBE32(remoteip->host);
        cout<<"Conexión aceptada ip: "<<ipaddr<<"  puerto: "<<remoteip->port<< endl;

        //Recibir mensaje
        result = SDLNet_TCP_Recv(client, buffer, 512);
...
    //cerrar conexión con el cliente
```

```
   SDLNet_TCP_Close(client);
```

Mediante la función *SDLNet_TCP_Accept* comprobamos si existe algún cliente intentando conectar, en tal caso se crea un nuevo socket (*cliente*) a través del cual se llevará a cabo la comunicación con ese cliente. Es una llamada no bloqueante por lo que si no hay clientes intentando conectar recibiremos un socket a *NULL* y se continuara la ejecución. Una vez se ha conectado con un cliente se muestra su ip y puerto, y por último se recibe un mensaje del cliente. La recepción se realiza mediante la función *SDLNet_TCP_Recv*, cuyos parámetros por orden de izquierda a derecha son: el socket por el que recibir el mensaje, una variable donde guardar el mensaje y el tamaño máximo del mensaje. La función devuelve el número de bytes recibidos, -1 para errores y 0 cuando la conexión se ha sido cortada por el cliente. Esta función es bloqueante, por lo que se detendrá la ejecución hasta que llegue algo por ese socket. Por último se cierra la conexión con el cliente.

Las tareas del cliente son muy parecidas:

- Conectar con el servidor.
- Enviar mensajes.
- Recibir mensajes

```
...
       IPaddress ip;
      TCPsocket sock;
      Uint16 port;
     char buffer[512];
...
       //Puerto
      port = 5555;
      //Resolver dirección del servidor
      if(SDLNet_ResolveHost(&ip, "localhost", port) == -1){

            cerr << "SDLNet_ResolveHost: " << SDLNet_GetError() << endl;
            exit(1);
      }
      //Abrir el socket
      sock = SDLNet_TCP_Open(&ip);
      if(!sock){

            cerr << "SDLNet_TCP_Open: " << SDLNet_GetError() << endl;
            exit(1);
      }
...
      len = strlen(buffer);
      int result;
      result = SDLNet_TCP_Send(sock, buffer, len);
...
```

Como se puede observar las dos primeras funciones son las mismas usadas para el servidor, con el detalle de que se indica el host en *SDLNet_ResolveHost*, de esta manera cuando se llama a *SDLNet_TCP_Open* se crea un socket de tipo cliente preparado para enviar y recibir, pero no para aceptar conexiones como en el otro caso. Ésta última función no es bloqueante, intentará

conectar y si no es posible tendremos un error del tipo “Imposible conectar con el servidor”. Una vez establecida la conexión podemos enviar y recibir mensajes. El envío es muy parecido a la recepción, la función es *SDLNet_TCP_Send*, el primer parámetro es el socket por donde enviar el mensaje, el segundo es el mensaje en si y el tercero es el tamaño en bytes a enviar. De igual forma que al recibir, la función devuelve el número d bytes enviados.

3.2 Por qué usar SDL

La principal razón por las que se decidió usar la SDL fue la sencillez de la biblioteca, realiza las tareas básicas que todo juego necesita para su desarrollo (gestión de las entradas, video, audio entre otros) y deja en nuestras manos el resto. Nos permite trabajar con múltiples subsistemas de una forma intuitiva y eficaz escondiendo infinidad de detalles. Otra razón importante por la que usar la SDL es la licencia bajo la que se distribuye, GNU LGPL, permitiendo el uso gratuito y acceso al código. Además ha pasado mucho tiempo desde que saliese su primera versión y todavía se sigue trabajando en ella, ampliando funcionalidades y corrigiendo errores, con una gran comunidad en la red. El último aspecto que se consideró fundamental para la elección fue la capacidad multiplataforma de la biblioteca, teniendo en cuenta que en la actualidad coexisten multitud de sistemas diferentes que son usados para ejecutar videojuegos, PC, Mac, Linux, IO´s, Android, XBOX, PlayStation, etc..., es una gran ventaja el poder tener código independiente que sea compilable con el mayor número de plataformas.

Otro motivo importante por el que se eligió la SDL, es porque se quería tener una primera experiencia a bajo nivel, de la misma manera que se hizo durante los estudios en otros casos. Como se verá en el siguiente apartado donde se tratan las alternativas disponibles a SDL, existen soluciones que resuelven multitud de problemas relacionados con el desarrollo de videojuegos, sistema colisiones, gestión del mapa, renderizado, etc. Usando estas herramientas especializadas se consiguen mejores resultados en menor tiempo, aunque el tiempo de aprendizaje es mucho mayor que con bibliotecas tipo SDL.

Logo oficial.

Otros aspectos menos importantes pero que también colaboraron para tomar la decisión fueron, por un lado el que esté escrita en lenguaje C, el cual fue estudiado durante la carrera lo que facilitaría el trabajo. La documentación oficial está bastante bien, con muchos ejemplos y explicaciones de algunos de los problemas que se pueden encontrar en el desarrollo de videojuegos

con esta biblioteca. Además por el tiempo que ha pasado desde su lanzamiento se puede encontrar por internet multitud de tutoriales y ejemplos de uso, además de libros que tratan específicamente sobre SDL y desarrollo de videojuegos. Para terminar, SDL nos permite trabajar cómodamente en 2D y en 3D mediante OpenGL.

3.3 Alternativas a SDL

En la actualidad existen infinidad de alternativas a SDL para el desarrollo de videojuegos, desde motores de videojuegos totalmente especializados como CryEngine 3 hasta bibliotecas parecidas a SDL como Allegro, o la solución de Microsoft, XNA.

Un motor de videojuegos o Game Engine[40] es una sistema software diseñado para la creación y desarrollo de videojuegos. Actualmente existe gran cantidad de en el mercado, tanto libres como de pago, orientados a las distintas plataformas. Estas aplicaciones proporcionan una serie de funcionalidades entre las que se incluyen un motor de renderización o "render" para gráficos 2D y 3D, un motor de física (motor de colisiones o motor de detección de colisiones), soporte para audio y sonidos, scripts, animaciones, inteligencia artificial, juego en red, streaming de contenidos, manejo de memoria, multijugador etc.

A continuación se presenta una lista con las principales alternativas estudiadas antes de tomar la decisión de usar SDL, presentados de menor a mayor sofisticación.

- **Sfml :** es una API multiplataforma, escrita en C++ pero también disponible en C, Python y Ruby. Su propósito principal es ofrecer una biblioteca alternativa a la biblioteca SDL, usando un enfoque orientado a objetos. Gracias a sus numerosos módulos, SFML puede ser usada como un sistema mínimo de ventanas para interactuar con OpenGL o como una biblioteca multimedia cuyas funcionalidades permiten al usuario crear videojuegos y programas interactivos. Muy parecida a SDL, su versión 2.0 también está a punto de salir. El motivo por el que no se seleccionó esta opción fue que SDL cuenta con más años de desarrollo y una gran comunidad online.

- **Allegro :** es una biblioteca para la programación de videojuegos desarrollada en C. Creada originalmente para Atari ST, con el tiempo ha ido evolucionando y hoy día es compatible con plataformas como DOS, Linux, Windows, QNX, y MacOS X. Sus funcionalidades principales están orientadas a los gráficos, sonidos, entrada de usuario (teclado, ratón y mandos de juego) y temporizadores. También tiene funciones matemáticas en punto fijo y coma flotante y funciones 3D mediante OpenGL. También muy parecida a SDL, más orientada al desarrollo de videojuegos y menor nivel de abstracción, se decidió usar SDL por la cantidad de documentación comparada con Allegro y su mayor nivel de abstracción.

- **Box2D** es una biblioteca libre que implementa un motor físico en dos dimensiones. Está programada en C++ por Erin Catto, y se distribuye bajo la licencia zlib. Algunos de los videojuegos más populares que la emplean son:
    - Crayon Physics Deluxe
    - Rolando
    - Fantastic Contraption

40 http://en.wikipedia.org/wiki/Game_engine

- Incredibots
- Angry Birds
- Tiny Wings
- Transformice

También se usa en mucho videojuegos Flash, o para iPhone, iPad y Android mediante el framework Corona y el framework AndEngine. Se llegó a probar en las primeras fases del proyecto, es una biblioteca muy interesante pero hubiese desviado el proyecto hacia el uso de esta biblioteca y el desarrollo de niveles en vez de centrarlo en el desarrollo de videojuegos .

- **M.U.G.E.N. :** es un motor de videojuegos de lucha en dos dimensiones (2D) gratuito lanzado el 17 de julio de 1999 por la empresa japonesa Elecbyte, desarrollado usando el Lenguaje de Programación C que originalmente utilizo la biblioteca de programación "Allegro". La versión actual utiliza la biblioteca "SDL". La licencia de MUGEN literalmente es definida como "M.U.G.E.N is free for non-commercial use", es decir, se distribuye sin razones comerciales. Aunque se pudo desarrollar versiones alternativas como NOMEN, basada en componentes exclusivamente abiertos bajo GPL.

- **Microsoft XNA :** es un API de Microsoft orientado al desarrollo de videojuegos para las plataformas XBOX 360, Zune y PC. Técnicamente es un “Marco de Trabajo” (Framework), basado en .NET Framework 2.0; aunque en una implementación que proporciona un manejo optimizado para la ejecución de videojuegos. XNA pretende simplificar el desarrollo de juegos al programador, y a través de su estructura facilita la gestión de gráficos, sonido, dispositivos, etc. Sin embargo, Microsoft ha decidido dejar de desarrollar XNA, y en su lugar se centrará en DirectX[41]. Este último detalle fue uno de los principales motivos para no escoger esta opción, unido a que es una API de pago y únicamente para las plataformas Microsoft.

- **Game Maker :** es una herramienta de desarrollo rápido de aplicaciones, basada en un lenguaje de programación interpretado y un kit de desarrollo de software (SDK) para desarrollar videojuegos, creado por el profesor Mark Overmars en el lenguaje de programación Delphi, y orientado a usuarios novatos o con pocas nociones de programación. El programa es gratuito, aunque existe una versión comercial ampliada con características adicionales. Actualmente se encuentra en su versión 8.1. Overmars liberó la primera versión pública el 15 de noviembre de 1999.

- **JmonkeyEngine :** es un motor de videojuegos libre orientado al desarrollo moderno de videojuegos en tres dimensiones. jME es uno de los engines 3D más completos escritos en Java, es una API basada en árbol de nodos, completamente de código abierto y publicado bajo la licencia BSD. No se escogió por la decisión de hacer juegos en 2D.

- **Havok Game Dynamics SDK :** es un motor físico (simulación dinámica) utilizada en videojuegos y recrea las interacciones entre objetos y personajes del juego. Por lo que detecta colisiones, gravedad, masa y velocidad en tiempo real llegando a recrear ambientes mucho más realistas y naturales. Havok en sus últimas versiones se ejecuta por entero por hardware mediante el uso de la GPU, liberando así de dichos cálculos a la CPU. Este se

41 http://www.gamasutra.com/view/news/185894

apoya en las bibliotecas Direct3D y OpenGL

- **Frostbite :** es el motor gráfico desarrollado y utilizado por EA Digital Illusions CE. Con el paso de los años, DICE ha ido mejorando el motor desde el Frostbite 1.0 hasta el Frostbite 2 (este último usado en Battlefield 3 y Need for Speed: The Run). Actualmente se encuentra en su tercera versión aparecerá en Need for Speed: Rivals , Battlefield 4,Batlefront, etc... para las nuevas consolas de próxima generación como PlayStation 4, Xbox One y PC. Es una opción de pago.

- **Unity 3D:** es un motor de videojuego multiplataforma creado por Unity Technologies. Está disponible como plataforma de desarrollo para Windows y OS X, y permite crear juegos para Windows, OS X, Linux, Xbox 360, PlayStation 3, Wii, Wii U, iPad, iPhone y Android. Gracias al Plug-In Web de Unity, también se pueden desarrollar juegos de navegador, para Windows y Mac. Hay dos licencias principales para desarrolladores: Unity y Unity Pro, que está disponible por $1500. La versión Pro tiene características adicionales, tales como render a textura, determinación de cara oculta, iluminación global y efectos de post-procesamiento. La versión gratuita, por otro lado, muestra una pantalla de bienvenida (en juegos independientes) y una marca de agua (en los juegos web) que no se puede personalizar o desactivar. Es una gran herramienta para el desarrollo de videojuegos, pero implica mucho tiempo el aprendizaje.

Entorno de desarrollo Unity 3D.

- **CryEngine 3 :** es un motor de videojuegos creado por la empresa alemana Crytek, su primera versión (CryENGINE 2003) era un motor de demostración para la empresa Nvidia, que al demostrar un gran potencial se implementa por primera vez en el videojuego Far Cry, desarrollado por la misma empresa creadora del motor. El CryEngine 3 se presenta en 2009, es compatible las plataformas Windows, Wii U, Playstation 3 y 4, Xbox 360 y Xbox One. En la actualidad es uno de los motores más importantes de la industria con el que se han

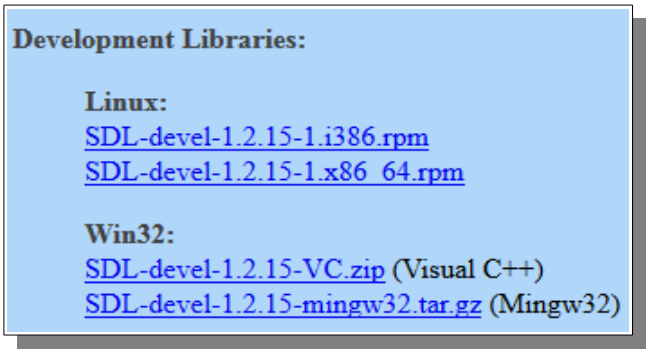
realizado juegos de éxito como las sagas FarCry y Crysis. Otro de sus puntos fuertes es que es gratuito para usos no comerciales y educativos, poniendo al alcance de cualquiera una herramienta de gran calidad. Al igual que con Unity la mayor desventaja es el tiempo que se necesita para tener unos conocimientos mínimos para poder empezar en serio.

Captura del juego "Crysis 3" desarrollado con el CryEngine3.

3.4 Preparación del entorno, SDL y Eclipse

En este punto del documento se analizarán los diferentes pasos a seguir para trabajar con la biblioteca SDL y el entorno de programación Eclipse. Se eligió Eclipse por se un entorno potente que ya se conocía por haberlo utilizado en varias asignaturas de la universidad, además es multiplataforma y libre, bajo la Licencia Pública Eclipse (EPL).

Una vez instalado Eclipse para trabajar con C++ debemos descargar los binarios de SDL de la página oficial[42]. Para trabajar con Eclipse y SDL deberemos usar MinGW[43], una implementación de los compiladores GCC para la plataforma Win32, que permite migrar la capacidad de este compilador en entornos Windows.



Primeros pasos por lo tanto, descargar e instalar Eclipse y MinGW. A continuación descargamos los citados binarios de SDL que colocaremos "dentro" de MinGW. Dentro del archivo comprimido que contiene los binarios encontramos además varias carpetas, una con ejemplos, otra con la documentación oficial y una con las páginas "man" asociadas a SDL. Las carpetas que nos

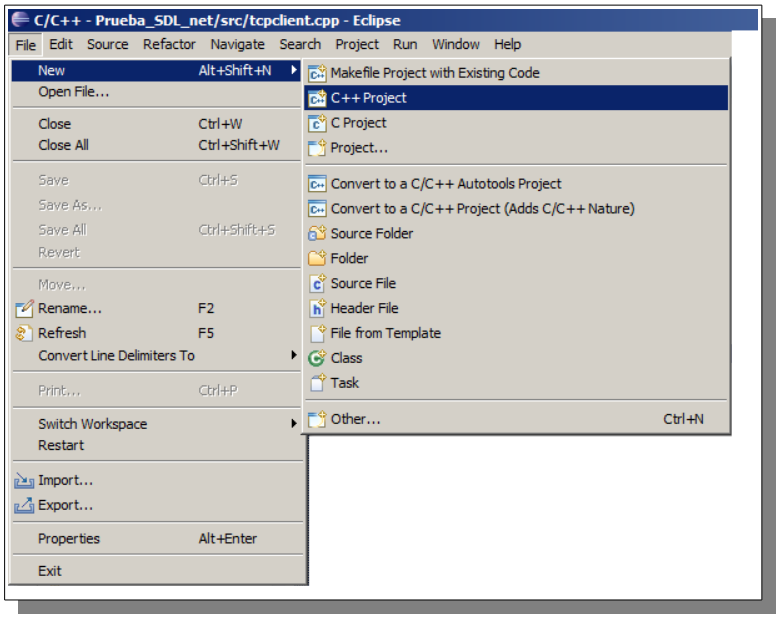
42 http://www.libsdl.org/download-1.2.php
43 http://es.wikipedia.org/wiki/MinGW

interesan en este momento son; *bin* donde encontraremos la biblioteca de enlace dinámico *SDL.dll*, *lib* donde se encuentran las bibliotecas estáticas **.a* e *include* donde se guardan los archivos cabecera **.h*. Habiendo instalado MinGW en *c:\mingw* las siguientes instrucciones organizan los archivos para poder usar SDL en Eclipse.

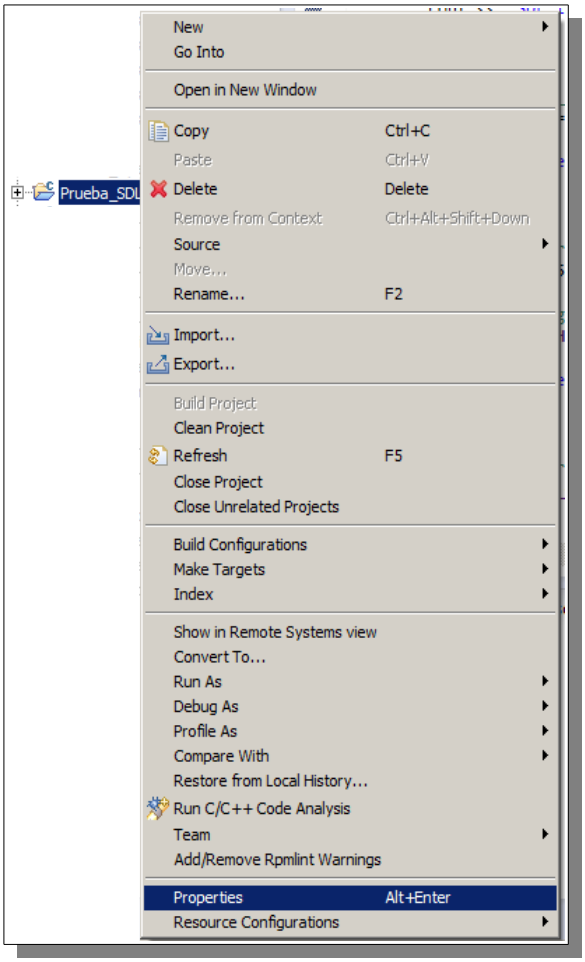
- El archivo *SDL.dll* se debe colocar en la carpeta *c:\windows\system32* o en *c:\mingw\bin*.
- Las bibliotecas estáticas hay que colocarlas en *c:\mingw\lib*.
- Por último las cabeceras las copiamos a *c:\mingw\include\SDL*.

De esta manera MinGW ya tiene todos los archivos necesarios para trabajar con SDL, sólo queda configurar el proyecto dentro de Eclipse para que use MinGW y SDL. A continuación se presentan los pasos a realizar para llevar a cabo esta configuración, que se debe realizar siempre que se comience un nuevo proyecto dentro de Eclipse.

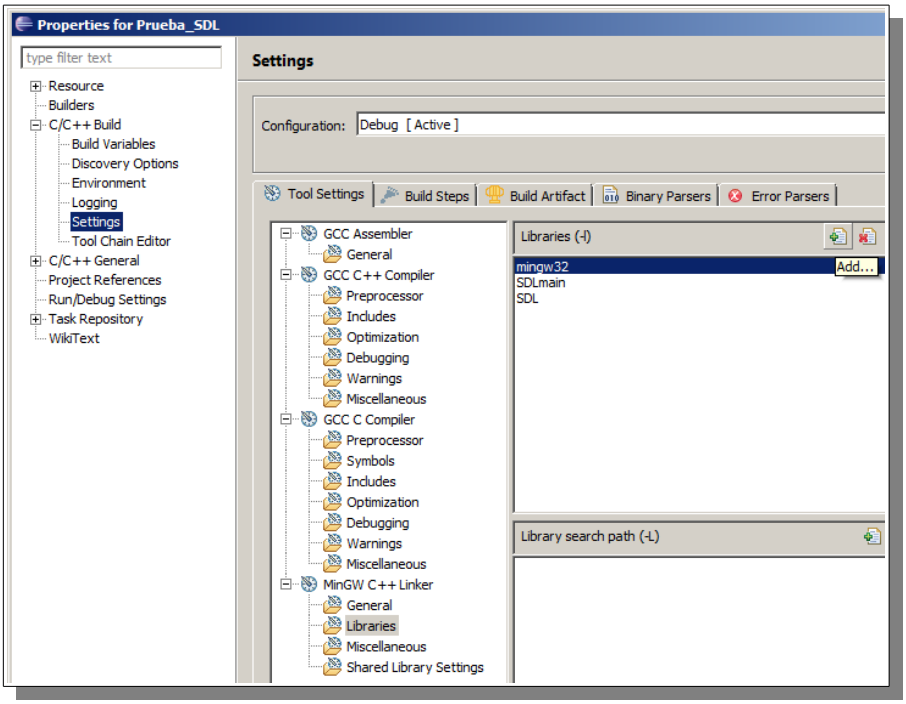
- Creamos un nuevo proyecto C++, de forma habitual.



- Luego añadimos las bibliotecas necesarias en las propiedades del proyecto.



- Dentro del menú “C/C++ Build”, en “Settings”, dentro de “Libraries” añadimos los nombres de las bibliotecas adicionales. Deberemos añadir, *`mingw32`*, *`SDLmain`* y *`SDL`*.

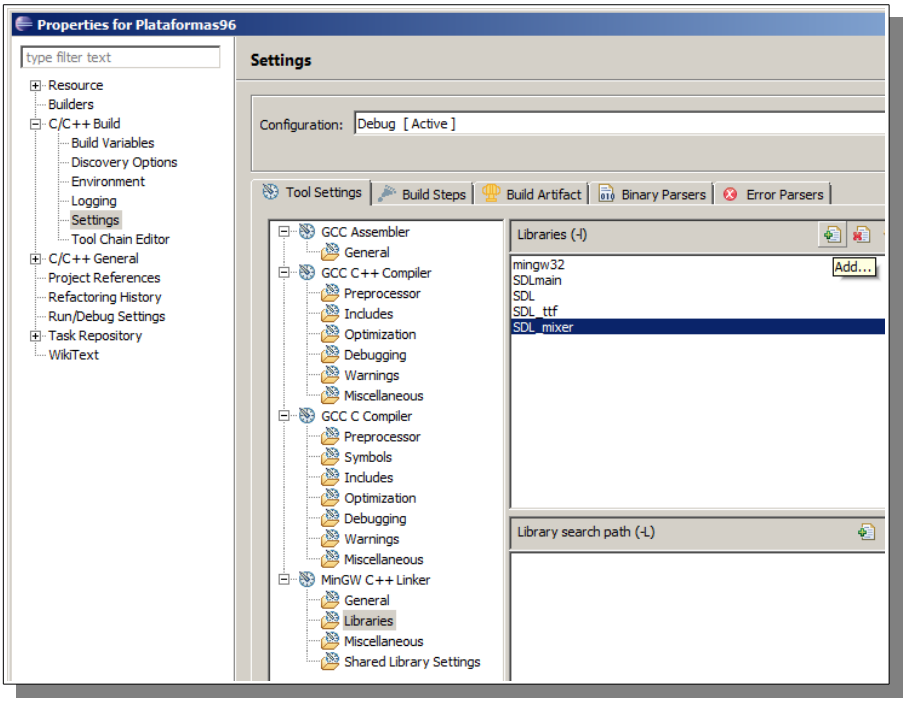


Una vez completados estos pasos queda el proyecto preparado para desarrollar con SDL. A continuación se comenta como "instalar" las bibliotecas adicionales a SDL vistas en el capítulo anterior. Los pasos a seguir son muy parecidos a los anteriores e iguales para cada una de las bibliotecas.

1. Lo primero es descargar la biblioteca en cuestión, en la documentación oficial de SDL se pueden encontrar enlaces a las diferentes páginas oficiales de éstas bibliotecas. Deberemos bajar la versión para desarrollo, marcada como "devel".

2. La instalación es igual que con la SDL, colocamos los diferentes archivos que trae la biblioteca donde deben. El contenido de la descarga es siempre la misma, una carpeta con el nombre "include" donde encontramos el archivo cabecera que colocaremos con los demás de SDL en *c:\mingw\include\SDL*. También encontramos otra carpeta con el nombre "lib" que contiene las bibliotecas dinámicas necesarias .dll y la biblioteca en sí .lib. Como antes las .dll las colocaremos en la carpeta *c:\windows\system32* o en *c:\mingw\bin*. Y las .lib en *c:\mingw\lib*.

3. Por último en Eclipse deberemos añadir la biblioteca a usar en las propiedades del proyecto, igual que se hizo antes para la SDL y MinGW, usando el nombre del archivo cabecera sin la extensión.



Un detalle a tener en cuenta es que la aplicación que se desarrolle con SDL y las bibliotecas adicionales necesitará para su ejecución las bibliotecas dinámicas (.dll) de las bibliotecas implicadas. Bastará con tener una copia de éstas bibliotecas (.dll) junto al ejecutable para que las use.

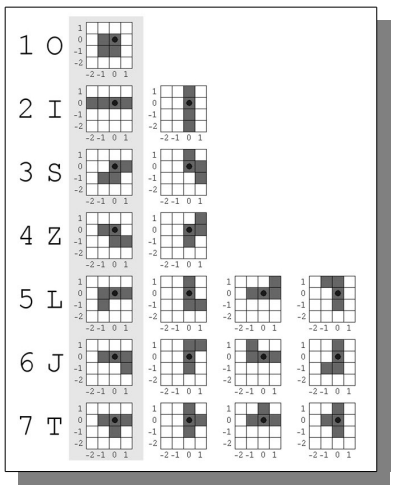
4. Primer Desarrollo: Tetris

Una vez estudiadas las diferentes herramientas que se han utilizado para el desarrollo del proyecto, se presenta el primer prototipo. Se ha escogido reproducir el conocido videojuego Tetris por varias razones: es un juego muy popular y conocido por el alumno, las mecánicas del juego son sencillas y el apartado gráfico se puede solucionar mediante primitivas, en este caso rectángulos. De esta manera se pretende tener un primer contacto con la SDL y el entorno de desarrollo.

4.1 Análisis

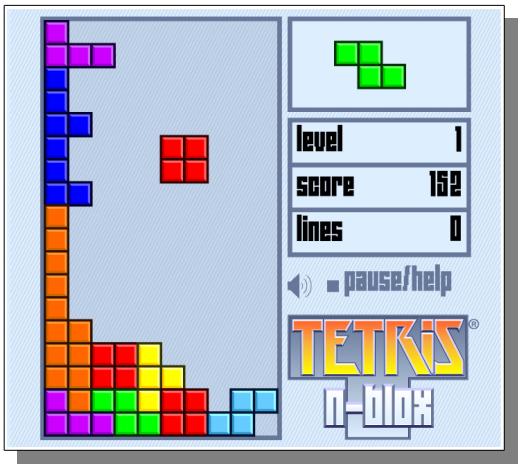
Como se ha comentado antes el Tetris es un juego de tipo puzzle, en el que tenemos que ir organizando distintos tetriminos, figuras geométricas compuestas por cuatro bloques cuadrados unidos de forma ortogonal, que caen de la parte superior de la pantalla. El jugador no puede impedir esta caída pero puede decidir la rotación de la pieza (0°, 90°, 180°, 270°) y en qué lugar debe caer. Cuando una línea horizontal se completa, esa línea desaparece y todas las piezas que están por encima descienden una posición, liberando espacio de juego y por tanto facilitando la tarea de situar nuevas piezas. La caída de las piezas se acelera progresivamente. El juego acaba cuando las piezas se amontonan hasta salir del área de juego. A continuación se realizará un análisis sobre las reglas y mecánicas que afectan al juego y los diferentes componentes que habrá que implementar.

- Piezas: son todas las combinaciones posibles de ordenar cuatro cuadrados de forma que al menos un lado de cada uno quede contiguo al de otro. Además estas piezas se pueden girar, siendo cada giro de 90º, hacia la derecha o a la izquierda. Algunas piezas por su forma no se verán afectadas por el giro.



Piezas y giros.

- Mapa: Está formado por diez columnas y veinte filas, cada casilla del tamaño de un cuadrado de las piezas.



Tetris, página oficial[44].

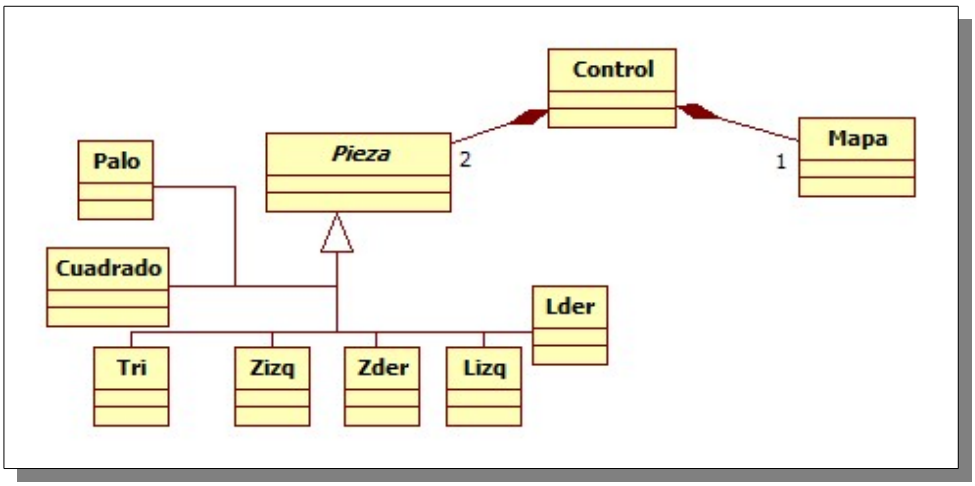
- Reglas y Mecánicas:

1. El juego comienza con el mapa completamente limpio.
2. Durante el juego habrá siempre una pieza dentro del mapa controlada por el jugador.
3. El jugador con la pieza que está bajo su control podrá, girarla a favor y en contra de las agujas del reloj, moverla a izquierda y derecha y hacerla caer más rápido, respetando los límites del mapa.
4. La pieza controlada por el jugador irá cayendo a una velocidad determinada hasta que colisione con el mapa, el movimiento de la pieza no es lineal sino a saltos del tamaño de uno de los cuadrados que forman las piezas y el mapa.
5. Una vez la pieza a sido colocada pasa a formar parte del mapa y se introduce la siguiente pieza al juego.
6. Se mostrará en un lateral la siguiente pieza a ser controlada por el jugador.
7. La velocidad de caída de las piezas aumenta con el paso de los niveles.
8. Se pasa de nivel cada diez lineas completadas.
9. Las filas totalmente llenas se limpiaran del mapa y se actualizará éste de forma que lo que había por encima de esta o estas lineas que han desaparecido ocupe su posición.
10. El juego acaba cuando la nueva pieza que entra en el mapa colisiona en la posición inicial.

44 http://www.tetris.com/

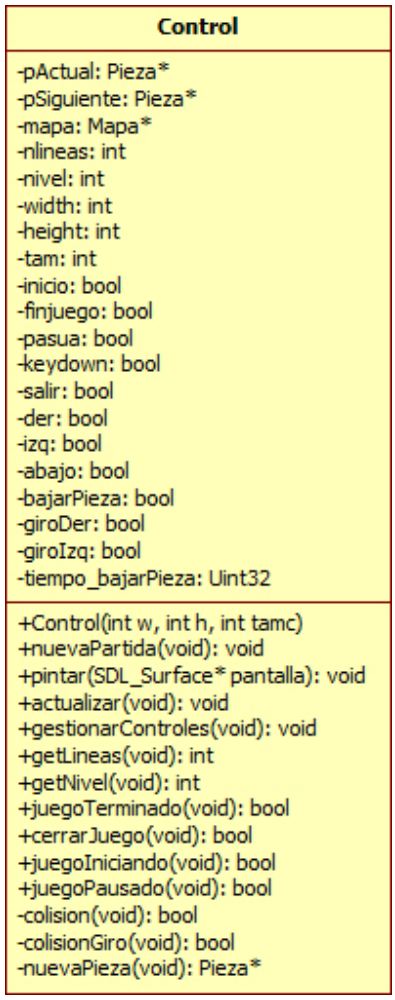
4.2 Diseño

A continuación se presenta el modelo de clases pensado y se comentan sus principales características.

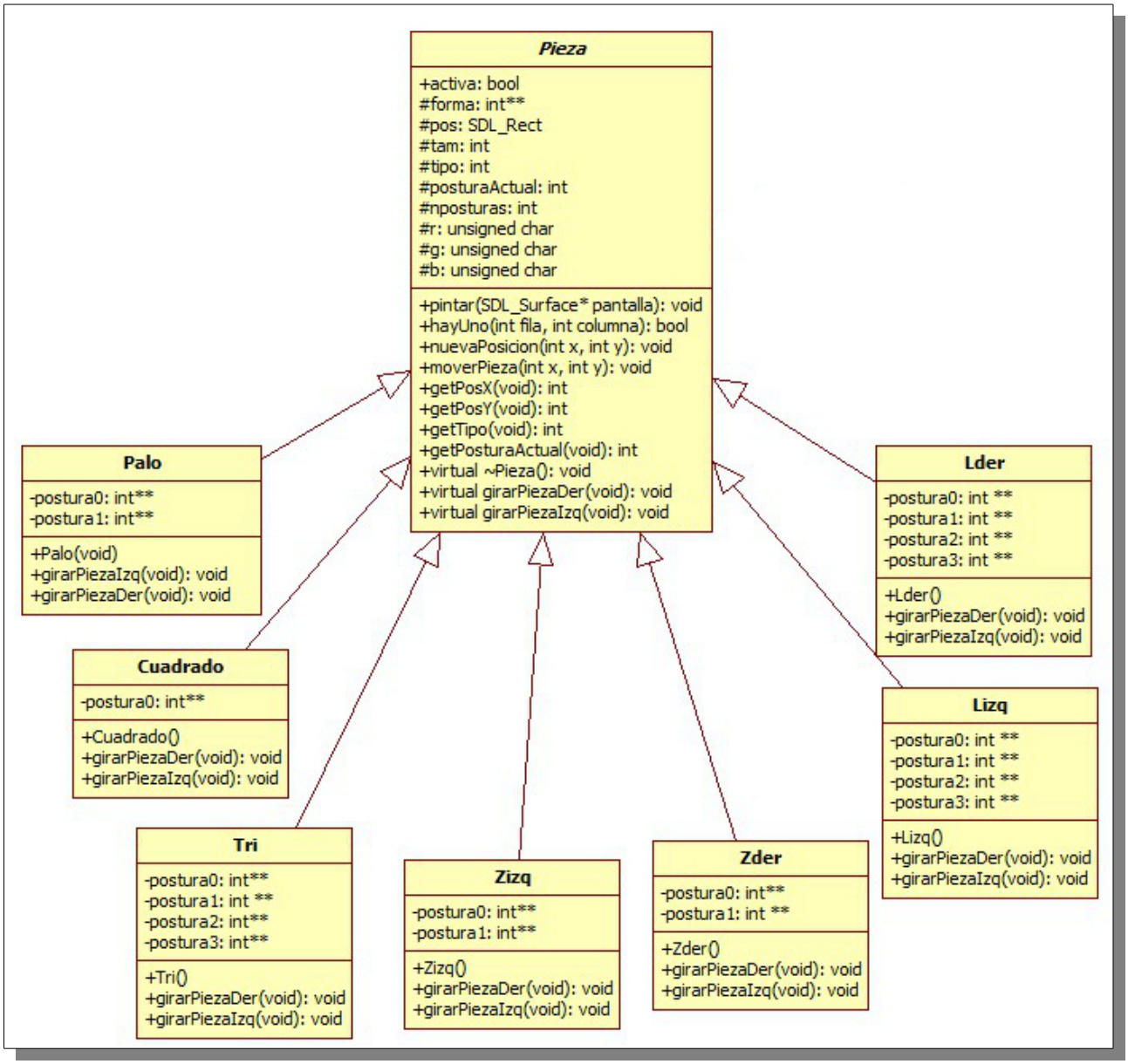


La clase *Control* se encarga de la lógica de la aplicación, la gestión de los controles y el pintado de los elementos del juego, piezas y mapa.

- Entre sus atributos se encuentran el mapa y dos piezas, una es la que el jugador controla (*pActual*) y la otra sirve para visualizar la siguiente (*pSiguiente*).
- El método principal es *actualizar()*, ya que es el ahí donde se implementan la mayoría de reglas y mecánicas del juego.
- Esta clase es además la encargada de gestionar la entrada de teclado y los diferentes eventos que se puedan dar mediante el método *gestionarControles()*.
- Controla los diferentes estados por los que puede pasar el juego; iniciando, en pausa o terminando.
- Controla que las colisiones de las piezas con el mapa y genera de manera aleatoria la siguiente pieza que entrará en juego.



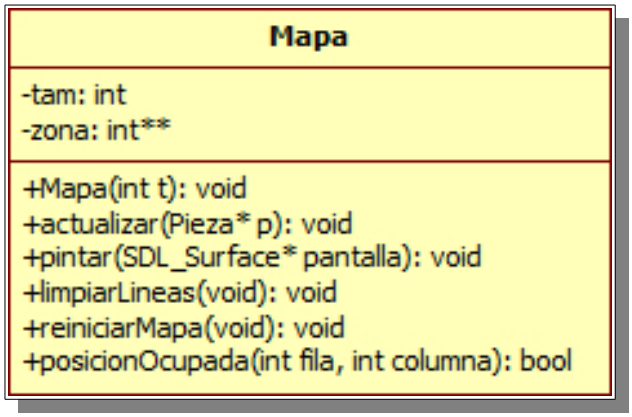
La clase abstracta `Pieza` es la base sobre la que se implementaran las características descritas de los tetriminos, siendo cada uno de ellos derivados de esta.



La clase padre proporciona la mayoría de funcionalidades, entre ellas el pintado y el movimiento de la pieza. Los atributos `r, g y b` representan las tres componentes que forman el color que se usará para pintar cada pieza. Para la estructura de las piezas se ha utilizado un matriz de cuatro por cuatro enteros, donde un uno significa que esa casilla forma parte de la figura y representa un cuadrado de la misma.

Los giros de cada pieza están representados por las distintas posturas que implementan las clases herederas de `Pieza`. Por eso cada una de estas clases implementa los métodos de giro. No es la mejor solución pero tampoco se quería gastar el tiempo en problemas colaterales.

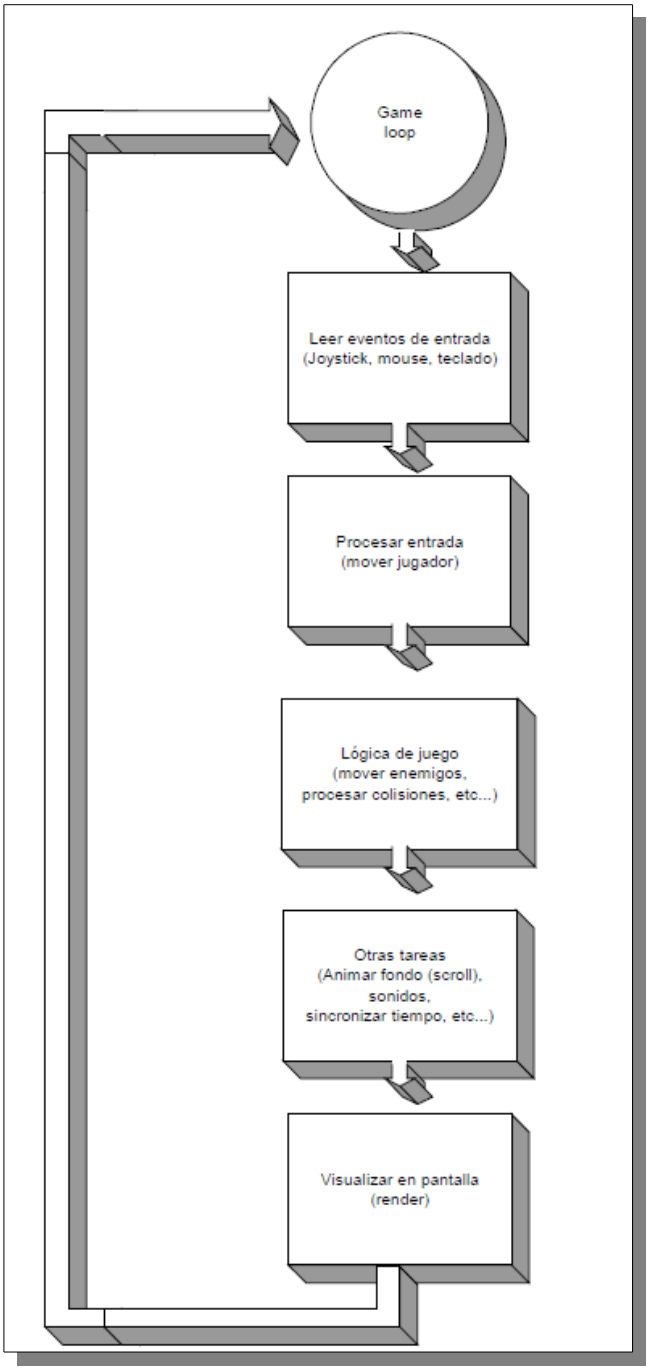
Por último la clase *Mapa* representara el tablero donde se desarrollará el juego, sus funciones principales son: estructura del mapa, proporcionar métodos para controlar las colisiones de las piezas con el mapa y para su pintado.



Se ha implementado mediante una matriz de enteros, en la que el número que contiene cada casilla se usará para determinar colisiones y de que color se pinta cada una. Su tamaño es fijo, doce columnas por veinte filas, siendo variable el tamaño de los subcuadrados. El mapa proporciona métodos para limpiar las lineas llenas, reiniciar el mapa y controlar si una casilla en particular está ocupada.

4.3 Implementación

La estructura del código de un videojuego presenta un esquema básico que se aleja en cierta forma del de otro tipo de software. La actividad principal se desarrolla dentro de el "game loop" o bucle del juego, que sólo se rompe bajo determinadas condiciones ocasionadas por el usuario o generadas por el propio desarrollo de la acción en sí. En la siguiente imagen se puede ver un ejemplo de "game loop", y las tareas que se realizan.



GARCÍA SERRANO A..Programación *de videojuegos con SDL.*

Lo primero que se hace es leer las entradas y traducirlas en acciones, de esta manera conocemos por ejemplo si el usuario quiere moverse, dispara, saltar o en este caso girar la pieza. El siguiente paso es la lógica del juego, esto implica realizar las acciones que el usuario ha introducido, mover a los enemigos, resolver colisiones, etc. Aquí es donde se desarrollan las mecánicas del juego. Por último se pintan los diferentes elementos para generar la escena que visualizará el usuario.

A continuación se muestra el bucle principal usado para este desarrollo, extraído del programa principal de la aplicación, *main.cpp*. Sigue el modelo antes presentado, la gestión de las entradas se lleva a cabo a través del método *gestionarControles()*, la lógica de la aplicación se realiza llamando a *actualizar()* y el pintado mediante el método *pintar()*.

```
...
//Bucle principal
while(enmarcha){

        frametime = SDL_GetTicks();

        //Leer y procesar las entradas
       control->gestionarControles();

       if(!control->juegoIniciando()){

              //Actualizamos si no estamos en pausa ni en game over
              if(!control->juegoPausado() && !control->juegoTerminado())

                     //Lógica del juego
                     control->actualizar();
       }

       //controlamos el tiempo que han tomado los cálculos, si sobra pintamos
       if(SDL_GetTicks() - frametime < MAX_TIEMPO_FRAME) pintar();


       frametime = SDL_GetTicks() - frametime;
       //Si sobra tiempo después de pintar dormimos
       if(frametime < MAX_TIEMPO_FRAME)
              SDL_Delay( Uint32(MAX_TIEMPO_FRAME - frametime) );


       //Se cierra la aplicación
       if(control->cerrarJuego())
              enmarcha = false;
}
...
```

Un detalle a tener en cuenta es que si no se controla, este bucle se ejecutará todas las veces que pueda por segundo, dependiendo de la carga que soporte la CPU en cada momento. Esto generaría un comportamiento inadecuado, la velocidad a la que se desarrolla el juego no sería siempre la misma. Para que esto no suceda se establece el tiempo de duración de cada vuelta por el bucle y se controla el tiempo que se invierte en las diferentes tareas. De esta manera si sobra tiempo una vez se ha leído las entradas, actualizado el estado del juego y pintado se duerme hasta cumplir con el tiempo. Esta es una manera de controlar el número de fotogramas o frames por segundo (fps

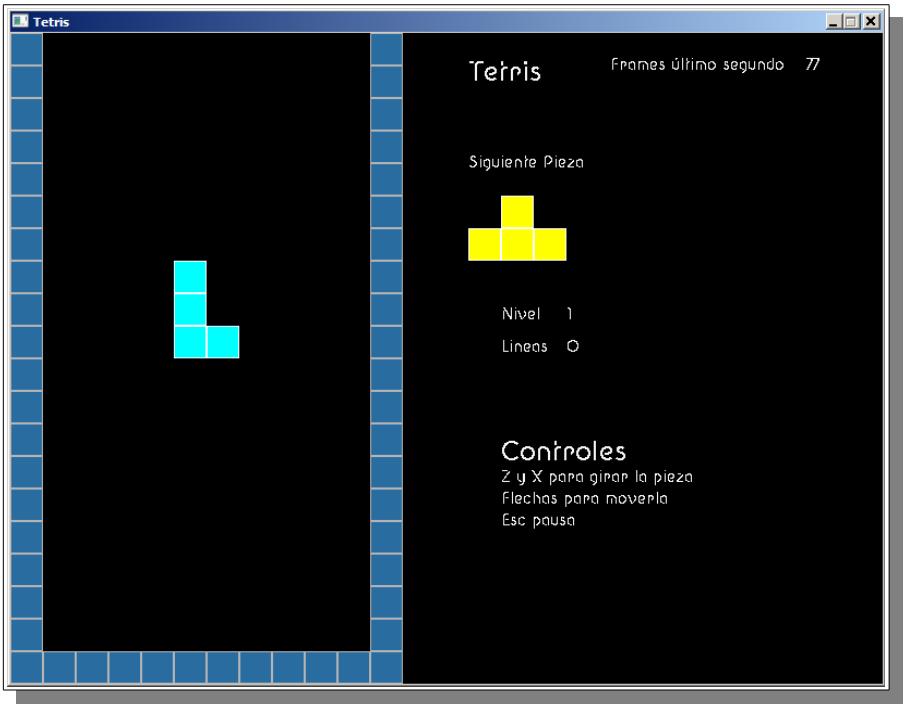
o Hz) a los correrá el juego, la televisión en Europa usa veinticinco fotogramas por segundo y en juegos lo deseable suele rondar los sesenta. Para los dos desarrollos de este proyecto se ha elegido un tiempo de frame de diecisiete milisegundos, lo que equivale a unas sesenta fotogramas por segundo.

Se lleva a cabo también un control del tiempo después de las fases de lectura de entradas y la lógica del juego para comprobar si sobra tiempo para pintar. En el caso de que las fases anteriores consumiesen todo el tiempo se saltaría la fase de pintado para mantener la velocidad del juego, mientras este comportamiento sea únicamente puntual el usuario no debería notar nada.

El resto del programa principal lo componen la inicialización y finalización de la SDL, una función auxiliar para pintar cadenas que se apoya en la librería SDL_ttf y la función *pintar()* que prepara la superficie principal para que el control pinte sobre ella.

```
void pintar(){

	//Limpiamos la pantalla pintando _todo de negro
	SDL_FillRect(pantalla, NULL, SDL_MapRGB(pantalla->format, 0, 0, 0));

	//Pintamos sobre la pantalla los diferentes elementos del juego
	control->pintar(pantalla);
	//aumentamos la cuenta de frames
	cuentafps++;

	//controlamos si ha pasado más de un segundo
	if(SDL_GetTicks() - tiempo_transcurrido > 1000){

		tiempo_transcurrido = SDL_GetTicks();
		fps = cuentafps;
		cuentafps = 0;
	}

	//Información para el usuario
	pintarTextos(pantalla);

	// Mostramos la pantalla, intercambio de buffers
	SDL_Flip(pantalla);
}
```

Primero limpia la superficie, pintando sobre ella un rectángulo negro que la recubre, después pinta los elementos del juego a través de la clase Control. Pinta varios textos, en los que se informa al usuario de los controles, el nivel, el número de líneas completadas e información sobre el estado del juego.



Captura del juego.

4.3.1 Control de eventos de entrada

Esta parte como se ha comentado antes se lleva a cabo en la clase `Control`, a través de la SDL. Se ha elegido el teclado como medio para controlar el juego, usando las siguientes teclas:

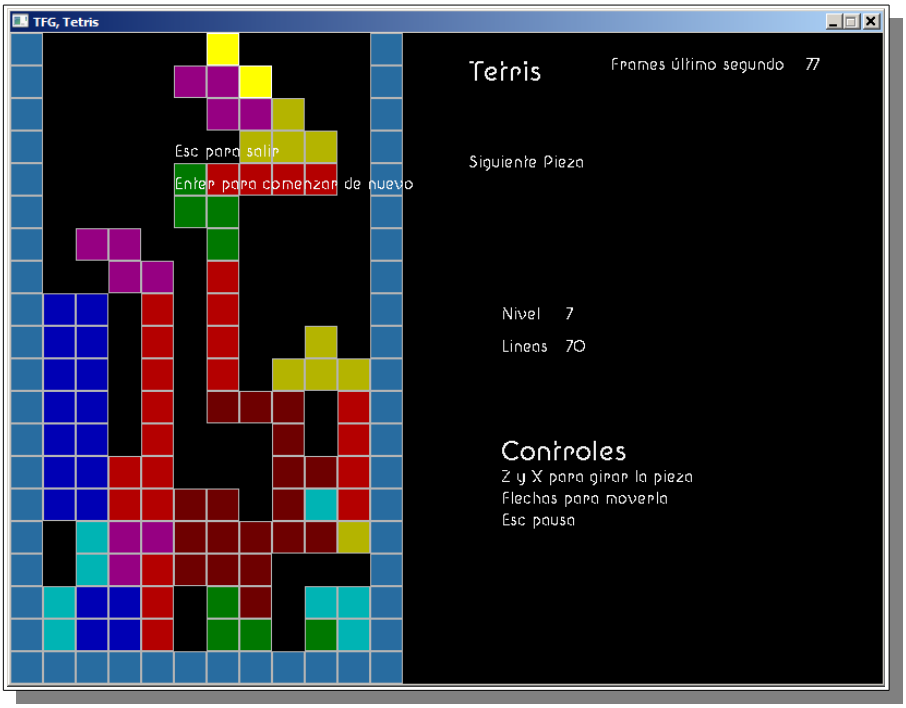
- Flechas direccionales para mover la pieza, izquierda, derecha y abajo.
- "Z" y "X" para girar la pieza.
- "Enter" y "Esc" para empezar, pausar y salir el juego.

Para movernos entre los diferentes estados se usan la tecla "enter" y la tecla "esc". La aplicación comienza con el juego parado, espera que apretemos "enter" para continuar y empezar a jugar o "esc" para abandonar. Durante el juego si se aprieta "esc" se pasa al estado pausa, desde donde se puede cerrar la aplicación pulsando otra vez "esc" o volver al juego pulsado "enter". Cuando el juego acaba se entra en un estado parecido al de entrada, en el que si pulsamos "enter" empieza otra partida y si pulsamos "esc" cerramos la aplicación.

```
void Control::gestionarControles(){

	SDL_Event keyevent;

	while (SDL_PollEvent(&keyevent)){   //Recuperamos eventos

		if (keyevent.type == SDL_KEYDOWN){

			switch(keyevent.key.keysym.sym){

			case SDLK_RETURN:

			//Controla el inicio del juego y la nueva partida tras game over
```

```
                        if(inicio || finjuego){

                                inicio = false;
                                finjuego = false;
                                pausa = false;
                                nuevaPartida();
                                break;
                        }
                        else{

                                //control pausa
                                if(pausa) pausa = false;
                        }
                        break;

                case SDLK_x:

                        if(!pausa)giroder = true;
                        break;

                case SDLK_z:

                        if(!pausa)giroizq = true;
                        break;

                case SDLK_LEFT:

                        if(!pausa)izq = true;
                        break;

                case SDLK_RIGHT:
                        if(!pausa)der = true;
                        break;

                case SDLK_DOWN:
                        if(!pausa)keydown = true;
                        break;

                //Activa la pausa durante el juego,
                case SDLK_ESCAPE:

                        if(inicio) salir = true;
                        else{

                                if(pausa) salir = true;
                                else pausa = true;
                        }
                        break;

                default:
                        break;
                }
        }

        if (keyevent.type == SDL_KEYUP){
```

```
                    switch(keyevent.key.keysym.sym){

                    case SDLK_DOWN:
                          keydown = false;
                          break;

                    default:
                          break;
                    }
              }

        }
}
```



Captura del juego.

Lógica del juego

La lógica del juego se lleva a cabo en el método *actualizar()* de la clase *Control* donde se manejan las dos piezas y el mapa. Las principales funciones son mover y girar la pieza controlada por el usuario por el mapa controlando las colisiones para que se cumplan las normas del juego. Además lleva la cuenta de las líneas que se han completado y el nivel en el que se encuentra la partida, se comienza en nivel uno y cada diez líneas completadas se incrementa. En base al nivel se calcula la velocidad a la que cae la pieza.

En el comienzo de cada partida se llama al método *nuevaPartida()*, donde se inicializan las dos primeras piezas, el mapa y las variables que controlan nivel, número de lineas y el tiempo que controla la caída de la pieza.

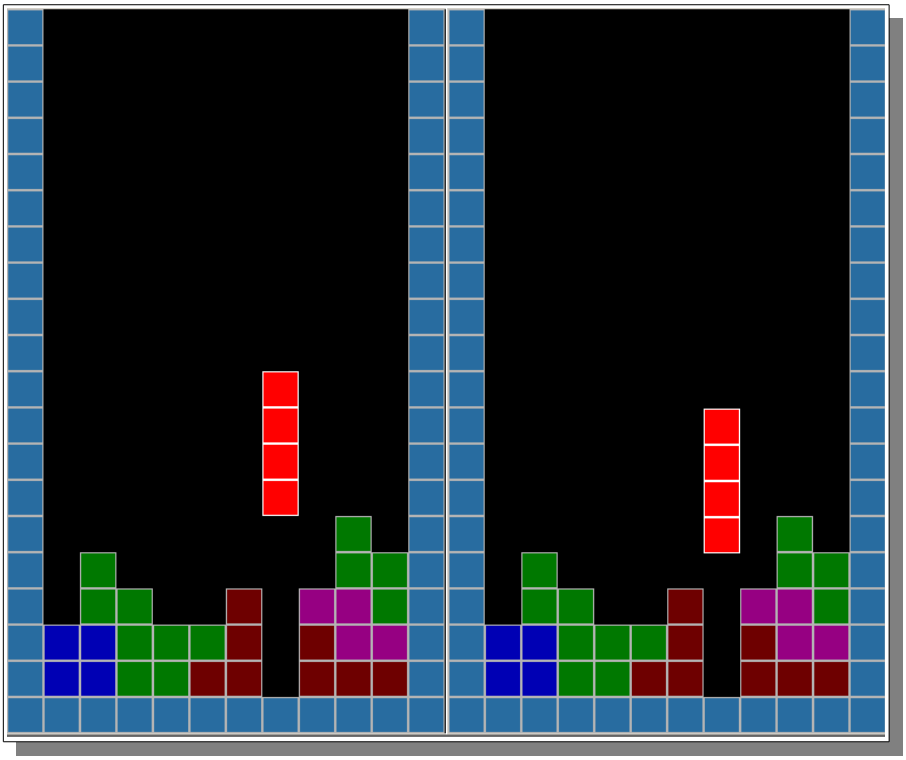
```
void Control::nuevaPartida(void){
```

```
        pActual = nuevaPieza();
        pActual->setTam(tam);
        pActual->nuevaPosicion(tam*4, 0);

        pSiguiente = nuevaPieza();
        pSiguiente->setTam(tam);
        pSiguiente->nuevaPosicion(tam*13,150);

        mapa->reiniciarMapa();

        nlineas = 0;
        nivel = 1;
        tiempo_bajarpieza = 700;

        pActual->activa = true;

        inicio = false;
        finjuego = false;
        pausa = false;
}
```

Una vez inicializadas los distintos componentes, se actualiza el estado del juego para que comience la partida. A partir de este momento el jugador tiene control sobre la pieza que se encuentra dentro del mapa.

La lógica se puede dividir en dos partes, por un lado los cálculos que se llevan a cabo mientras la pieza está activa y el jugador puede controlarla, y por el otro cuando la pieza se coloca sobre el mapa. Mientras el usuario tiene el control sobre la pieza, puede girarla y moverla para conseguir colocarla donde desea mientras esta cae. La pieza cae cuando se cumple un determinado tiempo que como se ha dicho antes se calcula en base al nivel, y no cae de manera progresiva si no a saltos, de una fila del mapa a la siguiente. Cuando este tiempo se cumple se comprueba si su posición en la siguiente fila es válido, si hay colisión se desactiva la pieza para que la segunda parte se encargue de ella, sino se baja la pieza.

El control de colisiones en este juego es muy sencillo por el movimiento a saltos de las piezas sobre el mapa. Según el movimiento que se quiera realizar se comprueban si las futuras posiciones de los subcuadrados de la pieza colisionan con alguno del mapa. En los giros se gira la pieza para la comprobación y si se genera colisión se deshace el giro.



Detalle del movimiento de las piezas y detección de colisiones.

```
//Devuelve verdadero si el giro de la pieza colisiona
void Control::colisionGiro2(void){

	//Giramos la pieza temporalmente para comprobar si colisiona
	if(giroder)  pActual->girarPiezaDer();

	if(giroizq) pActual->girarPiezaIzq();

	//var auxialiares
	int i,j,fila,columna;
	int x = pActual->getPosX();
	int y = pActual->getPosY();

	for(i=0; i<4; i++){
		for(j=0; j<4; j++){

			//comprobamos en la forma de la pieza si hay un 1
			if(pActual->hayUno(j,i)){

				//Traducimos el valor x,y del 1 de forma para comprobar en
el mapa
				fila = (x + i*tam) /tam;
				columna = (y + j*tam) /tam;

				//Comprobamos si algun 1 de la forma colisiona con mapa
				if(!mapa->posicionOcupada(fila, columna)){

					//Colisión, devolvemos la pieza a la postura con la
que entro
					if(giroder) pActual->girarPiezaIzq();
```

```
                                if(giroizq) pActual->girarPiezaDer();

                                if(giroder) giroder = false;
                                if(giroizq) giroizq = false;

                                return;
                        }
                }
        }
    }
    if(giroder) giroder = false;
    if(giroizq) giroizq = false;
}
```

Una vez la pieza ha sido colocada (ha colisionado con el fondo del mapa), la segunda parte del método consiste en actualizar el mapa con la nueva pieza que se ha colocado, comprobar si con la nueva pieza se ha completado alguna línea, en cuyo caso se contabiliza y se comprueba si se ha pasado de nivel (cada diez líneas). Además se libera la pieza que acaba de colocarse y se cambia por la que se mostraba como pieza siguiente, se introduce en el mapa y se comprueba si genera una colisión en cuyo caso se acaba la partida, sino se genera la que será la siguiente pieza.

```
void Control::actualizar(void){
//PRIMERA PARTE
    if(pActual->activa){

        //Controlamos si toca bajar la pieza
        if(tiempo_bajarpieza < SDL_GetTicks()){

            abajo = true;
        //Movemos la pieza hacia abajo un escalón sino colisiona con el mapa
            if(!colision()) pActual->moverPieza(0,tam);
            else pActual->activa = false;

            //Calculamos el tiempo hasta bajar la pieza de nuevo
            //según el nivel
            tiempo_bajarpieza = SDL_GetTicks() + (600 - nivel*100);
            //controlamos un máximo de tiempo de bajada, no mas de nivel 5
            if(nivel > 5) tiempo_bajarpieza = SDL_GetTicks() + 100;
        //si esta la tecla abajo presionada aumentamos la velocidad de caída
            if(keydown) tiempo_bajarpieza = SDL_GetTicks() + 30;
        }

        //moviendo y girando la pieza
        if(izq) if(!colision()) pActual->moverPieza(-tam, 0);
        if(der) if(!colision()) pActual->moverPieza(tam, 0);
        if(giroder) colisionGiro2();
        if(giroizq) colisionGiro2();

    }
//SEGUNDA PARTE
    else{

        //Actualiza el mapa, con la pieza que ha sido desactivada
        mapa->actualizar(pActual);
```

```
            //comprobamos si hay lineas completas, si hay incrementamos nlineas
            nlineas += mapa->limpiarLineas();

            //controlamos si el usuario pasa de nivel, nuevo nivel cada 10 lineas
            if(nlineas > (nivel*10)) nivel++;

            //liberar pActual antes de abandonarla y la actualizamos
            delete(pActual);
            pActual = pSiguiente;
            pActual->activa = true;
            pActual->nuevaPosicion(tam*4, 0);

            //si la nueva pieza colisiona en este momento la partida acaba
            abajo = true;
            if(colision()){
                  finjuego = true;
                  pausa = true;
                  return;
            }

            //Generamos la siguiente pieza
            pSiguiente = nuevaPieza();
            pSiguiente->setTam(tam);
            pSiguiente->nuevaPosicion(tam*13,150);
      }
}
```

4.3.2 Las piezas y el mapa

Estos son los dos elementos que componen este juego. Por un lado las piezas, cuyos función es implementar las formas y giros antes establecidos. Para ello cada pieza tiene como atributos las diferentes posturas que puede adoptar. Según se quiera girar a izquierda o derecha se calcula la postura y se copia sobre la matriz que representa a la pieza.

```
int Palo::postura0[4][4] = {     {0, 0, 0, 0},
                                 {1, 1, 1, 1},
                                 {0, 0, 0, 0},
                                 {0, 0, 0, 0} };

int Palo::postura1[4][4] = {     {0, 0, 1, 0},
                                 {0, 0, 1, 0},
                                 {0, 0, 1, 0},
                                 {0, 0, 1, 0} };
Palo::Palo(){

      posturaActual = 0;
      tipo = _PALO;
      nposturas = 2;
      tam=0;

      //color de la pieza
      r = 255;
      g = 0;
      b = 0;
```

```
	//postura inicial
	memcpy(forma, postura0, sizeof(postura0));
}

//Gira la pieza en sentido del reloj
void Palo::girarPiezaDer(void){

	posturaActual = (posturaActual + 1) % nposturas;

	if(posturaActual == 0) memcpy(forma, postura0, sizeof(postura0));


	if(posturaActual == 1) memcpy(forma, postura1, sizeof(postura1));
}
```

El mapa por su parte esta formado por una matriz de enteros de tamaño fijo. Cumple tareas sencillas como la de preparar el mapa para el comienzo de cada partida, responder sobre si una casilla está o no ocupada. Los métodos más representativos son *actualizar,* que introduce la pieza que se le pasa como parámetro dentro del mapa, y *limpiarLineas,* que comprueba y elimina líneas llenas.

```
void Mapa::actualizar(Pieza* p){
	int i,j,fila,columna;

	//Guardamos los "1´s" de la forma de la pieza en el mapa
	for(i=0; i<4; i++){
		for(j=0; j<4; j++){

			//comprobamos en la forma de la pieza si hay un 1
			if(p->hayUno(j,i)){
			//Traducimos el valor x,y del 1 de forma para guardarlo en el mapa
				columna = (p->getPosX() + i*tam) /tam;
				fila  = (p->getPosY() + j*tam) /tam;

//Guardamos el tipo como información en el mapa para mantener el color de la pieza
				zona[columna][fila] = p->getTipo();
			}
		}
	}
}

int Mapa::limpiarLineas(void){
	int nlineas = 0;
	int fila, columna, cont;

	for(fila = NFILAS-2; fila > 0; fila--){
	//reiniciamos el contador para comprobar si la fila esta llena
		cont = 0;
		for(columna = 1; columna < NCOLUMNAS-1; columna++){

			if(zona[columna][fila] > -1){

				cont++;
			}
```

```
            }
//comprobamos si esta llena, si lo esta se reorganiza el mapa y la contabilizamos
            if(cont == 10) {

                  int fil, col, auxcont;
                  //reorganizar el mapa tras detectar una linea llena
                  for(fil = fila; fil > 0; fil--){

                        auxcont = 0;
                        for(col = 1; col < NCOLUMNAS-1; col++){

                              if(zona[col][fil] > -1) auxcont++;
                              zona[col][fil] = zona[col][fil-1];
                        }
                        if(auxcont == 0) break;
                  }
                  //contabilizamos
                  nlineas++;
                  //ajustamos el contador de fila ya que hemos alterado el mapa
                  fila++;
            }
//si el contador no ha encontrado nada en la linea no hay nada por encima salimos
            if(cont == 0) return nlineas;
      }
      return nlineas;
}
```

4.3.3 El pintado

El apartado gráfico de este juego es muy sencillo, esta fue una de las razones para elegirlo como primer desarrollo. La representación de escena se hace mediante rectángulos de diferentes colores, tanto para las piezas como para el mapa. Esto se consigue a través de la función *SDL_FillRect* de la SDL que se ha comentado en el capítulo "Herramientas".

El pintado de cada pieza al ser algo común se implementa en la clase padre, cada pieza inicializa su propio color en el constructor, y aparte se dibuja un borde blanco al rededor de cada cuadrado que forme parte de la pieza. En el caso del mapa es prácticamente igual.

```
void Pieza::pintar(SDL_Surface* pantalla){

//Variables auxiliares, guarda las posiciones de los subcuadrados que forman la pieza
   SDL_Rect subcuadrado;


   for(int i=0; i<4; i++){
      for(int j=0; j<4; j++){

         if(forma[i][j] == 1 ){

// Inicializamos la variable con la posición de cada subcuadrado
            subcuadrado.x = j*tam + pos.x;
            subcuadrado.y = i*tam + pos.y;
            subcuadrado.w = tam;
            subcuadrado.h = tam;
```

```
//Dibujamos dos cuadrados uno encima de otro para crear un borde blanco
             SDL_FillRect(pantalla, &subcuadrado, SDL_MapRGB(pantalla->format, 255,
255, 255));
              subcuadrado.x += 1;
              subcuadrado.y += 1;
              subcuadrado.h -= 2;
              subcuadrado.w -= 2;
              SDL_FillRect(pantalla, &subcuadrado, SDL_MapRGB(pantalla->format,r,g,b));
          }
       }
   }
}
```

5. Segundo Desarrollo: Plataformas

En este capítulo se presenta las diferentes fases del segundo desarrollo del proyecto, una vez terminada la primera parte donde se estudiaban las herramientas y se realizaba el primer ejercicio, ya se tiene una pequeña experiencia con la SDL y se puede dedicar más tiempo a las mecánicas del juego. Este prototipo se centrará en desarrollar las principales características que se esperan de un juego tipo plataformas en 2D, del estilo de los conocidos Super Mario, Sonic o los anteriormente mencionados Super Meat Boy y Braid.

5.1 Análisis

A diferencia del desarrollo anterior, en el que se establecieron todos los requisitos de forma clara desde un principio, en esta parte se eligió un enfoque ligeramente diferente. Se partió con una idea principal concreta, desarrollar de un prototipo de juego estilo plataformas, pero a la vez muy generalista sobre la que se irían desarrollando funcionalidades. Se propuso hacer un desarrollo a través de prototipos, realizando reuniones a medida que el proyecto evolucionaba en las que se validaba el estado actual mediante pruebas en el laboratorio, y se discutían y planteaban nuevas funcionalidades a implementar.

La mayor ventaja de esta forma de trabajo es que se ven los avances día a día, se tiene una clara imagen de como va quedando el producto sin tener que esperar al final del desarrollo, evitando sorpresas inesperadas. De esta forma se puede asegurar que se va por el buen camino, por lo menos visualmente y en cuanto a funcionalidades.

Una de las desventajas de este proceso es que una nuevo requisito planteado en estas reuniones no encaje en el diseño actual, obligando a una refactorización del código ya escrito para darle cabida.

A continuación se presenta un resumen de las reuniones y los requisitos que se extrajeron de cada una, además de capturas del estado del proyecto en diferentes momentos.

Primera reunión 21/03/13, conclusiones alcanzadas:

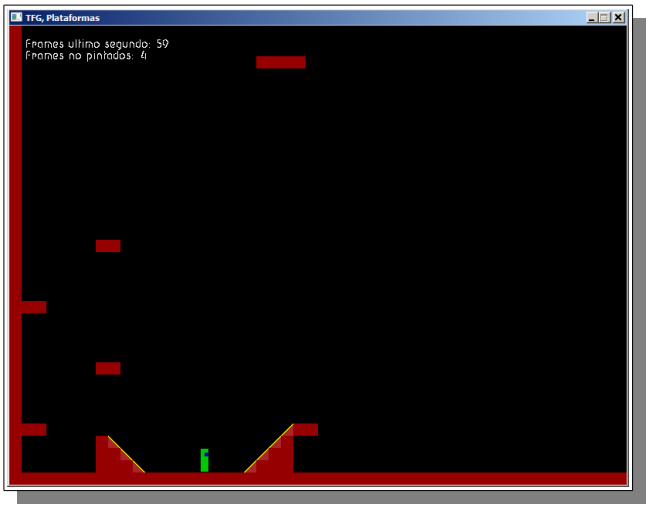
- Se propuso el desarrollo de un juego estilo plataformas, haciendo un breve repaso a las funcionalidades más comunes.
- Se planteó el diseño inicial que debería contemplar: personajes, mapas, colisiones y físicas.

Requisitos extraídos de la reunión:

- Realizar una aplicación ejecutable, para su posterior prueba.
- Aplicación fiable que finalice sin errores.
- Personaje principal, que se mueva sobre el mapa.
- Mapa a base de tiles (casillas).
- Colisiones del jugador con el mapa.
- Físicas del jugador.

- Representar la escena pintando los elementos del juego (mapa y personaje).

Reunión 04/04/13, captura del prototipo presentado,



En esta primera versión ya se tiene control sobre el personaje (rectángulo verde), se puede mover a izquierda, derecha y saltar. El planteamiento del mapa y las colisiones están casi terminadas, en esta versión existía un problema en los límites de las rampas. Se hace un primera implementación de las físicas.

La representación del los elementos que forman el juego es muy simple todavía, se utilizan rectángulos de colores para representar tanto al personaje como las diferentes casillas o tiles que representan el mapa.

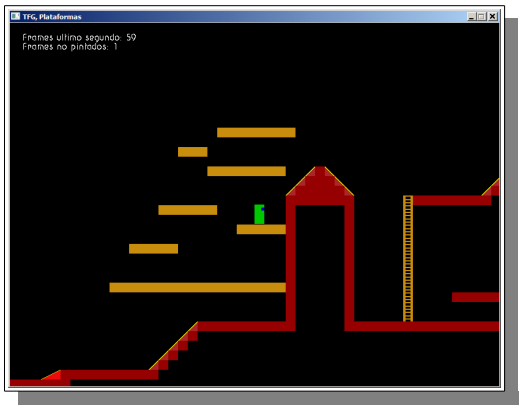
Una vez analizado el prototipo se propusieron los siguientes pasos a seguir;

- Resolver los problemas con las rampas en el mapa.
- Añadir la capacidad de disparar al jugador.
- Añadir enemigos y controlar las colisiones con el jugador y disparos.
- Mejorar la representación de la escena, añadiendo imágenes para mostrar los elementos; jugador, enemigos, mapa y fondo.
- Elegir editor mapas libre, estudiar formato salida, e implementar un lector.

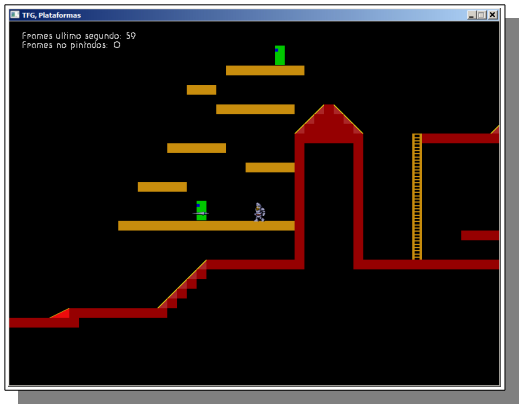
Nuevos requisitos extraídos de la reunión:

- El jugador debe disparar.
- Debe haber enemigos en el nivel.
- Colisiones entre; jugador, enemigos y disparos.
- Usar imágenes para la representación de la escena.
- Implementar un lector de mapas generados por un editor externo.

Etapas intermedias antes de la siguiente reunión;

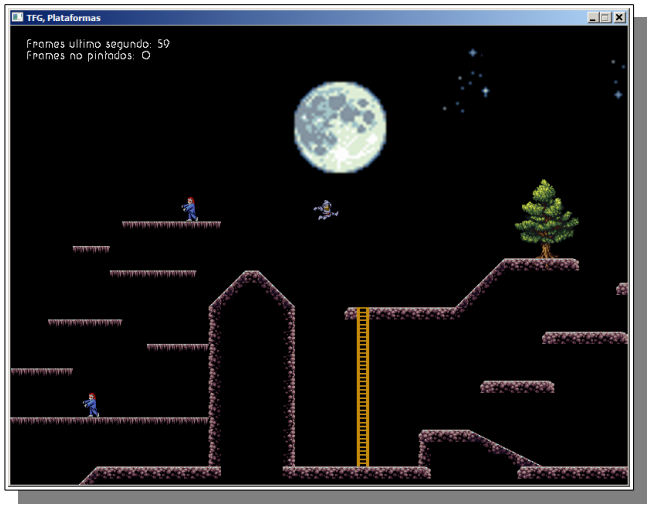


Implementado el lector de mapas, además de nuevas funcionalidades del mapa; escaleras y "one way plataforms"[45].



Añadida al jugador la capacidad de disparar, añadidos la capacidad de tener enemigos en los niveles, y detección de colisiones entre enemigos, disparos y jugador. Se implementa el uso de sprites para el jugador en sus diferentes estados (disparando, saltando, etc...).

Reunión 06/05/13, captura del prototipo presentado,



Se empiezan a usar imágenes para la representación de la escena, para el jugador, enemigos, para el mapa y el fondo.

Conclusiones tras analizar el prototipo;

- Añadir música de fondo y efectos (saltar, disparar, etc..).
- Añadir objetos que se puedan recoger.
- Añadir IA para los enemigos.
- Implementar un sistema de daño para el jugador.

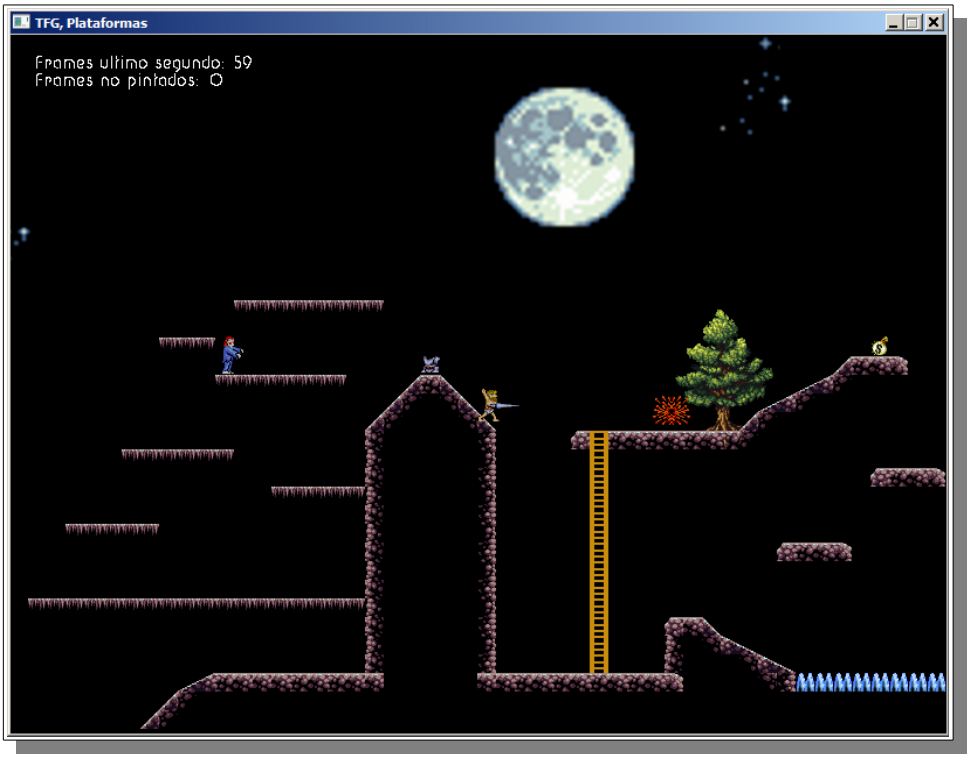
45 Se refiere a plataformas que estando por debajo de ellas no producen colisión pero si desde arriba. En la imagen son aquellas pintadas de marrón.

- Crear otro nivel e implementar la transición entre niveles.

Nuevos requisitos:

- Música fondo.
- Efectos sonido.
- Objetos para recoger al pasar por encima.
- IA para los enemigos.
- Reflejar el daño sufrido por el jugador.
- Niveles tiene que tener un final.
- Cuando se acaba un nivel se debe cargar el siguiente.

A continuación se presenta una captura del último prototipo presentado, y se comentan las nuevas funcionalidades.

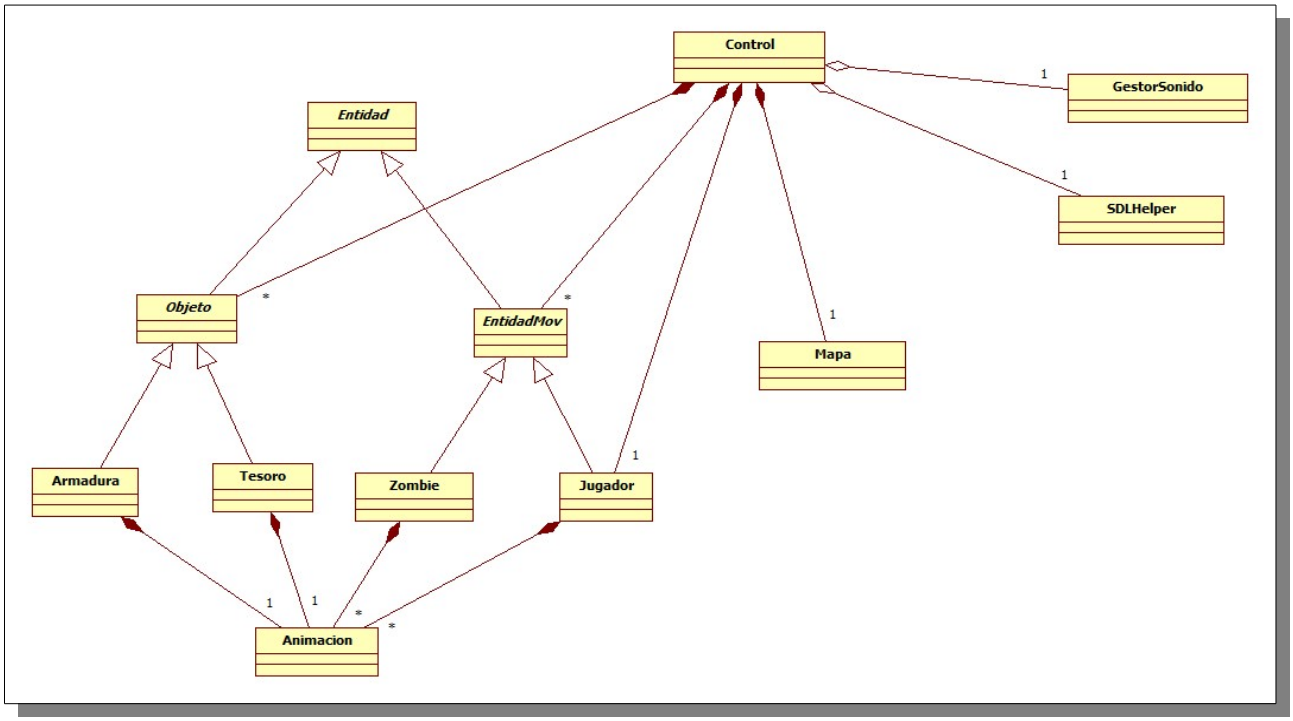


Captura donde se observan muchas de las características del prototipo, sprites, escaleras, pinchos, enemigos y objetos.

En esta versión ya está implementado la gestión del sonido, suena música de fondo al entrar a los niveles, hay efectos de sonido cuando se salta, se recoge un objeto o cuando una bala golpea a un enemigo. Añadidos al mapa la capacidad de albergar objetos (se implementaron: armadura y tesoro) que se pueden recoger por el jugador, inteligencia artificial para que los enemigos se movieran. Una casilla con pinchos, muy común en estos juegos, que daña al jugador, el cual ahora pierde la armadura con el primer impacto y muere con el segundo. Además se creó otro nivel para que se apreciara el cambio de uno a otro tras llegar a la salida.

5.2 Diseño

Durante el desarrollo de este juego el diseño de clases ha ido evolucionando para satisfacer las nuevas funcionalidades propuestas. A continuación se presenta la estructura utilizada en el último prototipo y se comentan brevemente las diferentes clases, sus métodos y atributos más importantes.



Clase Control

Esta clase es una de las más importantes, ya que se encarga de llevar a cabo la lógica del juego, contiene todos los elementos que componen el juego;

- El mapa sobre el que se cargan los niveles, de tipo *Mapa*.
- El jugador de tipo *Jugador*.
- Los enemigos del nivel en el que se encuentre el jugador, agrupados en un vector de tipo *EntidadMov* para facilitar su manejo.
- Los objetos, al igual que los enemigos, se agrupan en un vector de tipo *Objeto*, aprovechando que la lógica que los afecta es la misma

Control

-*helper: SDLHelper
-*sonidos: GestorSonido;
-mapa: Mapa
-jugador1: Jugador
-enemigos: vector<EntidadMov*>
-objetos: vector<Objeto*>
-entidadesVisibles: vector<Entidad*>
-scroll_x: int
-scroll_y: int
-limiteHorizontal: int
-limiteVertical: int

+Control(SDLHelper *helper)
+actualizarEstado(void)
+pintar(void)
~Control(void)
-cargarMapa(void)
-colisionesConMapa(EntidadMov *e)
-colisionesDinamicas(void)
-colisionesConEnemigos(void)
-colisionesConBalas(void)
-colisionEntreEntidades(Entidad *e1, Entidad *e2)
-colisionEntreRectangulos(int a1, int a2, int a3, int a4, int b1, int b2, int b3, int b4)
-entidadEnSuelo(EntidadMov *e): bool
-entidadEnEscalera(EntidadMov *e): bool
-entidadEnPinchos(EntidadMov *e): bool
-entidadEnSalida(EntidadMov *e): bool
-entidadVisible(Entidad *e): bool
-fisicas(EntidadMov *e)
-agruparEntidadesAPintar(void)
-actualizarScroll(EntidadMov *e)
-gestionarControles(void)
-iaZombie(EntidadMov *e)

independientemente de su tipo.

- Tiene acceso al sonido mediante la referencia al objeto de tipo `GestorSonido` y gracias a la otra referencia al objeto de tipo `SDLHelper` accede a la `SDL_Surface` principal en la que se ha establecido el modo de video y donde se pintará el juego.

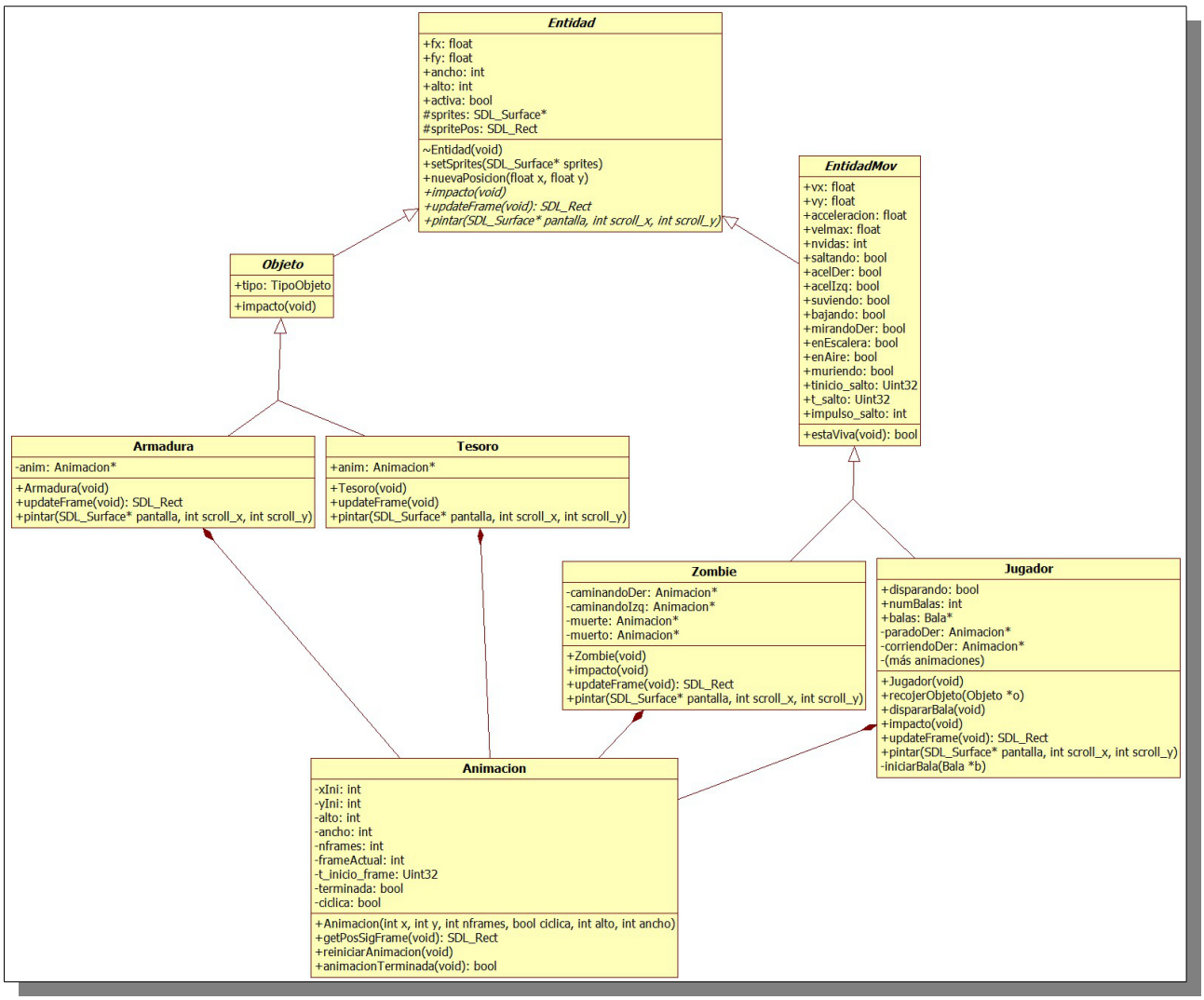
Su cometido principal es resolver las interacciones que se producen entre los elementos del juego y el mapa, en cada ciclo del bucle principal del juego cuando se llama al método `actualizarEstado()` de esta clase. Este método primero evalúa los controles, luego aplica las físicas oportunas y por último resuelve los diferentes tipos de colisiones que se pueden dar.

Esta clase es responsable también de llamar a las funciones de pintado de los elementos del juego, cumpliendo la importante regla de pintar solamente lo que es visible en cada momento, por ejemplo pintando sólo los enemigos que estén en la parte visible de la pantalla, en el apartado de implementación se explica como se lleva a cabo para cada elemento.

En esta clase también se ha implementado una función que mueva a los enemigos a modo de inteligencia artificial. No es un objetivo de este proyecto este campo, pero tampoco se quería dejar a los enemigos parados, por lo que se implementó algo sencillo que les permite moverse detectando cuando llegan a un precipicio o chocan contra una pared. También se ha de mencionar que esta función debería formar parte de una clase aparte que implementara diferentes inteligencias para varios enemigos, en un principio se dejó así para poder dedicar más tiempo a otras tareas y dejar esta refactorización para el final.

Clase Entidad

En la siguiente imagen se observa la jerarquía de clases usada para representar las entidades (jugador, enemigos y objetos) que forman parte del juego y así simplificar el manejo de estos componentes.



La clase abstracta *Entidad* sirve de base sobre la que se construirán el resto de elementos del juego, establece los atributos y métodos comunes;

Atributos:

- Posición en 2D, *fx* e *fy*.
- Tamaño en 2D, *ancho* y *alto*.
- Control sobre la visibilidad, *activa*.
- Referencia a la imagen que contiene los sprites de esa entidad, para su pintado, *sprites*.
- Información del sprite que hay que pintar, posición dentro de la imagen que los contiene a todos y tamaño, *spritePos*.

Métodos:

- *pintar(SDL_Surface* pantalla, int scroll_x, int scroll_y)* pinta sobre pantalla a la entidad, teniendo en cuenta los valores

del scroll para ajustar su posición. Este es un método abstracto a implementar por las clases herederas.

- *setSprites(SDL_Surface* imagen)* se utiliza para establecer la imagen que contiene todos los sprites usados por la entidad.
- *updateFrame(void)* actualiza *spritePos* según el estado de la entidad.

En el siguiente escalón nos encontramos con dos pequeñas clases, *Objeto* y *EntidadMov* también abstractas. La primera se usará para representar aquellos objetos con los que el jugador puede interactuar y la segunda para aquellas entidades que se puedan mover, en este caso el jugador y los enemigos.

- **Objeto y derivadas**

Como se ha comentado la clase *Objeto* también es abstracta, está pensada para representar cualquier objeto con el que el jugador pueda interactuar. La clase añade un atributo que especifica que tipo de objeto es, e implementa la función *impacto().* De esta clase derivan las clases Tesoro y Armadura, que son los dos objetos que se han implementado durante el desarrollo de esta parte del proyecto. Además se sirven de la clase *Animacion* para su pintado, sus principales características son;

- Tesoro: implementa las funciones abstractas heredadas, updateFrame y pintar. Representado por la imagen de un saco con el signo del dólar, se recoge pasando por encima. La función es aumentar la puntuación del jugador, parte que no se llego a implementar.
- Armadura: igual que el anterior implementa las funciones abstractas heredadas. Cuando el jugador se encuentra sin armadura, al recoger este objeto el jugador pasará a tener armadura, lo que supone aguantar dos golpes antes de morir. Se representa por la imagen de una armadura.

Es una implementación simple que cumple su propósito y que puede extenderse fácilmente, dejando abierta la creación de nuevos objetos.

- **EntidadMov y derivadas**

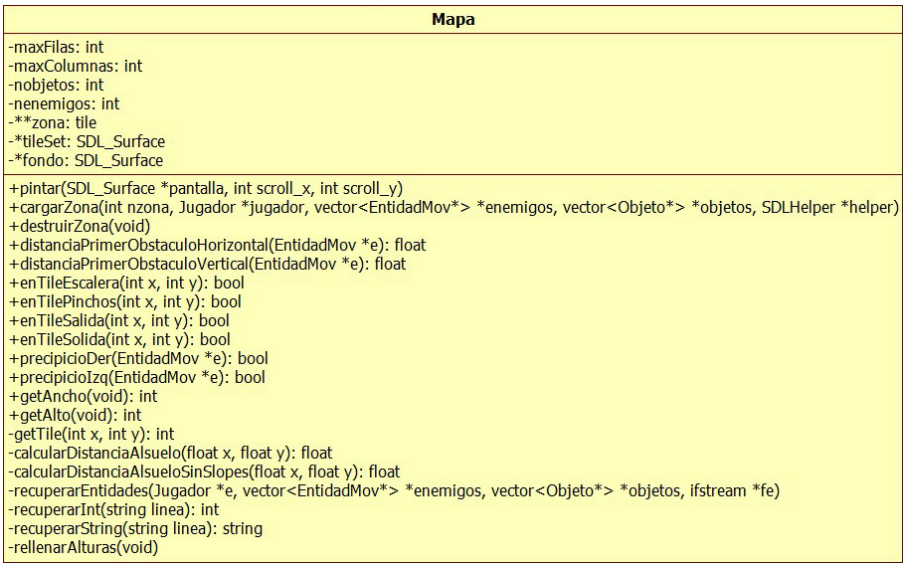
La clase *EntidadMov* añade a la clase *Entidad* los atributos necesarios para que se pueda mover (*vx*, *vy*, *acceleracion*, *velmax*), una serie de variables booleanas que representan diferentes estados en los que se puede encontrar (*saltando, acelDer, acelIzq, subiendo, bajando, mirandoDer, enEscalera, enAire, muriendo*), el número de vidas que tiene (*nvidas*) y un grupo de variables para controlar el salto (*tinicio_salto*, *t_salto*, *impulso_salto*).

- Jugador: esta clase como su nombre indica es la que se usará para representar al jugador dentro de la aplicación. Sobre las funcionalidades de Entidadmov añade la capacidad de disparar balas, y de recoger objetos, que son exclusivas de esta clase. Las balas están almacenadas en un vector y son definidas por una estructura sencilla que contiene información sobre su velocidad en x, posición, tamaño y un control booleano para saber si esta viva. La clase cuenta además con varias animaciones para representar la entidad en diferentes estados.
- Zombie: esta clase implementa un tipo de enemigo, al igual que las clases anteriores utiliza

varias animaciones para su pintado, y aunque deriva de Entidadmov al igual que Jugador, no se ha implementado la capacidad de saltar o subir escaleras.

Clase Mapa

Es un componente fundamental del juego, se representa a través de una matriz de tiles(casillas) cada una guardando información sobre su tipo y un mapa de alturas(vector que guarda la altura de cada pixel en el eje x de esa casilla, en el apartado de implementación se explica como y porque). Sus principales funciones se pueden dividir en tres bloques;

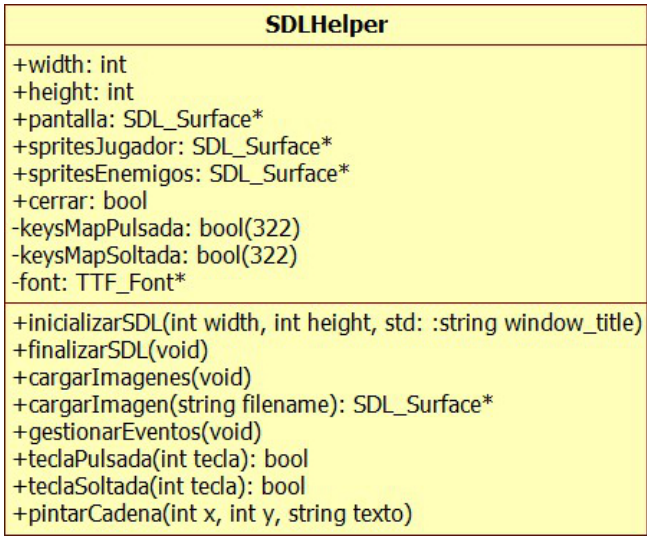


- Cargar una zona[46] creada con el editor seleccionado y cumpliendo con el formato establecido. Para ello se llama al método `cargarZona`, que recupera toda la información desde el archivo especifico de cada zona. Se crean los enemigos y objetos de la zona, se ubica al jugador y se crea diatónicamente la matriz que albergara la información de las tiles o casillas.
- Pintar el mapa de manera adecuada, dibujando solamente aquella parte que es visible, usando imágenes para representar cada casilla y el fondo. Hacer scroll vertical y horizontal de manera fluida, manteniendo al jugador en el centro de la pantalla.
- Procurar una serie de métodos para controlar las colisiones de una `Entidadmov` con las diferentes tiles o casillas del mapa. Esto se consigue a través de las funciones que calculan la distancia desde la entidad hasta al primer obstáculo del mapa.

Clase SDLHelper

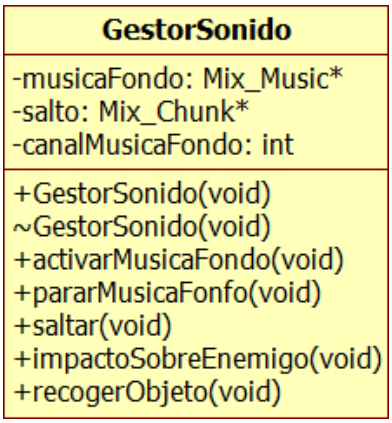
Esta es una clase de apoyo, la idea era que se encargara de todas las funciones relacionadas con la SDL, intentando así que el código del juego estuviese desligado de la SDL. Sus funciones principales son iniciar la SDL, finalizarla, gestionar los eventos, pintar cadenas de texto a través de una librería de apoyo, SDL_ttf, y cargar imágenes.

46 Zona se refiere a la información que se carga en el mapa. Zona se utiliza como sinónimo de nivel, el personaje supera el nivel o la zona no el mapa, y cuando esto pasa se actualiza la información del mapa con la nueva zona.



Clase GestorSonido

Esta es una clase muy sencilla que se encarga de gestionar todos los sonidos de la aplicación, para ello inicializa la librería de audio (SDL_Mixer), carga los sonidos y procura métodos para poder reproducirlos.



5.3 Implementación

A continuación se realizará un estudio detallado de las principales características desarrolladas y su implementación, la finalidad de este punto no es la de comentar el código de las clases, si no profundizar en los aspectos mas relevantes del desarrollo; el movimiento del jugador y sus habilidades (disparar y saltar), todos los aspectos relacionados con el mapa, tanto su implementación como la creación a través del editor Tiled y su posterior carga dentro del juego, el sistema de colisiones utilizado y la gestión de los objetos y enemigos.

En cada punto de los anteriores se analizara como se lleva a cabo el pintado y las imágenes utilizadas, como se crearon o de donde se obtuvieron. Por último se analizará la implementación de otros elementos adicionales de relevancia, como son la gestión del sonido y controles.

5.3.1 Jugador

Durante la implementación de esta clase se han tendiendo en mente tres objetivos principales:

1. Conseguir un desplazamiento fluido del jugador en pantalla.
2. Desarrollar las habilidades.
3. Animar los movimientos de forma correcta.

- Movimiento del jugador y habilidades.

Este es un pilar básicos en el desarrollo de videojuegos, ya que es la principal forma que tiene el usuario de interactuar con nuestra aplicación, por muy buena que sea la idea del juego, o los millones que se gasten en desarrollo, sin no se consigue un buen control sobre el personaje el juego estará abocado al fracaso.

Para conseguir un buen control tenemos que tener en cuenta varios factores, por un lado deben ser inmediatos, cuando pulsamos el botón de saltar el personaje debe saltar en ese instante, es importante que el personaje refleje lo que el usuario quiere hacer en cada momento, de esta forma el usuario tiene la sensación de tener un completo control sobre la aplicación. La precisión de los controles es otro factor a tener en cuenta, tener la posibilidad de movernos lo justo para colocarnos para el siguiente salto o disparar el aire justo cuando se tiene al enemigo a tiro, esto se puede conseguir por ejemplo diferenciando cuando el usuario mantiene pulsada la tecla o botón de la acción o cuando hace cortas pulsaciones.

Otro detalle importante es la complejidad de los controles, un sistema con muchas posibilidades de por si no es mejor. En algunos casos, como en simuladores de aviones que nos encontramos teniendo que manejar muchísimas teclas para poder manejar nuestro vehículo, no existe alternativa, pero en un juego de corte mas comercial el número de controles debe ser reducido, ajustándose a lo que el usuario espera encontrarse.

Explicado lo anterior se entiende que una de las grandes claves del éxito de la saga de Mario haya sido su cuidado sistema de control, que se centra en el dominio absoluto del personaje por parte del usuario y la improvisación más primitiva de nuestro cerebro al tener que reaccionar

instintivamente ante las nuevas amenazas. No es casualidad que hasta ahora jamás se haya alterado su control simplificado a pesar de que otros juegos tienen sistemas muchos más complejos.

Tan importante son los controles del juego y que el usuario los domine, que en los actualidad se ha potenciado el dedicarle los primeros minutos del juego a que el usuario se familiarice con los controles en lo que se suele llamar "Tutorial", que suele ser un nivel en el que se presentan los obstáculos básicos del juego y se hace un repaso a todos los controles y mecánicas.

- Entrada de teclado.

La entrada de controles se lleva a cabo a través del método *gestionarControles()* de la clase *Control*, este se ejecuta al principio del método *actualizarEstado()*, de la misma clase, y verifica si alguna tecla que corresponda a una acción se ha pulsado. El trabajo sucio lo lleva a cabo la clase *SDLHelper* que registra los eventos del teclado, cuando se pulsa o se suelta una tecla, en vectores de booleanos para que se pueda acceder a ellos en cualquier punto de la aplicación.

```
// 322 es el número de eventos de teclado en SDL
bool keysMapPulsada[322];
bool keysMapSoltada[322];
```

```
void SDLHelper::gestionarEventos(void){

	SDL_Event event;

	// bucle hasta que no queden eventos en la lista
	while (SDL_PollEvent(&event)){

		switch (event.type)// Procesando según el tipo

		//Cerrar aplicación
		case SDL_QUIT:

			cerrar = true;
			break;

		case SDL_KEYDOWN:

			//Apuntamos la tecla que se acaba de pulsar
			keysMapPulsada[event.key.keysym.sym] = true;
			//cout << "Tecla pulsada: " << event.key.keysym.sym << endl;
			break;

		case SDL_KEYUP:

			//Eliminamos la tecla pulsada
			keysMapSoltada[event.key.keysym.sym] = true;
			//cout << "Tecla levantada: " << event.key.keysym.sym << endl;
			break;

		default:

			//cout << "evento no tratado" << endl;
```

```
                    break;
                }
        }
}
```

El acceso a esos vectores se realiza a través de los métodos siguientes, siempre que se tenga acceso al objeto de la clase *SDLHelper*.

```
bool SDLHelper::teclaPulsada(int tecla){

        if(keysMapPulsada[tecla]){

                keysMapPulsada[tecla] = false;
                return true;
        }
        else return false;
}

bool SDLHelper::teclaSoltada(int tecla){

        if(keysMapSoltada[tecla]){

                keysMapSoltada[tecla] = false;
                return true;
        }
        else return false;
}
```

Como se comento antes, dos factores importantes sobre los controles son la inmediatez y la precisión. El primero se consigue gracias a la SDL y su sistema de eventos, a continuación se comentaran diferentes aspectos que envuelven a la precisión y se comentaran unas consideraciones generales sobre diferentes formas que se usan en los videojuegos de este tipo para implementar el movimiento del jugador.

- Consideraciones Generales

Aceleración

Uno de los factores importantes que afectan a los juegos de tipo plataformas es la aceleración del personaje. La aceleración es el ratio de cambio en la velocidad, cuando es pequeña el personaje tarda mucho tiempo en llegar a su velocidad máxima, o para pararse del todo una vez que el usuario a soltado los controles. Esto hace que tengamos la sensación de que el personaje resbala, lo que dificulta su control. Este tipo de control esta más asociado a la saga Super Mario.

Cuando la aceleración es alta, el personaje tarda muy poco tiempo (o nada de tiempo) para ir desde cero a su máxima velocidad y viceversa, consiguiendo así unos controles de respuesta muy rápida, común en la saga Mega Man.

Super Mario World (aceleración baja), Super Metroid (aceleración media), Mega Man 7 (aceleración alta).

Para este desarrollo se eligió la primera opción comentada, una aceleración baja, para introducir un punto de dificultad en el control del personaje sin salirnos de la que el usuario podría esperarse.

```
if(e->acelDer){

	//Control de parada
	if(e->vx < 0.0) e->vx += e->aceleracion*FACTOR_PARADA;

	else if(e->vx < e->velmax) e->vx += e->aceleracion;
	else e->vx = e->velmax;
}

if(e->acelIzq){

	if(e->vx > 0.0) e->vx -= e->aceleracion*FACTOR_PARADA;

	else if(e->vx > -e->velmax) e->vx -= e->aceleracion;
	else e->vx = -e->velmax;
}
```

Si no se está apretando la tecla para moverse a la izquierda o a la derecha, entra en juego la fricción, siendo su valor es el mismo que el de la aceleración. En cualquier momento en el que no haya intención del usuario por moverse a izquierda o derecha, se le va substrayendo aceleración a la velocidad de la entidad, hasta dejarla a cero.

```
if(!e->acelDer && !e->acelIzq){

	if(e->vx > 0.0){

		e->vx -= e->aceleracion;
		if(e->vx < e->aceleracion || e->vx < 0.0) e->vx = 0.0;
```

```
        }

        if(e->vx < 0.0){

                e->vx += e->aceleracion;
                if(e->vx > e->aceleracion || e->vx > 0.0) e->vx = 0.0;
        }
}
```

Es importante integrar la aceleración a la velocidad antes de mover el personaje en el mapa, de lo contrario estaríamos introduciéndolo un frame más tarde, creando la sensación de retardo en los controles. Es una buena idea también, cuando el personaje choca contra un obstáculo poner a cero la velocidad en ese eje.

Queda aclarar como se ha implementado la secuencia de métodos para llevar a cabo lo recién comentado. En cada ciclo del bucle principal, *`actualizarEstado()`*, se analizan los controles *`gestionarControles()`* para luego en *`fisicas()`* realizar las operaciones relacionadas con la aceleración, fricción, gravedad y lógica de salto.

Control de salto

Super Mario Bros.

Saltar en un juego de plataformas puede ser tan sencillo como comprobar si el jugador esta en el suelo (o a veces, si ha estado en el suelo en los últimos n frames), y si se confirma, dar al personaje una velocidad inicial negativa en el eje y (en términos físicos, un impulso) y dejando que la gravedad haga el resto.

A continuación se presentan diferentes factores a tener en cuenta para la implementación del del salto:

1. Impulso: visto en juegos como *Super Mario World* y *Sonic the Hedgehog*, el salto conserva

la velocidad que el jugador tenía antes del salto. En algunos juegos esta es la única manera de afectar el arco del salto, como en la vida real.

2. Aceleración aérea: se refiere a cuando tenemos control sobre el movimiento horizontal cuando estamos en el aire. Aunque esto es físicamente imposible, es una características muy popular, haciendo más controlable al jugador. La mayoría de los juegos tipo plataformas cuentan con esta característica, salvo excepciones como por ejemplo *Prince of Persia*.
3. Control del ascenso: esta es otra característica muy popular, es importante porque al igual que la anterior afecta de manera positiva al control de jugador por parte del usuario. Se trata de que cuanto más tiempo mantengamos pulsado la tecla de salto, más saltara el personaje. Normalmente se consigue sumando impulso mientras se mantiene la tecla apretada, o suprimiendo la gravedad. Se ha de establecer un limite de tiempo para controlar el final del salto.
4. Múltiples saltos: una vez en el aire, algunos juegos permiten al usuario saltar de nuevo, normalmente por un número limitado de veces (el doble salto es lo más común). Esto se puede conseguir manteniendo un contador que se incrementa con cada salto y que se reinicia cuando se esta en el suelo.

Para la implementación de esta funcionalidad en el proyecto, se tomaron en cuenta todos los puntos anteriormente mencionados menos el último, múltiples saltos. De esta forma el usuario tiene mucho más control sobre el personaje, aunque esto no signifique que sea más fácil de controlar. Estos factores unidos a los comentados para la aceleración, hacen necesario un tiempo de adaptación al manejo del personaje.

La lógica del salto se lleva a cabo en el método *`fisicas()`*, donde se controla el impulso que se añade y el tiempo del salto,

```
...
if(e->saltando){

	e->vy = -e->impulso_salto;
	if((SDL_GetTicks() - e->tinicio_salto) > e->t_salto)
		e->saltando = false;
}
...
```

y en *`gestionarControles()`* donde se verifica que cumple las condiciones para saltar, en este caso estar en suelo y no estar montado en una escalera.

```
...
if(helper->teclaPulsada(SDLK_SPACE)){

	if(entidadEnSuelo(&jugador1) && !jugador1.enEscalera){

		jugador1.saltando = true;
		jugador1.tinicio_salto = SDL_GetTicks();
		sonidos->saltar();
	}
}
if(helper->teclaSoltada(SDLK_SPACE)) jugador1.saltando = false;
...
```

Hay que tener en cuenta que aunque no se comente nada de los enemigos, todas las características comentadas son generalidades de la clase `Entidadmov`, por lo que se aplican tanto a los enemigos como el jugador.

Disparar balas

Esta si es una característica propia del jugador, no se llegó a implementar ningún enemigo que disparase. Esta habilidad es la única manera que tiene el jugador de deshacerse de los enemigos que se encuentre. En casi todos los videojuegos suele existir alguna forma de eliminar enemigos, no siempre se hace a través de un sistema de disparos, como por ejemplo en la saga *Sonic the Hedgehog* en el que hay que caerles encima para matarlos. En muchas ocasiones coexisten varios sistemas como en la mencionada saga *Super Mario* en la que se puede disparar y a algunos enemigos caerles encima.

Sonic the Hedgehog 2

Esta habilidad se implementa en la clase `Jugador` mediante un vector del tipo `Bala` que las contiene.

```
...
struct Bala{

	float fx, fy, vx;
	int alto, ancho;
	bool viva;
};
...
```

Cuando se detecta en los controles que el usuario quiere disparar, se llama al método `disparar Bala()` que comprueba dos cosas:

- si hay alguna bala disponible, ya que se ha limitado el número que puede haber activas a la vez para aumentar la dificultad del juego. Aunque Se deja abierta la posibilidad de aumentar la cantidad de balas activas mediante algún objeto.
- Si el jugador se encuentra en un estado en el que se esta permitido disparar.

```
void Jugador::dispararBala(){

	//estados en los que no se puede disparar
	if(enEscalera || muriendo || !activa) return;

	for(int i=0; i < numBalas; i++){

		//activar bala
		if(!balas[i].viva){

			iniciarBala(&balas[i]);
			disparando = true;
			return;
		}
	}
}

void Jugador::iniciarBala(Bala *b){

	b->fx = fx;
	b->fy = fy + ALTO_SPRITE/2;
	b->vx = mirandoDer ? VEL_BALA : -VEL_BALA;
	b->viva = true;
}
```

- El pintado y la clase *Animacion*

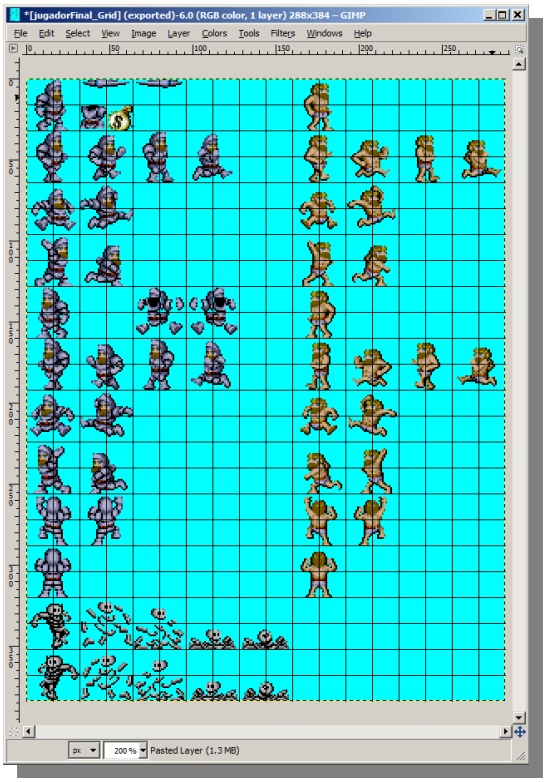
Podemos definir animación como una simulación de movimiento mediante una secuencia de imágenes. Al mostrar estas imágenes (llamadas cuadros o frames) sucesivamente en una misma posición producen la ilusión de movimiento que en realidad no existió.

En este principio se basan el cine o la televisión. Muestran imágenes una detrás de otra con pequeñas variaciones mediante las cuales (por el fenómeno phi[47]) nuestro cerebro construye un movimiento rellenando los huecos entre una imagen y la siguiente. Junto con la persistencia de la retina son la base de la teoría de la representación de movimiento en cine, ordenador o televisión.

En los videojuegos la animación de los personajes es conocida como Sprites (en juegos 2d) , estas son las secuencias de movimientos que va a realizar el personaje. La creación de sprites es un arte en si mismo, conocido como “Pixel Art”, y es un trabajo más de diseñador gráfico que de programador.

47 http://es.wikipedia.org/wiki/Fen%C3%B3meno_phi

Para resolver este problema se desarrolló la clase `Animacion`, la cual controla que imagen en concreto, del grupo de imágenes que forman la secuencia de un movimiento, toca pintar. Los sprites se suelen agrupar en "sprite sheets", que es una imagen que contiene todos los sprites asociados a un mismo tema. La siguiente imagen es la utilizada por el personaje de nuestro juego estilo plataformas.



Existen a primera vista varios factores a comentar. Lo primero que salta a la vista es el color de fondo, conocido como croma, inserción croma o llave de color (del inglés chroma key, o color

key), es una técnica audiovisual utilizada ampliamente tanto en cine y televisión como en fotografía, que consiste en extraer un color de la imagen (usualmente el verde o el azul) y reemplazar el área que ocupaba ese color por otra imagen, con la ayuda de un equipo especializado o un ordenador. Esto se hace cuando es demasiado costoso o inviable rodar al personaje en el escenario deseado, o para evitar el laborioso recorte del personaje fotograma a fotograma.

En nuestro caso, como se ha comentado antes la SDL por si sola, sin usar bibliotecas externas como SDL_Image, sólo trabaja con archivos del tipo Windows BMP, más que suficiente para nuestro desarrollo. El problema de este formato es que no trabaja con transparencias, por eso la necesidad de trabajar con el color key, otros formatos como .PNG si ofrecen un canal alpha para las partes transparentes de la imagen. Hay que tener en cuenta que ningún elemento del sprite debe ser del mismo color del fondo (pues sería eliminado).

También se observa en la imagen una cuadrícula que nos permite detectar detalles importantes; todas las imágenes que representan al personaje son del mismo tamaño, están organizadas por acciones en diferentes filas y estas acciones en diferentes frames (la cuadrícula que se ve es sólo para que se note lo comentado, la imagen que se usara en el juego no lleva cuadrícula). Además como la SDL no cuenta con funciones internas para rotar imágenes se decidió por introducirlas las dos versiones necesarias.

Se ha diferenciado entre tres tipos de animaciones, cíclicas o no y aquellas que sólo tienen un frame. Con cíclicas se refiere a aquellas animaciones en las que una vez llegados al último frame de la secuencia, se vuelve a comenzar desde el principio, como por ejemplo la animación de correr. Las no cíclicas son aquellas que se ejecutan una vez y no se repiten, en este caso se encuentran las animaciones para disparar y la de muerte. El tercer caso son aquellas que sólo cuentan con un frame, aunque técnicamente no es una animación, junto a los demás casos se consigue un buen acabado.

La clase no trabaja directamente con la imagen que contiene a los sprites, se apoya en la organización de esta, para en base a; el tamaño, número de frames, tipo de animación y la posición del primer frame, calcular las coordenadas de los frames que componen la secuencia. Es la clase que contiene las animaciones la que conoce que imagen usar.

```
...
corriendoDer = new Animacion(0, 32, 4, true, ALTO_SPRITE, ANCHO_SPRITE);
...
```

```
Animacion::Animacion(int x, int y, int nframes, bool ciclica, int alto, int ancho){

	this->nframes = nframes;
	this->ciclica = ciclica;
	this->alto = alto;
	this->ancho = ancho;
	xIni = x;
	yIni = y;

	frameActual = 0;
	terminada = false;
	t_inicio_frame = 0;
}
```

El método principal de la clase es *getPosSigFrame()*, que devuelve un tipo *SDL_Rect* que contiene la información del frame que tocar pintar. Además de controlar el tiempo de cada frame, que se ha establecido en 85 milisegundos, número al que se llegó tras varias pruebas.

```
SDL_Rect Animacion::getPosSigFrame(){

	SDL_Rect pos;

//Control del tiempo de cada frame de la animación
	if(SDL_GetTicks() > (t_inicio_frame + TIEMPO_FRAME)){

		if(t_inicio_frame == 0) t_inicio_frame = SDL_GetTicks();
		else{

			t_inicio_frame = SDL_GetTicks();
			frameActual = frameActual+1;
		}
	}

//Controlamos los diferentes tipos de animaciones: cíclica, playandstop y estática

	if((frameActual == nframes) && ciclica) frameActual = 0;

	if((frameActual == nframes) && !ciclica){

		frameActual--;
		terminada = true;
	}

	if(nframes == 1) frameActual = 0;

	pos.x = xIni + (frameActual * ancho);
	pos.y = yIni;
	pos.w = ancho;
	pos.h = alto;

	return pos;
}
```

En el caso de la clase Jugador en la que se hace uso de varias animaciones, es necesario implementar una lógica para su gestión. Dependiendo del estado en el que se encuentre el personaje se usará una u otra animación. Además hay que tener un control especial para reiniciar las animaciones en el momento adecuado, para su correcta representación. El método *updateframe()* es el encargado de controlar que animación toca según el estado.

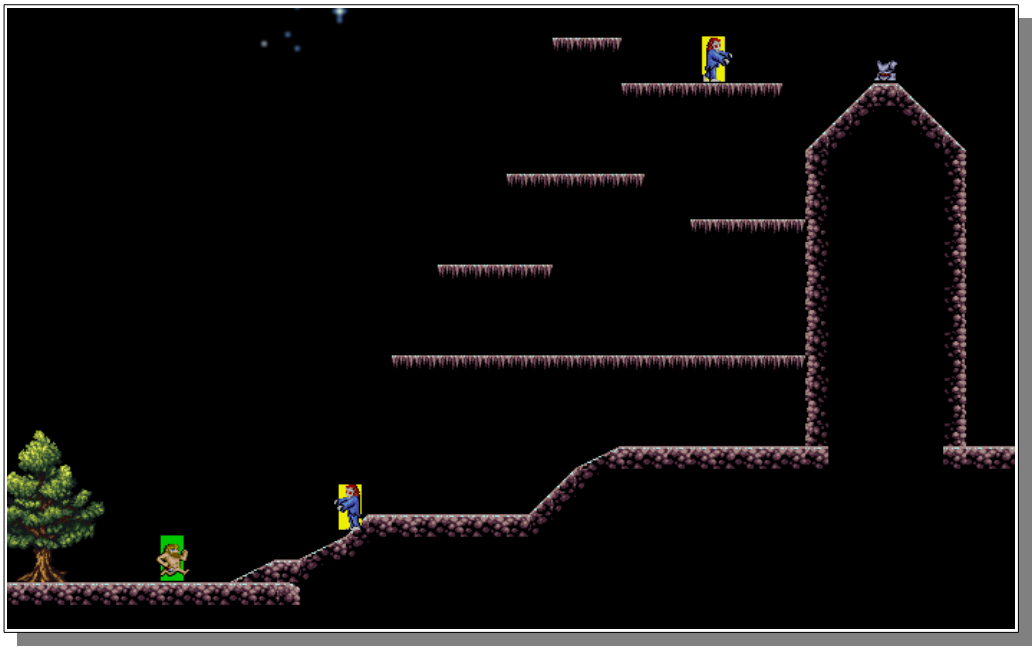
```
SDL_Rect Jugador::updateFrame(){

	//Muerto
	if(!activa && mirandoDer) return muertoDer->getPosSigFrame();
	if(!activa && !mirandoDer) return muertoIzq->getPosSigFrame();

	//Reiniciado animaciones de correr
	if(vx == 0){
```

```
            if(!corriendoIzq->animacionTerminada())corriendoIzq->reiniciarAnimacion();
            if(!corriendoDer->animacionTerminada())corriendoDer->reiniciarAnimacion();
      }

      //Muriendo
      if(mirandoDer && muriendo){

            if(muriendoDer->animacionTerminada()){

                  muriendoDer->reiniciarAnimacion();
                  muriendo = false;
                  activa = false;
                  return muertoDer->getPosSigFrame();
            }
            else return muriendoDer->getPosSigFrame();
      }
...
}
```

Por último el método pintar, donde todo lo comentado hasta ahora se une para representar en pantalla a una entidad concreta. Por un lado se ajusta la posición en pantalla donde se va a pintar la entidad, teniendo en cuenta el scroll que se tenga y se calcula la posición del sprite a pintar. Por último se pinta en la superficie principal la imagen recortada en la posición ajustada.

```
void Zombie::pintar(SDL_Surface* pantalla, int scroll_x, int scroll_y){

//Ajustamos la posición teniendo en cuenta el scroll
    SDL_Rect posAjustada;

    posAjustada.x = int(fx) - scroll_x;
    posAjustada.y = int(fy) - scroll_y;
    posAjustada.w = ancho;
    posAjustada.h = alto;

    spritePos = updateFrame();

    //Hitbox
    //SDL_FillRect(pantalla, &posAjustada, SDL_MapRGB(pantalla->format, 238, 238, 0));

    //Ajustando la imagen sobre el rectángulo
    posAjustada.x -= ((ANCHO_SPRITE - ancho)/2)-1;
    posAjustada.y -= ((ALTO_SPRITE - alto)/2);

    SDL_BlitSurface(sprites, &spritePos, pantalla, &posAjustada);
}
```

Para la comprobación visual de colisiones se usó la linea que aparece comentada, con título “Hitbox”, este concepto se refiere al rectángulo que representa el área de colisión de la entidad. En la siguiente captura se pueden observar tanto el del personaje como la de los enemigos.



Captura del proyecto mostrando los hitboxes de enemigos y jugador.

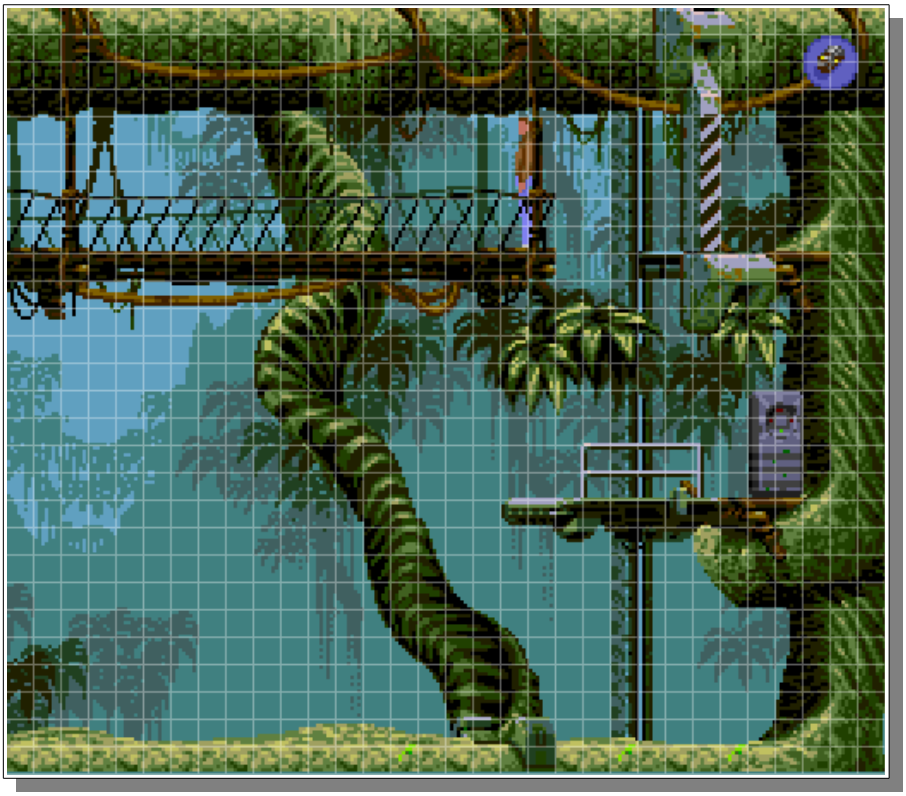
5.3.2 Mapa

Es un elemento fundamental y clave en el desarrollo de videojuegos, representa el el entorno por el que el personaje podrá moverse e interactuar. Su implementación varía mucho según el tipo de juego que vayamos a desarrollar, pero de una forma o de otra la mayoría de juegos cuentan con una estructura de este tipo. A continuación se realiza un pequeño estudio sobre los diferentes tipos de implementación de esta estructura que nos podemos encontrar en videojuegos tipo plataformas en dos dimensiones.

1. Basado en casillas (tiles), versión pura.

El mapa es tratado como un tablero, dividido en casillas, guardando estas información sobre sus características. El movimiento del personaje esta limitado a las casillas, lo que significa que nunca podrá estar a mitad de camino entre una casilla y otra.

Se suelen usar animaciones para el paso de una casilla a otra, para crear la ilusión de movimiento suave, pero en cuanto a a lógica del juego se refiere, el personaje siempre esta encima de una casilla determinada. Esta técnica es la más fácil de implementar, pero impone grandes restricciones en el control del personaje, haciéndola poca conveniente para el desarrollo de un juego de plataformas tradicional. Es muy popular en el desarrollo de juegos tipo puzzle, y es la técnica utilizada para en el primer desarrollo de este proyecto.



Flashback, mostrando los límites de las casillas.

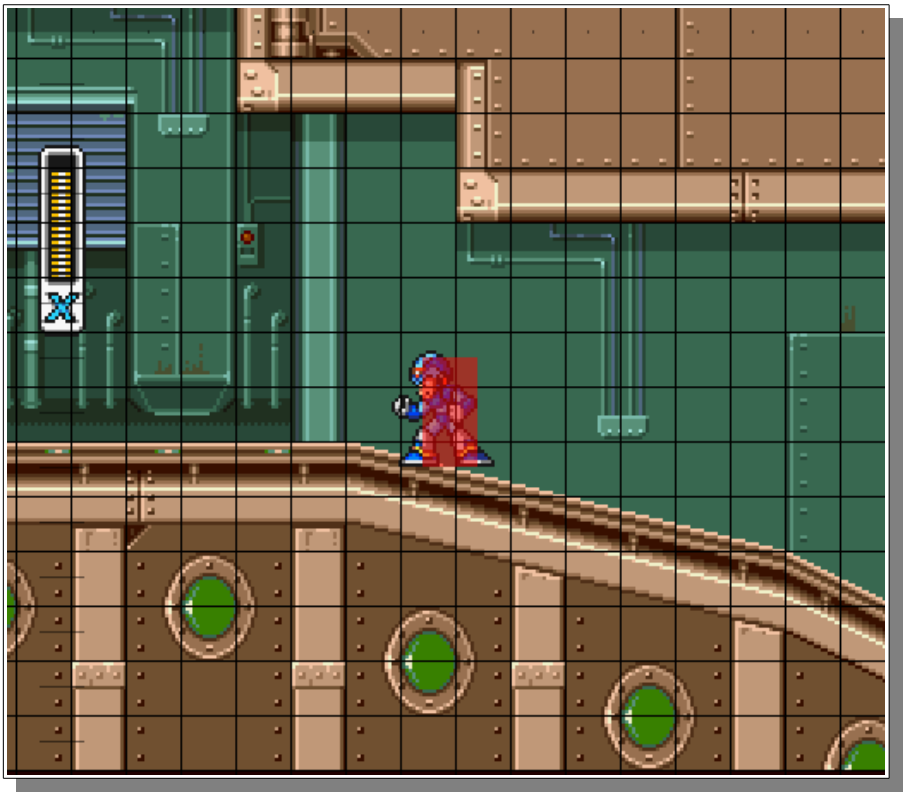
En este tipo de juegos es muy raro ver al jugador haciendo movimientos diagonales, aunque al descomponer el movimiento por ejes y mover varias casillas de una vez, se puede crear esta sensación. Las ventajas de este sistemas son la simplicidad y la precisión, ofreciendo mucho control sobre las mecánicas del juego y disminuyendo la necesidad de manejar casos especiales. Ejemplos de juegos conocidos que usen este método: Flashback[48] y Prince of Persia[49].

2. Basado en casillas (tiles), versión suave.

Al igual que en el caso anterior, las colisiones se siguen determinando por un mapa de casillas, pero el personaje ahora puede moverse libremente por todo el mapa. Esta era la forma más común de implementar juegos tipo plataformas (saga Mario, saga Sonic, y muchos más) en las consolas de 8 y 16 bits (NES, Super Nintendo, Sega MegaDrive, etc ...), y sigue siendo popular a día de hoy. Es un sistema relativamente fácil de implementar y hace que la edición de niveles sea más simple que con técnicas más sofisticadas.

El mapa sigue siendo una cuadrícula como en el caso anterior, en la cual cada casilla guarda información sobre si misma; tipo, imagen a mostrar, sonidos a reproducir al entrar en ella, etc... . La principal diferencia con la versión pura de esta técnica es como se mueve el personaje. Ahora está representado por un rectángulo alineado con los ejes (Axis-Aligned Bounding Box o hitbox) que representa su área de colisión, su tamaño suele ser múltiplo del tamaño de la casilla. Es importante que este rectángulo que envuelve al personaje este bien ajustado al sprite que lo representa, para dar al usuario una sensación más justa en lo que se refiere a las colisiones del jugador.

48 http://en.wikipedia.org/wiki/Flashback_%28video_game%29
49 http://en.wikipedia.org/wiki/Prince_of_Persia_%281989_video_game%29



Mega Man X , mostrando los límites de las casillas y el hitbox del personaje.

Este sistema es simple y fácil de implementar si a las casillas las tratamos como bloques sólidos o vacíos. Se complica cuando introducimos rampas por las que puede avanzar el personaje, como se ve en la imagen superior. Mejora mucho la calidad del juego, tanto en jugabilidad como en el aspecto visual, pero complica la implementación.

Este sistema con rampas fue el elegido para el segundo desarrollo, usando el editor Tiled para la creación de niveles. Se describirá después de comentar otros dos sistemas más complejos.

3. BitMask.

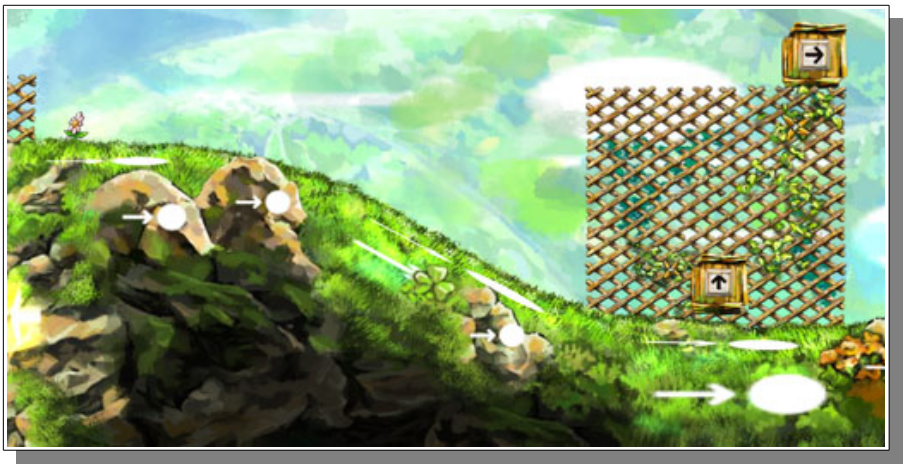
Parecido al sistema anterior, “Basado en casillas, versión suave”, pero en vez de usar grandes mapas de casillas, se usa una imagen para determinar las colisiones para cada pixel. Esto incrementa significativamente la complejidad, uso de memoria y requiere de un editor de imágenes para crear los niveles. Esto normalmente implica que no se usaran casillas para el pintado, por lo que se requerirá un gran trabajo artístico para cada nivel. Debido a estos asuntos, es una técnica poco usada, aunque puede producir resultados de mayor calidad que usando casillas. Encaja muy bien en entornos dinámicos, ya que se puede dibujar en el bitmask para cambiar el escenario.

Worms World Party, con terreno destruible.

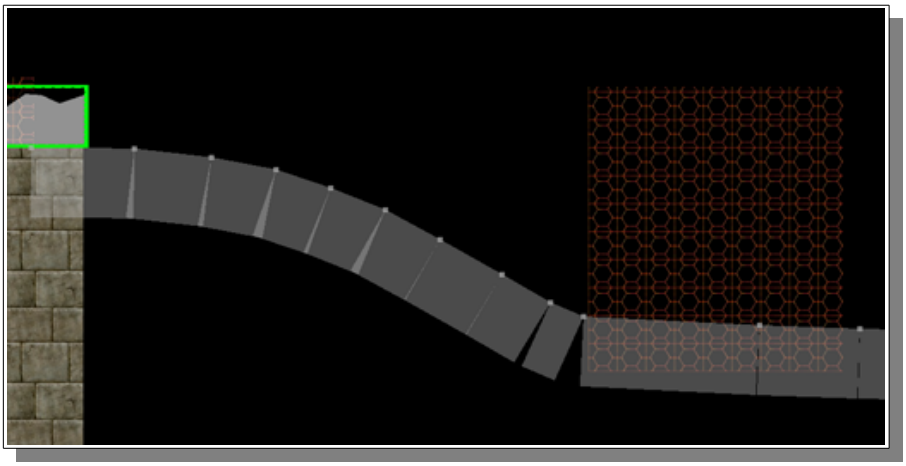
Una manera de implementarlo sería tratando cada pixel como una casilla de los sistemas anteriores, y controlando las colisiones de manera similar. Al igual que antes gestionar las rampas o zonas inclinadas será la parte más complicada.

4. Vectorial.

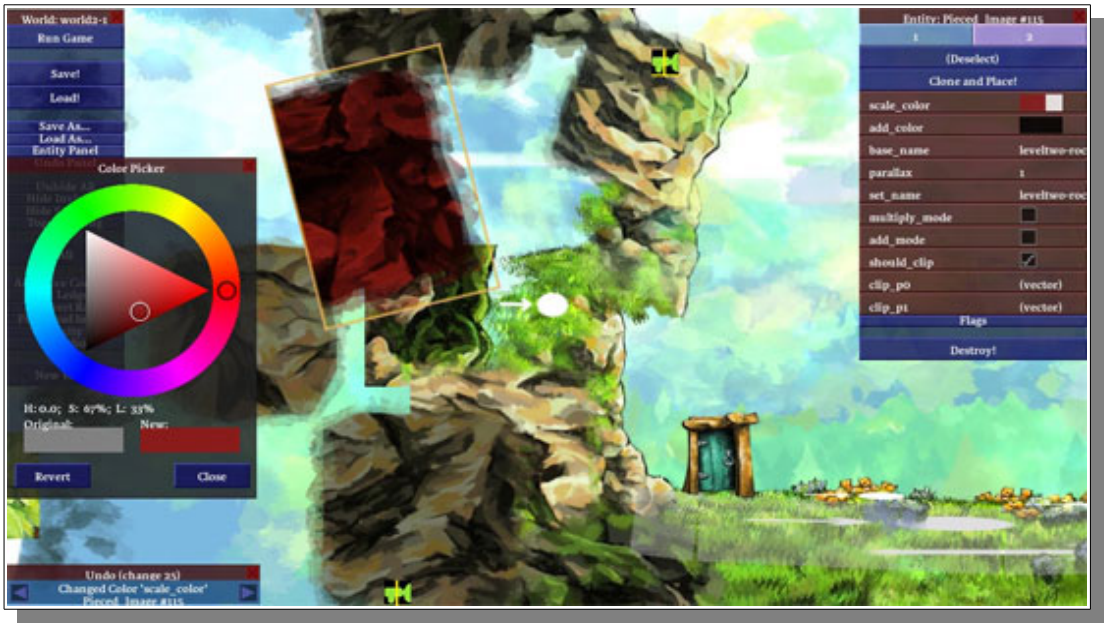
Esta técnica usa polígonos y lineas para determinar el perímetro de las áreas de colisión. Complicado de implementar correctamente, está ganando en popularidad gracias al aumento en la disponibilidad de motores de físicas 2D, como Box2D, los cuales simplifican la implementación. Proporciona beneficios similares a la técnica anterior, pero sin la sobrecarga de memoria y usando un ángulo completamente diferente para la edición de niveles.



Braid, capa visible.



Braid, capa colisiones.



Editor de niveles de Braid.

Como se ha comentado, una posibilidad sería usar un motor de físicas que nos solucione el movimiento y las colisiones. Es una opción popular pero con el inconveniente de que al usar un motor que está disposición de cualquiera, nos quede un juego muy parecido, en sensaciones, a cualquier otro que use ese mismo motor.

Captura del juego Limbo, también usa la técnica vectorial.

En este sistema el editor de niveles es una herramienta fundamental, que seguramente se tenga que desarrollar junto al juego, ya que es un elemento muy ligado a la implementación elegida.

Una vez vistas las diferentes técnicas se entrará en detalle en la solución escogida, basado en casillas versión suave. Comentando por un lado cómo se controlan las colisiones y cómo se gestionan las rampas, el uso del editor seleccionado y cómo se cargan los mapas en el juego. Por último se comentará como se lleva a cabo el pintado y la gestión del scroll.

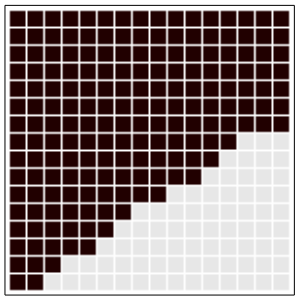
- **Estructura y control de colisiones.**

Para la estructura del mapa se eligió una matriz, cuyo tamaño varía según el nivel y siendo cada elemento del tipo *tile*, que almacena el tipo de la casilla representado por un entero y un vector que representa la altura interna de la casilla, utilizado para controlar las rampas.

```
...

struct tile{

	int tipo;
	int heightmap[TAM_TILE];
};

class Mapa{

public:

...

private:

	tile **zona;
...
}
```

La estructura *tile* cuenta con dos atributos, el primero, *tipo*, se usa para el control de colisiones y el pintado. El segundo es un vector de enteros que representa la altura interna de cada pixel que forma parte de esa casilla, forma parte de la solución elegida para resolver el problema de

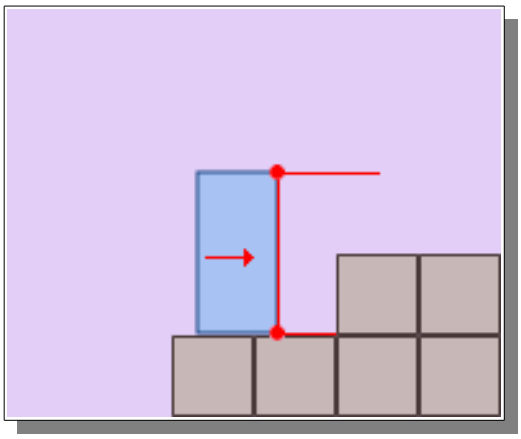
las rampas. Los tamaños de casillas más comunes son múltiplos de 2 y suelen estar en el rango 16 – 128. Por ejemplo, para tamaño de casilla 16 la representación visual y lógica de una rampa sería la siguiente:

Visualmente. 

El heightmap asociado a esa casilla sería: 0, 0, 1, 2, 2, 3, 4, 5, 5, 6, 6, 7, 8, 9, 9, 9. Se observa como cada valor representa la altura interna de la casilla.

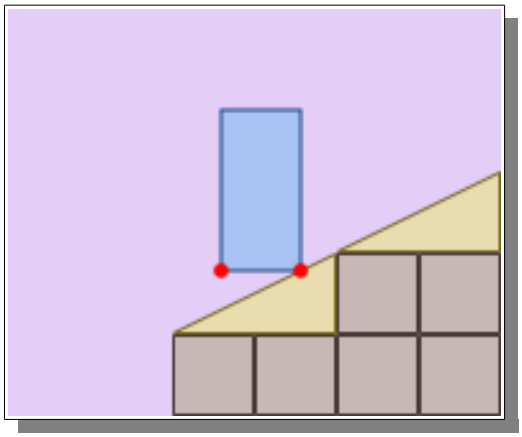
Esta clase proporciona dos métodos principales para resolver las colisiones de las entidades con el mapa. Estos métodos calculan la distancia desde una entidad hasta el obstáculo del mapa más cercano. Gracias a este cálculo el control es capaz de gestionar el desplazamiento de las entidades en cada ciclo, permitiendo moverse a la entidad la diferencia entre lo que quiere y lo que realmente puede. El cálculo de esta distancia se realiza en varios pasos, empezando por descomponer el movimiento por ejes, y luego:

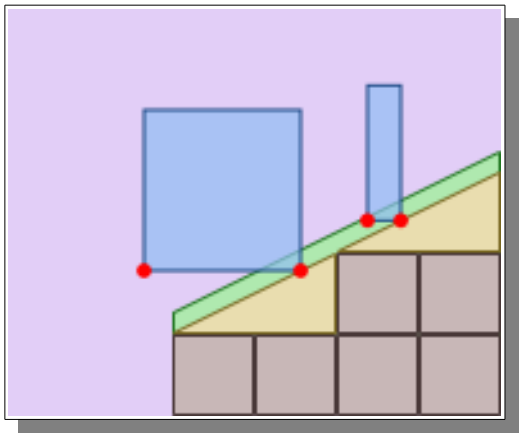
1. Obtener la coordenada del lado, en sentido del movimiento, de la entidad a la que se le quiere comprobar las colisiones. Si está caminado a la izquierda, coordenada del lado izquierdo del rectángulo que la representa.
2. Calcular el valor máximo y mínimo de casilla del mapa con las que intersecta el rectángulo de colisiones de la entidad.
3. Escanear en el mapa la zona encerrada por las lineas obtenidas, en dirección del movimiento hasta encontrar un obstáculo.



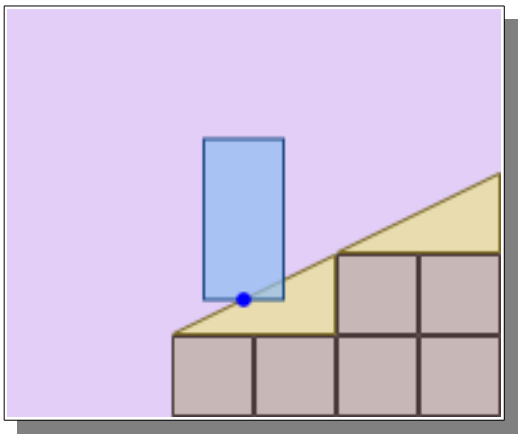
Este es el algoritmo básico sobre el que se construyeron los métodos, hay que añadirle el control para las rampas y casillas especiales. A continuación se explican diferentes problemas encontrados durante el desarrollo de esta parte, principalmente para la implementación de las rampas.

A continuación se comentan los problemas encontrados al usar rampas en el mapa, junto a las soluciones adoptadas.

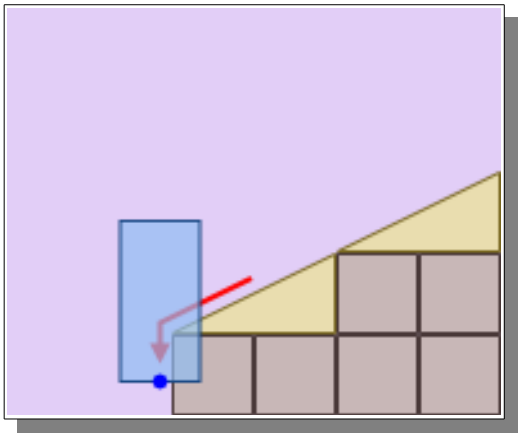




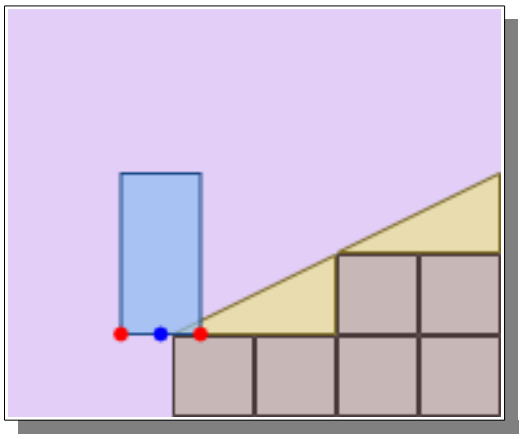
En la primera imagen se observa el efecto que produce el uso de las esquinas del rectángulo y las rampas, dando una mala sensación. Una solución rápida es la que se ve en la segunda imagen, en la que se añade una franja verde que forma parte de fondo y no se toma para el control de colisiones. Solución que depende mucho del tamaño de las entidades, ya que se ve que para el rectángulo fino funciona pero no para el grande.

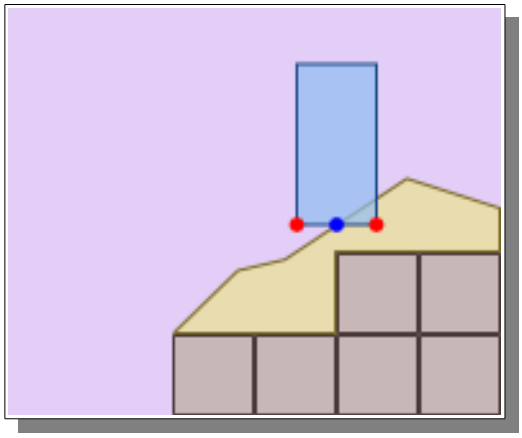


Una solución a este problema es utilizar un punto medio en vez de las esquinas, lo que arregla la visualización sobre las rampas. Pero esto también trae consigo el problema que se observa en la siguiente imagen.



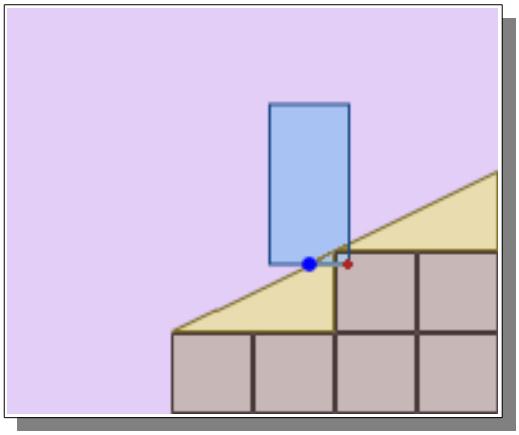
El problema es que la entidad caería por el precipicio antes de los esperado, lo que daría una mala sensación al usuario. La solución que resolvió todos estos problemas fue usar a la vez el punto medio y las esquinas.





Durante el cálculo de la distancia vertical hacia abajo en busca de obstáculos, el punto medio tiene en cuanta las rampas, mientras que las esquinas solamente cuenta las casillas totalmente sólidas. De estos tres valores se selecciona el más pequeño como resultado. Hacia arriba el sistema de las esquinas es suficiente y no se detectaron problemas.

Otro detalle que se observa es que las esquinas inferiores del rectángulo generaran colisiones con las rampas, en las dos imágenes anteriores se observa como la esquina inferior derecha se encuentra dentro de la rampa. Este detalle se salvó omitiendo las rampas en el cálculo del eje X, aunque esto generó el siguiente problema, ilustrado por la siguiente imagen.



La esquina inferior derecha genera una colisión con una casilla sólida, lo que nos impediría continuar moviéndonos a la derecha. Esta situación se da siempre con las casillas sólidas contiguas a rampas, por lo que con ignorar este preciso caso soluciona el problema.

A continuación el código para el cálculo de la distancia al primer obstáculo hacia la derecha. En el código aparece mucho la palabra slope, su significado es rampa en inglés.

```
//Se calculan 2 distancias desde las esquinas de la hitbox, se devuelve la menor
float Mapa::distanciaPrimerObstaculoHorizontal(EntidadMov *e){

  int coorObst;
  float ccolision;
  int columna, minFila, maxFila;

    if(e->vx > 0){

       //Obtener la coordenada del lado en dirección del movimiento
      ccolision = e->fx + e->ancho;
      columna = ccolision / TAM_TILE;

       //Calcular el valor máximo y mínimo de tiles del mapa con las que intersecta el
rectángulo de colisiones de la entidad
       minFila = (int(e->fy)) / TAM_TILE;
       maxFila = (int(e->fy) + e->alto) / TAM_TILE;

       //Calcula la distancia al obstáculo estático mas cercano, Escaneando en el mapa
la zona encerrada por las lineas obtenidas, en dirección del movimiento
       int i, j;
       for(i = columna; i < columna+3; i++) {
         for(j = minFila ; j <= maxFila; j++){

       //Colisiones con tiles solidas, tipo 0 al 17, y limite der del mapa
         if((zona[j][i].tipo > 0 && zona[j][i].tipo < 17) || i == maxColumnas-1){

//comprobamos si la tile adyacente es una rampa orientada a la der(tipo de 17 a 32)
           if(zona[j][i-1].tipo > 16 && zona[j][i-1].tipo < 33) return 99;
           coorObst = i*TAM_TILE;
           return coorObst - ccolision - 1;
         }
       }
```

```
      }
    }
...
}
```

Existen varios detalles a comentar, el más tres del primer "for" controla que la búsqueda de obstáculos no vaya más allá de 3 casillas, en vez de seguir hasta el final del mapa o encontrar un obstáculo. Además generan colisión los limites del mapa, para evitar posibles accesos fuera de rango.

Hacia la izquierda y hacia arriba son muy parecidos, mientras que como se ha explicado el cálculo hacia abajo es el más problemático, el cual se muestra a continuación extraído del método *distanciaPrimerObstaculoVertical*.

```
if(e->vy >= 0){

   int posx, posy;
   float distancia1, distancia2, distancia3;

   //punto medio
   posx = e->fx + e->ancho/2;
   posy = e->fy + e->alto;

   //Ajuste si esta en escalera
   if(e->enEscalera) return 0;

   distancia1 = calcularDistanciaAlsuelo(posx, posy);
   //-1 para mantener la hitbox por encima del mapa
   if(distancia1 < 5) return distancia1 - 1;

   //segundo punto, esquina inferior der
   posx = e->fx + e->ancho;
   posy = e->fy + e->alto;
   distancia2 = calcularDistanciaAlsueloSinSlopes(posx, posy);

   //tercer punto, esquina inferior izq
   posx = e->fx;
   distancia3 = calcularDistanciaAlsueloSinSlopes(posx, posy);

   //-1 para mantener a la entidad 1 pixel por encima del mapa
   return min(distancia2, distancia3) - 1;
}
```

Se apoya en los métodos *calcularDistanciaAlsuelo* que se usa para calcular la distancia del punto medio, teniendo en cuenta las rampas, y *calcularDistanciaAlsueloSinSlopes* que como su nombre indica no tiene en cuenta las rampas. El primer método es el más interesante, ya que el segundo es muy parecido a los anteriores al no tener en cuenta las rampas.

```
float Mapa::calcularDistanciaAlsuelo(float posx, float posy){

      float distancia, alturaPunto;
```

```
	int fila, columna, alturaMapa, xentile, i;

	//calcular en que tile del mapa se encuentra ese punto
	fila = posy / TAM_TILE;
	columna = posx / TAM_TILE;

	//Comprueba si el punto esta en medio de una one way plataform o escalera, en
ese caso no la tiene en cuenta
	if(zona[fila][columna].tipo == 50 || zona[fila][columna].tipo == 49){

		return 99;
	}

	//calcular la posición relativa de ese punto dentro de la tile
	xentile = posx - columna*TAM_TILE;

	//calculamos altura del punto con respecto al mapa
	alturaPunto = maxFilas*TAM_TILE - posy;

	//calculamos la altura del mapa hasta el primer obstáculo, desde el punto
	alturaMapa = 0;
	for(i = fila; i < maxFilas; i++){

		//sumamos las alturas parciales hasta encontrar un obstáculo
		alturaMapa += zona[i][columna].heightmap[xentile];
		//Contamos todas las tiles
		if(zona[i][columna].tipo > 0) break;
	}
	//corregimos el valor de la altura
	alturaMapa += ((maxFilas - 1)-i)*TAM_TILE;

	distancia = alturaPunto - alturaMapa;

	return distancia;
}
```

Este método comprueba primero casos especiales, “one way plataforms” y escaleras, en ese caso se omite el resto y se devuelve una altura ficticia, que representa que no hay colisión. Sino calcula la altura utilizando los heightmap (mapas de alturas) de las casillas involucradas y su propia altura con respecto al mapa.

Además de estos métodos, que son los más importantes para el sistema de colisiones, se crearon otros para resolver los casos especiales. Se trata de métodos booleanos que comprueban si la entidad se encuentra en una casilla de un tipo especifico, por ejemplo *`enTileSalida(int x, int y)`* que devuelve verdadero si el punto a comprobar se encuentra en una casilla de tipo salida. También implementa dos pequeños métodos que se necesitaron para la IA de los enemigos, comprueban si una entidad se encuentra al borde de un precipicio, y se usa para que los enemigos cambien de sentido al detectar uno.

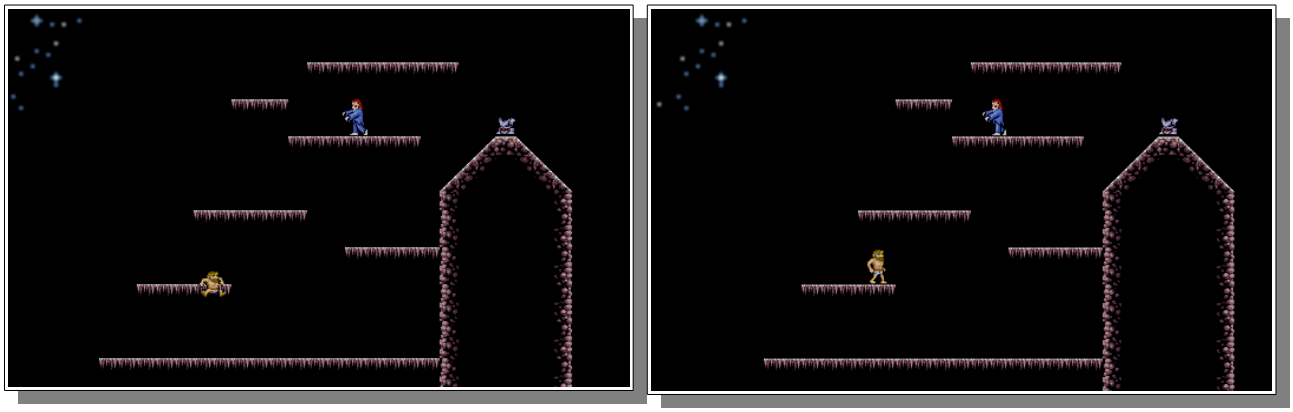
A continuación se comentaran brevemente funcionalidades especiales del mapa, las mencionadas “one way plataforms” y escaleras.

- One way plataforms: como se ha comentado antes, son plataformas que generan

colisión cuando el personaje esta por encima de ellas, pero no al estar debajo. La implementación es sencilla, basta tener en cuenta los siguientes factores:

1. En el eje X no generan colisiones.
2. En el eje Y, son obstáculo solamente si el personaje esta completamente por encima de la plataforma.

En nuestro caso al estar separados el control hacia abajo del de arriba, estas casillas sólo se tienen en cuenta hacia abajo.



Capturas del proyecto en la que se ve el personaje en mitad de una "one way plataform" (izquierda), y de pie en la misma plataforma (derecha).

- Escaleras: también sencillas de implementar, cuando el personaje se encuentra en una de ellas se trata como un caso especial. Para subir o bajar por ellas el usuario debe colocarse dentro de una casilla de este tipo y apretar hacia la tecla adecuada. Mientras se está en la escalera el usuario no podrá ni saltar ni disparar, si podrá moverse horizontalmente y saldrá de la escalera en cuanto el punto medio de lado inferior abandone este tipo de casilla, o llegando al punto máximo o mínimo de la escalera.

Captura del proyecto con el personaje en medio de una escalera.

- **Creación y carga de niveles.**

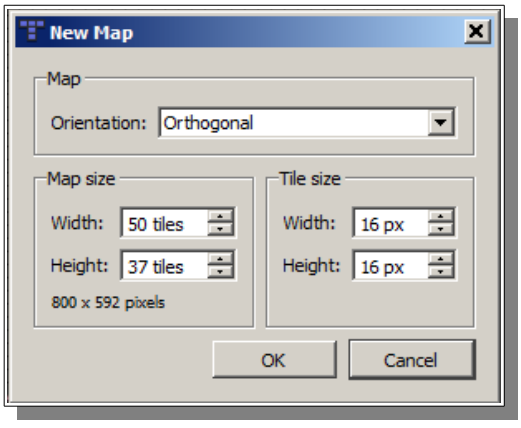
Un aspecto fundamental del proyecto era el utilizar un editor libre de mapas para la creación

de niveles para el juego. Esto requiere establecer el formato que deberán tener todos en común para que nuestra aplicación pueda leerlos sin problemas y de esta forma separar la creación de niveles de la implementación del juego. De esta manera cualquier persona que sepa usa el edito seleccionado y respetando el formato establecido puede crear mapas para el juego. El editor seleccionado como se ha comentado anteriormente es Tiled, es un editor de mapas de casillas(tile map) de propósito general, fácil de usar que proporciona una gran cantidad de herramientas y varios formatos de salida. Ademas cuenta con funcionalidades comunes a cualquier editor; deshacer la ultima acción, cortar, copiar y pegar trozos de mapa, hacer zoom, reajustar las dimensiones del mapa, etc... siendo para mi la más importante el poder trabajar con capas.

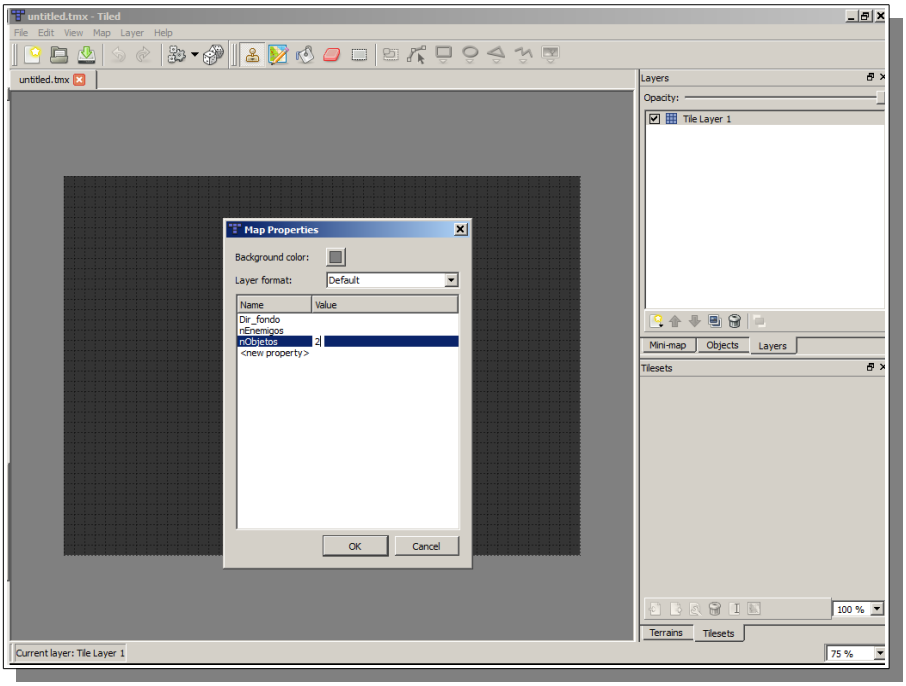
El programa permite guardar el mapa en diferentes formatos, en este proyecto se usará el formato ".txt" para la carga por parte de nuestra aplicación, por ser el más sencillo para leer y así poder dedicar mas tiempo a otras tareas. De todas formas durante la creación y edición del mapa se trabajará con el formato ".tmz", y una vez terminado se exportara a formato ".txt", guardando siempre el original en ".tmz". Esto es así porque al guardar en ".txt" se pierde información que para el uso del mapa en el juego no afecta, pero si queremos volver a editar el mapa (esto pasará seguro) si que se notará la perdida y es muy probable que se tenga que empezar de nuevo.

Tiled es un editor muy completo con muchas funcionalidades, esta parte no pretende ser un manual de como se usa, simplemente un resumen de como se ha usado. A continuación se comentaran los pasos seguidos para crear un mapa siguiendo el formato aceptado por nuestra aplicación.

1. Crear el mapa
    1. File -> New.
    2. Seleccionamos orientación ortogonal, pudiendo también elegir isométrico.
    3. Introducimos el tamaño del mapa, en alto y ancho.
    4. Introducimos el tamaño de las casillas, para este proyecto usaremos 16x16, también en alto y ancho.

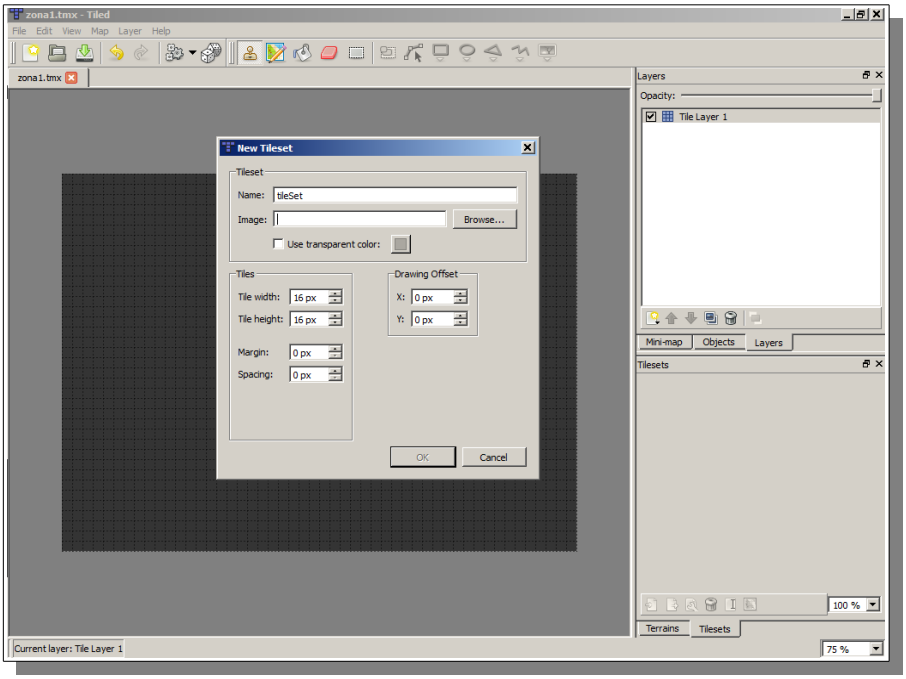


2. Añadir nuevas propiedades al mapa, Map-> Map Properties:
    1. `Dir_fondo`, tendrá como valor la dirección de la imagen que se usará de fondo para esta zona en concreto.
    2. `nEnemigos`, tendrá como valor el número de enemigos que hay en la zona.
    3. `nObjetos`, número de objetos en la zona.

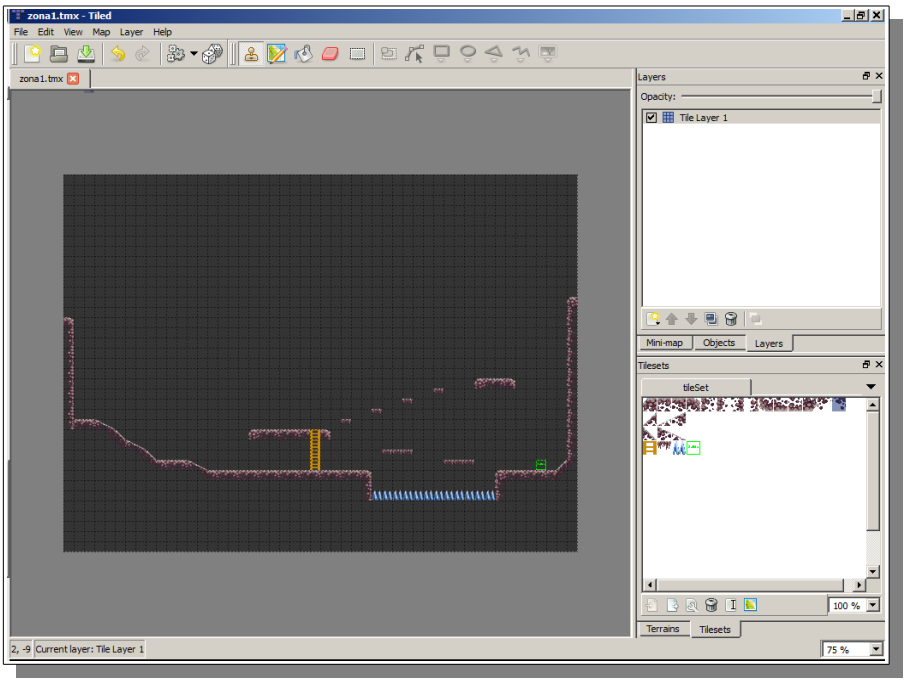


3. Añadimos la imagen que se usara para pintar el mapa (tileset), contiene las diferentes imágenes para cada tile o casilla.

    1. Map -> New Tileset.
    2. Seleccionamos el tamaño de cada tile, en este caso 16x16.
    3. Margen entre tiles, dentro de la imagen. En este caso no hay margen.
    4. Tampoco usaremos el Drawing Offset.

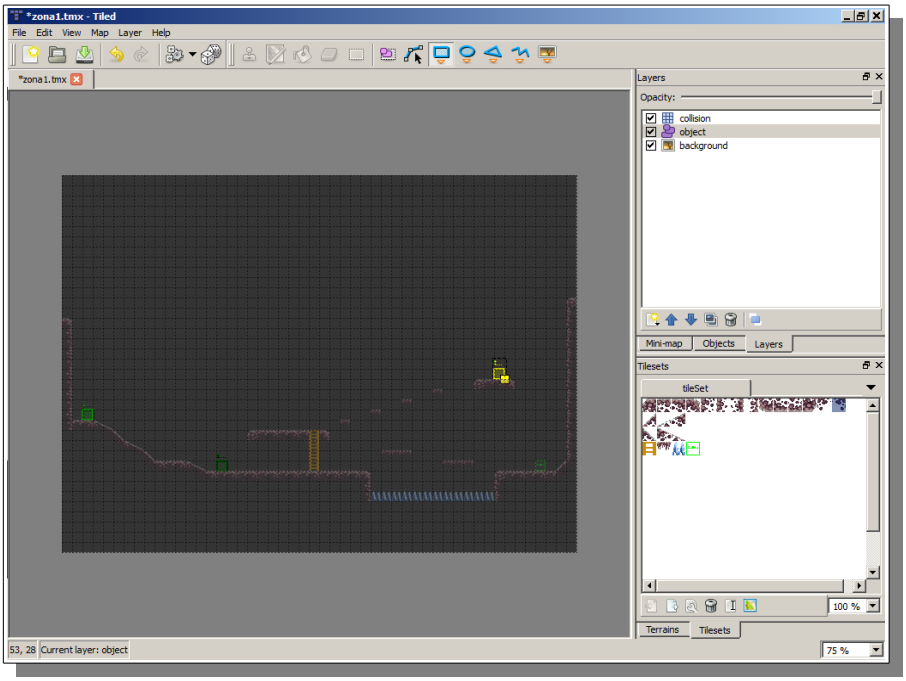


4. Después de estos pasos ya podemos empezar a crear el mapa seleccionando del TileSet que aparece abajo a la derecha el tile que queremos introducir y haciendo click en la casilla donde queramos ponerla.

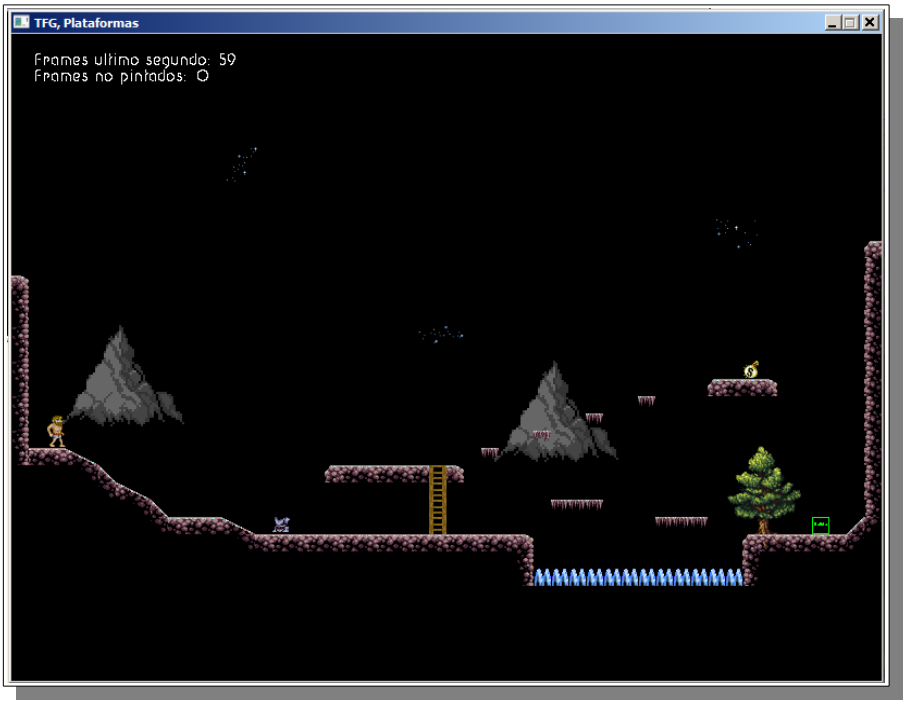


5. Una vez terminado de establecer las casillas, hay que añadir una capa tipo Object que contendrá las entidades del nivel y otra tipo Image, que no se usará pero es necesaria por el programa.

    1. Layer -> Add Object Layer y Layer -> Add Image Layer .
    2. En este momento tenemos tres capas, debemos renombrarlas; la capa “Object ...” con nombre “object”, la capa “Tile …” a “collision” y la capa “Image” a “background”. Esto es una obligación si queremos trabajar con el formato ".txt", sino al exportar nos dará error.
    3. A la capa background no se le da un verdadero uso, ya que la dirección del fondo a usar la paso como propiedad del mapa. Se ha usado para comprobar como iba quedando el fondo sobre el mapa.
    4. El lector de mapas espera que las capas estén organizadas, el orden en el editor ha de ser: la primera la collision, segunda la de object y tercera background.



Captura de la zona uno en el editor.

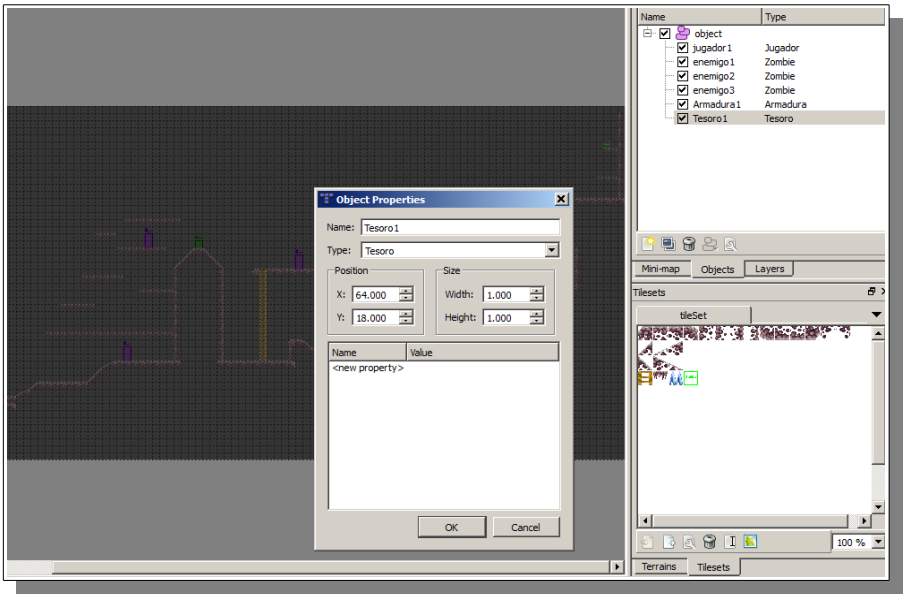


Captura de la zona uno en el juego.

6. La capa object se usa para localizar al jugador, los enemigos y objetos del mapa.

    1. Lo primero es definir nuestros propios objetos, Edit -> Preferences -> Object Types, aquí definiremos nombre y color representativo.
    2. Con la capa object seleccionada, podemos introducir objetos de diferentes formas; polígonos, rectángulos, elipses o polilineas. Yo he usado rectángulos para representar los diferentes objetos.

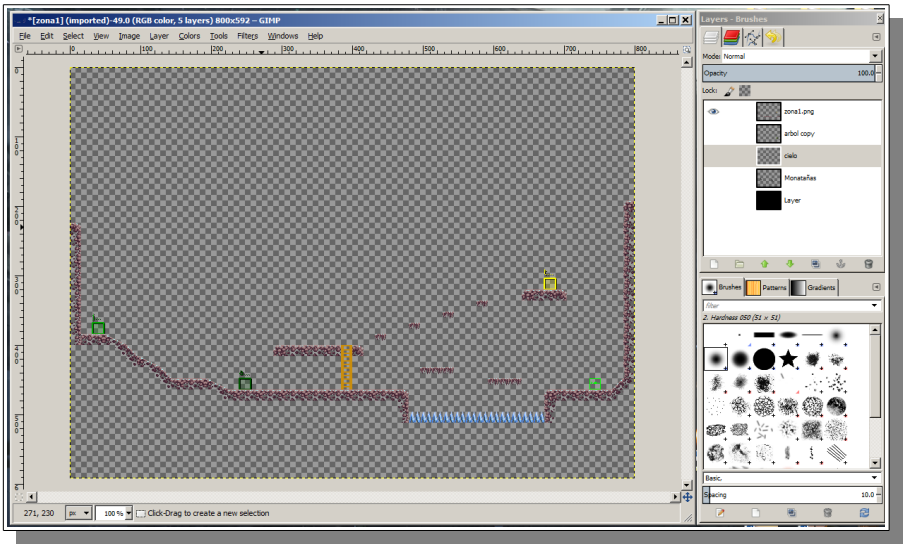
3. El primer objeto de todos debe ser el jugador, después enemigos y por último los objetos que el personaje puede recoger.
4. Una vez introducido el objeto, podemos acceder a sus propiedades en donde asignarle el tipo.



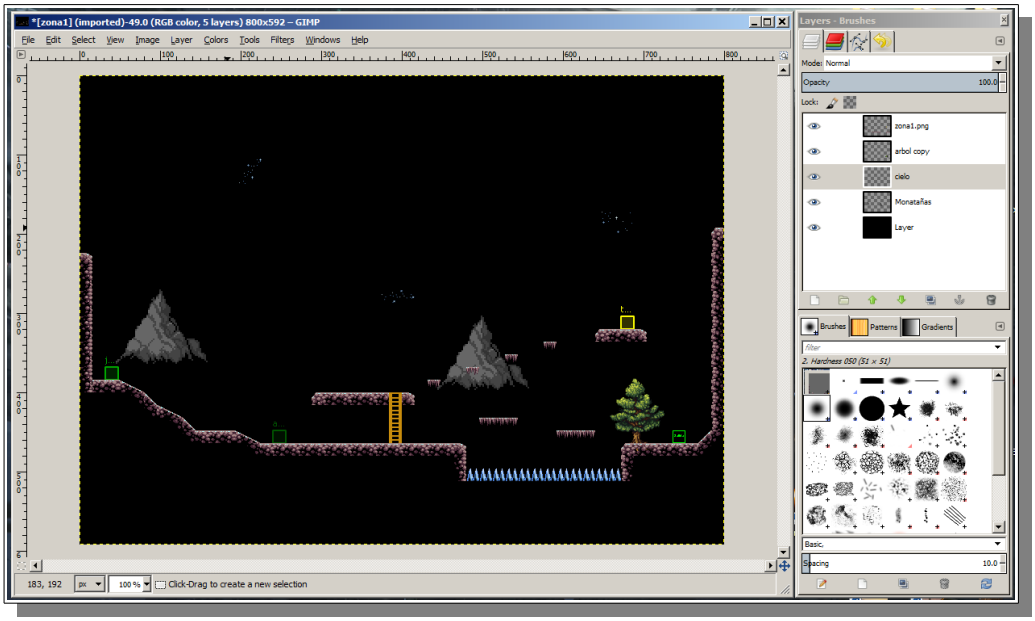
Una parte fundamental de cada nivel o zona es la imagen de fondo que se usará, ya que mejora el apartado visual notablemente. Estos fondos están creados con el editor de imágenes Gimp, es otra aplicación muy conocida también libre y gratuita, que ofrece infinidad de posibilidades. La creación de las imágenes que componen un juego suelen estar a cargo de profesionales de la materia, para conseguir un buen acabado. Para facilitar esta tarea lo que se hizo fue buscar pequeñas imágenes en internet (arboles, montañas, etc ...) con las que componer los fondos. El acabado podría mejorarse mucho empleando más imágenes y tiempo en el desarrollo, pero por cuestiones de tiempo y que es una tarea un poco fuera del ámbito del proyecto se limitaron las horas dedicadas a esto. Todas las imágenes utilizadas en este proyecto no son de mi propiedad, han sido sacadas de internet como se ha dicho, por lo que no se podrían usar en un desarrollo comercial. A continuación se explican los pasos seguidos para la creación de los fondos para los niveles.

Para la creación del fondo es necesario tener el mapa terminado, ya que en base a este diseñaremos el fondo. Los principales pasos son los siguientes:

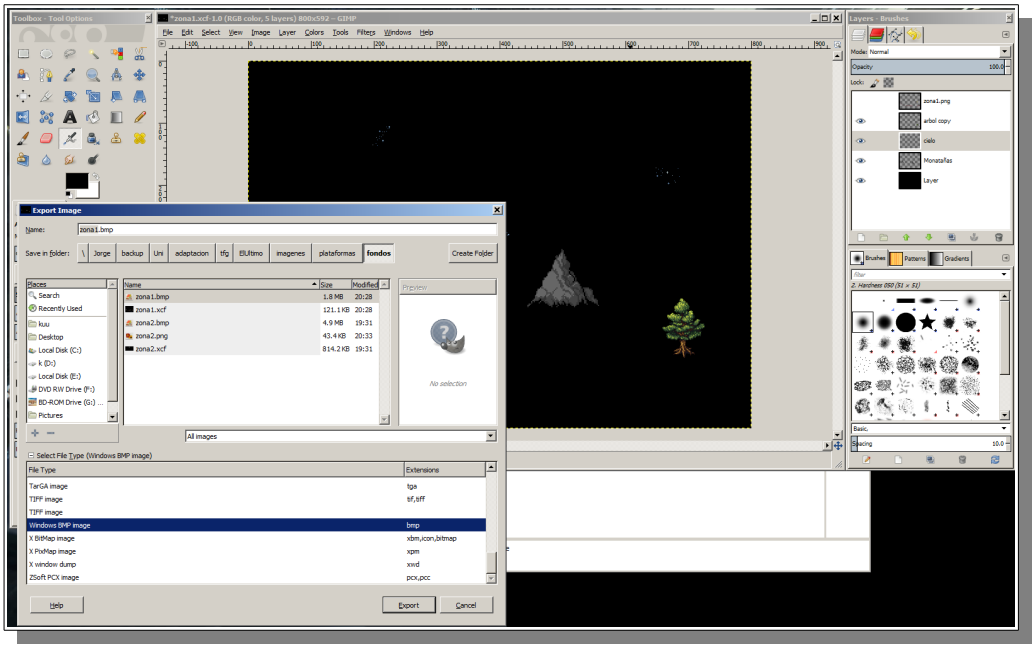
1. Con el programa de creación de mapas (Tiled) generamos una imagen a tamaño real del mapa creado. File -> Save As Image.



2. Con esta imagen nos vamos a un editor de imágenes como Gimp y la usamos de guía para ir colocando diferentes elementos para formar nuestro fondo.



3. Quitando la visibilidad de la capa del mapa, exportamos nuestra composición en formato ".bmp" con el nombre adecuado.



Por último se comentará brevemente el formato final que tendrá el nivel y como se lleva a cabo la carga durante el juego. Siguiendo los pasos antes citados al exportar el nivel, con File->Export As... y usando el formato ".txt" obtendremos algo parecido a:

```
[header]
width=50
height=37
tilewidth=16
tileheight=16
Dir_fondo=../imagenes/zona1.bmp
nEnemigos=0
nObjetos=2

[tilesets]
tileset=../imagenes/tileSet.bmp

[object]
# jugador1
type=Jugador
location=2,23,1,1

[object]
# armadura1
type=Armadura
location=15,28,1,1

[object]
# tesoro1
type=Tesoro
location=42,19,1,1

[layer]
type=collision
data=
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,2,3,4,0,0,0,0,0,5,
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,50,0,0,0,0,0,0,0,0,0,0,0,0,6,
8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
8,2,3,34,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
0,0,0,0,14,33,0,0,0,0,0,0,0,0,0,0,0,0,1,2,3,2,3,2,49,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
0,0,0,0,0,14,34,35,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,49,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,
0,0,0,0,0,0,0,14,33,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,49,0,0,0,0,0,0,50,50,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,
0,0,0,0,0,0,0,0,14,3,2,3,34,35,0,0,0,0,0,0,0,0,0,0,49,0,0,0,0,0,0,0,0,0,0,0,0,50,50,50,0,0,0,0,0,0,52,0,17,13,
0,0,0,0,0,0,0,0,0,0,0,0,0,14,3,2,3,2,3,2,3,2,3,2,3,2,3,2,3,4,0,0,0,0,0,0,0,0,0,0,0,0,1,2,3,2,3,2,13,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,51,51,51,51,51,51,51,51,51,51,51,51,6,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Este archivo tiene tres partes fundamentales, la primera va desde [header] hasta antes del primer [object], en esta parte nos encontramos con las características del mapa; altura, anchura, número de objetos y enemigos, la dirección de la imagen de fondo y el tileset a utilizar. Estas imágenes son cargadas a través del método *cargarImagen* de la clase *SDLHelper*. La segunda parte la componen todas las entradas con cabecera [object], son todas las entidades que participan en el nivel, siendo el primero el jugador, después los enemigos y por último los objetos. Por último nos encontramos con la matriz que representa el mapa de colisiones del nivel, con cada entero representando el tipo de la casilla que ocupa.

La carga se realiza a través del método *cargarZona(int nzona, Jugador *jugador, std::vector<EntidadMov*> *enemigos, std::vector<Objeto*> *objetos, SDLHelper *helper)* que se encarga de rellenar los diferentes vectores con las entidades correspondientes ya inicializadas e inicializar la posición del jugador. Además crea la estructura que contendrá el nivel según la altura y anchura indicadas en el archivo. Una vez creado el mapa se introducen los valores de cada casilla y se inicializan los heightmaps de cada una según su tipo.

- **Pintado.**

El pintado del mapa consta de dos partes, por un lado gestionar el movimiento relativo del mapa con respecto al jugador (scroll) y por el otro pintar únicamente aquellas casillas que sean visibles, junto a la parte del fondo que corresponda.

Algo que caracteriza a los juegos de tipo plataformas, es como se presenta la escena del

juego. Lo común en esta clase de videojuegos es que el personaje pueda moverse libremente por la pantalla hasta que llega a un límite prefijado, normalmente a mitad de esta. A partir de ese momento el personaje no avanza más en la pantalla sino que es el mapa el que se mueve para dar la sensación de movimiento y así permitir al usuario poder ver los obstáculos y enemigos que vienen con antelación. Esta técnica se conoce como scrolling y puede ser horizontal, vertical o multidireccional. Es una técnica sencilla de implementar pero tiene un gran impacto en el juego. En nuestro desarrollo se han implementado el scroll multidireccional, de esta forma los mapas pueden tener cualquier tamaño, incluso ser más pequeños que la pantalla.

Detalle del tamaño de la escena con respecto al mapa.

```
void Mapa::pintar(SDL_Surface* pantalla, int scroll_x, int scroll_y){

	int i,j;
	SDL_Rect tile, tileImgPos;

	int mapx = scroll_x / TAM_TILE;
	int mapy = scroll_y / TAM_TILE;

	//Fine scroll.
	int map_xoff = scroll_x & (TAM_TILE - 1);
	int map_yoff = scroll_y & (TAM_TILE - 1);

	SDL_Rect fondoVisible;
	fondoVisible.x = scroll_x;
	fondoVisible.y = scroll_y;
	fondoVisible.w = ANCHO_PANTALLA;
	fondoVisible.h = ALTO_PANTALLA;

	//Se pinta primero el fondo
	SDL_BlitSurface(fondo, &fondoVisible, pantalla, NULL);

	int maxfilas = min(MAX_FILAS_PANTALLA, maxFilas);
	int maxcolumnas = min(MAX_COL_PANTALLA, maxColumnas);
```

```
	for(i = 0; i < maxfilas; i++){
		for(j = 0; j < maxcolumnas; j++){

			if(zona[i + mapy][j + mapx].tipo > -1){

				tile.x = j*TAM_TILE - map_xoff;
				tile.y = i*TAM_TILE - map_yoff;
				tile.h = TAM_TILE;
				tile.w = TAM_TILE;

				tileImgPos.w = TAM_TILE;
				tileImgPos.h = TAM_TILE;

				switch(zona[i + mapy][j + mapx].tipo){

				case 0:

					tileImgPos.x = 0;
					tileImgPos.y = 64;
					break;

				case 1:
			 ...
			...
			}
			//Pintado del mapa, casilla a casilla
			SDL_BlitSurface(tileSet, &tileImgPos, pantalla, &tile);
		}
	}
}
```

El cálculo del scroll se lleva a cabo en la clase *Control* en base a la posición del jugador e indica, por cada eje, cuanto ha avanzado el personaje más allá de la mitad de la pantalla. Con esa información se calcula la fila y columna inicial por la que se empezará a pintar el mapa (*mapx* y *mapy*).

Además se necesita un cálculo adicional para suavizar el movimiento del mapa, sino sería tosco, avanzando a saltos casilla a casilla. Para ello se realiza lo que en el código está comentado como "Fine scroll", se aprovecha el resto de la división *scroll_* / TAM_TILE* para el movimiento del mapa hasta que se alcance una nueva casilla. Para facilitar este cálculo es preferible usar un tamaño de casilla múltiplo de 2 (2, 4, 8, 16, 32, 64, etc). Por ejemplo; con un valor para scroll_x de 100, el resultado que se guarda en *mapx* sería la división entera, 100/16=6, mientras que la división real sería 100/16=6.25. Para obtener el resto se realiza la operación AND entre la coordenada y el tamaño de las casillas menos uno, 100&15= 4. Este valor nos indica el número de pixeles que debemos restar a la posición de cada casilla para conseguir que el scroll o movimiento del mapa sea suave.

Una vez realizados los cálculos para el scroll se recorta la parte del fondo que está visible, apoyándose también en el valor del scroll, y se pinta sobre la superficie principal. A continuación se pinta el mapa casilla a casilla, habiendo definido antes el máximo de filas y columnas que se

pintaran, que varía entre el número de casillas que entra en pantalla[50] o si el mapa es más pequeño que la pantalla el número máximo de filas o columnas del mapa. En este último caso, aunque se siguen realizando los cálculos para el scroll, no producen ningún efecto.

Las imágenes de cada casilla están todas contenidas en una, se suele llamar "tile set", y puede variar según el mapa.

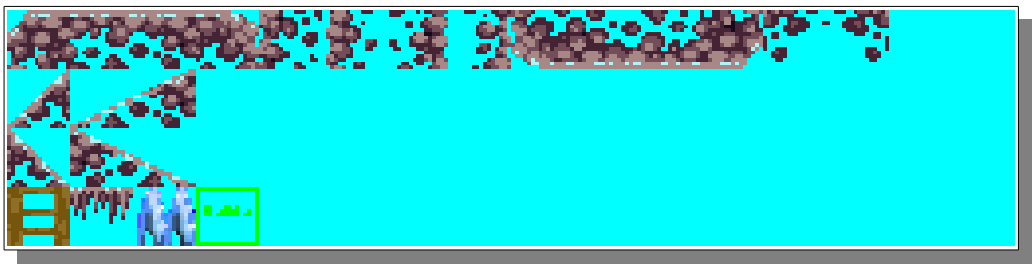


Imagen que contiene todos los tipos de casillas utilizadas.

5.3.4 Colisiones y la clase `Control`.

La clase Control es la encargada de la lógica del juego, controlar que se cumplan las reglas y permitir las mecánicas. Su método principal es ***`actualizarEstado`***, este es llamado por el programa principal para actualizar el estado de todos los componentes del juego. Su principal cometido es el control de colisiones, el cual se realiza en dos fases. Por un lado el movimiento del jugador y enemigos sobre el mapa, tras haber sido ajustados por las físicas implementadas, aceleración, gravedad y saltos. Y por otro lo que se ha llamado colisiones dinámicas, que se refiere a las que se producen entre entidades. Entre ellas están el choque entre jugador y enemigos, los impactos de las balas de jugador con los enemigos y al jugador recogiendo los diferentes tipos de objetos.

```
void Control::actualizarEstado(void){

	//if(jugador1.viva && !jugador1.muriendo) gestionarControles();
	gestionarControles();

//Actualizamos el jugador
	fisicas(&jugador1);
	colisionesConMapa(&jugador1);

//Eliminamos del vector los enemigos que no estén activos y actualizamos el resto
	for(unsigned int i=0; i < enemigos.size(); i++){

		if(!enemigos[i]->activa) enemigos.erase(enemigos.begin()+i);
		else if(enemigos[i]->estaViva()){

			iaZombie(enemigos[i]);
			fisicas(enemigos[i]);
			colisionesConMapa(enemigos[i]);
		}
	}
```

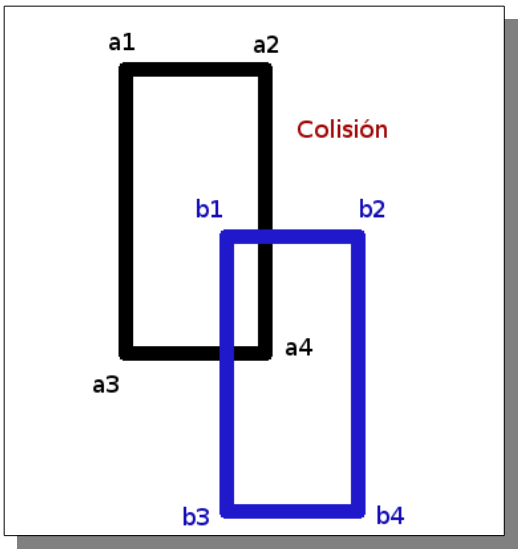
50 En este caso se ha utilizado un tamaño de ventana de 800x600, lo que implica por ejemplo que el máximo número de filas a pintar sería 800/16=50,

```
	//Controlar colisiones entre jugador, objetos, enemigos y balas
	colisionesDinamicas();

//Caso especial que el jugador entra en el final de zona, cargamos siguiente mapa
	if(entidadEnsalida(&jugador1)) cargarMapa();
}
```

Anteriormente se vio como el mapa calculaba las distancias a los obstáculos, según el eje y el movimiento de la entidad que se comprueba. El control con este dato comprueba si lo que se quiere mover la entidad en ese momento es mayor a esta distancia, en tal caso esta distancia se convierte en la cantidad de movimiento que esa entidad realizará en ese momento, y se anulará su velocidad ya que ha colisionado con el mapa en ese eje. Las colisiones en el control se resuelven igual para cada eje, pero respetando un orden, primer el eje X luego el Y. Además al final de este método se comprueban los casos especiales, subir por una escalera o caer en los pinchos.

```
if(e->vx != 0){

	distanciaObst = mapa.distanciaPrimerObstaculoHorizontal(e);

//movemos la entidad lo que quiere si hay espacio
	if((abs(e->vx) <= distanciaObst)) e->fx += e->vx;

//Colision, movemos la distancia que hay hasta el obstáculo y anulamos velocidad
	else{
		if(e->vx > 0)e->fx += distanciaObst;
		else e->fx -= distanciaObst;
		e->vx = 0.0;
	}
}
```

Para las colisiones entre entidades (jugador, enemigos y objetos) se ha elegido un método muy común y fácil de implementar. Aprovechando que internamente todas las entidades tiene una coordenada x e y y un tamaño, formando un rectángulo, las colisiones se resolverán comprobando la intersección de estos.



Representación visual de la intersección de rectángulos.

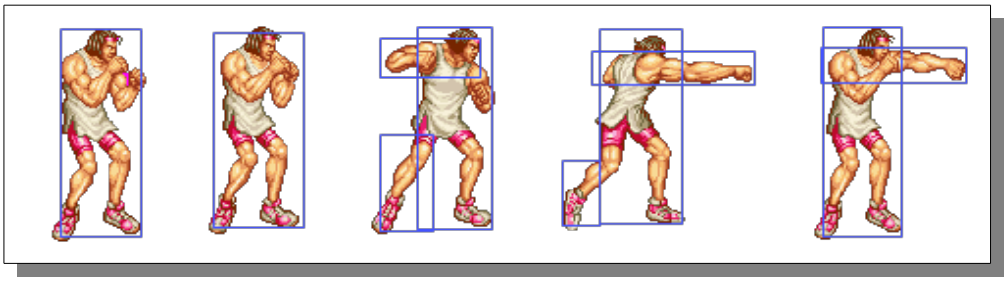
```
bool Control::colisionEntreRectangulos(int a1, int a2, int a3, int a4, int b1, int b2,
int b3, int b4){

	return (((a1 <= b1) && (b1 <= a2)) || ((a1 <= b2) && (b2 <= a2)))
		&& (((a3 <= b3) && (b3 <= a4)) || ((a3 <= b4) && (b4 <= a4)));
}
```

Existen otras técnicas más complejas como la detección de colisiones por pixel, en el que se comprueba si algún pixel del sprite que representa a los personajes colisiona con uno de otro personaje. Es una técnica que se parece mucho a la comentada antes de Bitmask para los mapas. Es un sistema más preciso de cara al programador, pero no es algo que el usuario vaya a notar claramente.

El sistema de rectángulos aparte de la ventaja de ser fácil de implementar y se puede mejorar notablemente si se necesita. El primer método se ha comentado antes cuando se hablaba de la clase *Animacion* donde se hablaba de la importancia de ajustar el rectángulo al sprite para no generar colisiones poco claras.

Ajuste del rectángulo de colisiones sobre un sprite de Mario.

En algunos caso no es suficiente con este ajuste, y se utilizan varios rectángulos para controlar las colisiones. Consiguiendo de esta manera mejores sensaciones para el usuario y mucha más precisión en los cálculos.



Múltiples rectángulos de colisiones.

El uso de esta última técnica depende mucho del sprite en sí como se observa en la imagen. En nuestro caso se han elegido sprites cuya forma se mantenga siempre dentro del rectángulo principal ajustado. Se ha utilizado para detectar las colisiones entre el jugador y los enemigos, las balas del jugador y los enemigos, el jugador y los objetos a recoger.

```
bool Control::colisionEntreEntidades(Entidad *e1, Entidad *e2){

return colisionEntreRectangulos(e1->fx, e1->fx + e1->ancho, e1->fy, e1->fy + e1->alto,
                                e2->fx, e2->fx + e2->ancho, e2->fy, e2->fy + e2->alto);
}
```

Otro uso interesante que se le da a este método es el de detectar que entidades se encuentran dentro de la escena, esto se usa en el método *pintar* para que en cada ciclo solamente se pinte aquello que es visible. En un juego como este con pocos elementos en el mapa no se nota la mejora, pero con cientos o miles de elementos dispersos por el mapa es algo fundamental Para ello se comprueba si la entidad en cuestión colisiona con el rectángulo que forma la escena.

```
void Control::agruparEntidadesAPintar(void){

   entidadesVisibles.clear();

   entidadesVisibles.push_back(&jugador1);

//Enemigos
   for(unsigned int i=0; i < enemigos.size(); i++)
      if(entidadVisible(enemigos[i])) entidadesVisibles.push_back(enemigos[i]);

//objetos
   for(unsigned int i=0; i < objetos.size(); i++)
      if(entidadVisible(objetos[i]))  entidadesVisibles.push_back(objetos[i]);
}

bool Control::entidadVisible(Entidad *e){

   return colisionEntreRectangulos(scroll_x, scroll_x + helper->width, scroll_y,
scroll_y + helper->height, e->fx, e->fx + e->ancho, e->fy, e->fy + e->alto);
}
```

Con estos cálculos hechos el método pintar solamente tiene que actualizar el scroll en base al jugador y pintar los elementos que componen la escena en orden, primero el mapa luego las entidades.

```
void Control::pintar(void){

       actualizarScroll(&jugador1);

       //Mapa
       mapa.pintar(helper->pantalla, scroll_x, scroll_y);

       agruparEntidadesAPintar();
       for(unsigned int i=0; i < entidadesVisibles.size(); i++)
              entidadesVisibles[i]->pintar(helper->pantalla, scroll_x, scroll_y);
}

void Control::actualizarScroll(EntidadMov *e){

//Scroll en horizontal
```

```
   scroll_x = e->fx - limiteHorizontal;

//Limitamos el scroll con el final del mapa
   if(scroll_x + 2*limiteHorizontal > mapa.getAncho())
              scroll_x = mapa.getAncho() - 2*limiteHorizontal;

   if(scroll_x < 0) scroll_x = 0;

//Scroll vertical
   scroll_y = e->fy - limiteVertical;

//Limitamos el scroll con el final del mapa
   if(scroll_y + 2*limiteVertical > mapa.getAlto())
              scroll_y = mapa.getAlto() - 2*limiteVertical;

   if (scroll_y < 0) scroll_y = 0;
}
```

Otros elementos

A continuación se analizan brevemente las dos clases que dan apoyo al jugo, manejando funcionalidades de la SDL necesitadas por el juego: inicialización de la misma y los subsistemas necesarios, gestión del sonido, carga de imágenes, etc.

- **SDLHelper**

Sus funciones principales son la inicialización y finalizacion de la SDL y los subsistemas que se vayan a utilizar, en este caso han sido el subsistema de video, el de timer, el de audio. Como se ha comentado antes se encarga de gestionar los eventos que se producen, tanto de teclado como de cierre de ventana. Proporciona un método para cargar imágenes durante el juego (diferentes fondos o tilesets para los niveles), además de cargar automáticamente aquellas que siempre se usan (los sprites tanto de jugadores como enemigos). Lo importante de este método es que optimiza la imagen al formato de la superficie principal, sino al hacer "blit" con dos superficies con diferente formato, la SDL cambiará el formato en ese momento lo que generaría una pérdida de rendimiento.

```
SDL_Surface * SDLHelper::cargarImagen(string filename) {

//Imagen a cargar
   SDL_Surface* loadedImage = NULL;

//Imagen optimizada
   SDL_Surface* optimizedImage = NULL;

//Cargar la imagen
   loadedImage = SDL_LoadBMP(filename.c_str());

//Comprobar que todo va bien
   if(loadedImage != NULL) {

      //Optimizar la imagen
      optimizedImage = SDL_DisplayFormat( loadedImage );
```

```
        //Liberar la imagen inicial
        SDL_FreeSurface( loadedImage );

        //Comprobar que ha ido bien la optimización
        if(optimizedImage != NULL) {

                //Establecemos el color llave
                Uint32 colorkey = SDL_MapRGB( optimizedImage->format, 0, 0xFF, 0xFF );
                //Eliminamos el color llave
                SDL_SetColorKey( optimizedImage, SDL_SRCCOLORKEY, colorkey );
        }
    }
    else{

        cerr << "No se pudo cargar la imagen: " << SDL_GetError() << endl;
    }

//Retornamos la imagen optimizada
    return optimizedImage;
}
```

Para escribir en pantalla se usa una la librería adicional SDL_ttf, sobre la que se ha hablado antes. En esta clase se inicializa la librería y se crea la fuente que se usará. El siguiente método permite escribir la cadena especificada comenzando en el punto dado.

```
void SDLHelper::pintarCadena(int x, int y, string text){

        SDL_Rect pos;
        pos.x = x;
        pos.y = y;

        SDL_Color color = {255,255,255};
        SDL_Surface *text_surface;

        text_surface = TTF_RenderText_Solid(font, text.c_str(), color);

        if (text_surface != NULL){

                SDL_BlitSurface(text_surface, NULL, pantalla, &pos);
                SDL_FreeSurface(text_surface);

        }
        else {
                 //error
                cerr << "No se pudo escribir en pantalla: " << SDL_GetError() << endl;
        }
}
```

- **GestorSonido**

Esta es una pequeña clase que se encarga de la gestión del sonido, inicializa y liberar la librería de audio SDL_mixer y los diferentes sonidos que se usaran en el juego. Además proporciona métodos para iniciar y para la música de fondo y para reproducir diferentes efectos de

sonido.

```
GestorSonido::GestorSonido(){

	// Inicializamos SDL_mixer
	if(Mix_OpenAudio(44100, AUDIO_S16SYS, 2, 4096)) {

		cerr << "No se puede inicializar SDL_mixer " << Mix_GetError() << endl;
		exit(-1);
	}
	 // Número de canales simultáneos a usar
	Mix_AllocateChannels(6);

	// Cargamos sonidos
	musicaFondo = Mix_LoadMUS("../sonidos/musicaFondo.ogg");
	if ( musicaFondo == NULL ) {

		cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
		exit(-1);
	}

	salto = Mix_LoadWAV("../sonidos/salto.wav");
	if ( salto == NULL ) {

		cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
		exit(-1);
	}

	impacto = Mix_LoadWAV("../sonidos/impacto_bala.ogg");
	if ( impacto == NULL ) {

		cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
		exit(-1);
	}

	item = Mix_LoadWAV("../sonidos/item_pickup.ogg");
	if ( item == NULL ) {

		cerr << "No pude cargar sonido: " << Mix_GetError() << endl;
		exit(-1);
	}

	canalMusicaFondo = 0;
}

GestorSonido::~GestorSonido(){

	// liberamos recursos
	Mix_FreeMusic(musicaFondo);
	Mix_FreeChunk(salto);
	Mix_FreeChunk(item);
	Mix_FreeChunk(impacto);

	Mix_CloseAudio();
}
```

6. Conclusiones

En cuanto al proyecto en general la conclusión más clara que se saca es que una vez terminado, se tiene una visión más amplia y global del proceso, tanto de la elaboración de un proyecto como del desarrollo de videojuegos a pequeña escala. No cabe duda que de cara al futuro, gracias a la experiencia adquirida, se agilizará el proceso y se conseguirán mejores resultados.

Otra parte importante ha sido todo el trabajo de búsqueda de información y estudio de las herramientas, aunque gracias a la documentación oficial y a la gran cantidad de artículos, blogs y foros siempre se consiguieron resolver los problemas que se presentaban. Con respecto a las herramientas, SDL y las bibliotecas adicionales, el trabajo con éstas no supuso ningún problema, como se ha comentado antes son relativamente intuitivas de usar y sencillas. Como ya se ha visto cumplen su función perfectamente en el desarrollo de videojuegos, aunque uno de los inconvenientes, es que nos encontramos ante una biblioteca de bajo nivel que no está pensada estrictamente para esto, sino para aplicaciones multimedia. Por este motivo el desarrollo puede ser más lento que si se usara algún entorno dedicado a los videojuegos. Habiendo terminado y mirando en perspectiva creo que la elección de usar SDL estuvo bien, ya que se ajustaba a lo que se quería hacer y no se llegó a tener ningún gran problema. Otra conclusión que extraigo es que la elección de una herramienta u otra está muy relacionado con el tipo de proyecto que vayamos a realizar, no por ser más potente o con más características, será siempre la mejor opción la misma herramienta.

El único detalle negativo que se tuvo con la SDL y que supuso un pequeño retraso fue que la biblioteca tal como la bajamos de la red tiene la salida por consola redireccionada a unos archivos de texto, para consultarlos una vez se termine la ejecución. No es un comportamiento habitual y existen varias formas de volverlo al estado natural y tener salida por consola durante la ejecución. El problema fue que los métodos sencillos para revertir éste efecto no funcionaron y se tuvo que ir por el camino difícil, recompilar la biblioteca con una configuración específica, para lo que se tuvo que instalar MSYS en Windows.

Con respecto a los prototipos desarrollados, uno de los objetivos más importantes era conseguir una buena jugabilidad, ya que desde mi punto de vista es uno de los principales factores que posibilitan que un juego triunfe. Se considera que en los dos prototipos se han conseguido buenos resultados, en el caso del Tetris no supuso mucha dificultad, pero con el de plataformas se realizaban continuamente pruebas para que el control del personaje tuviese las mejores características de este tipo de juegos. Este es un aspecto crucial en los videojuegos, por muchos gráficos que se tengan o por muy innovador que sea, si el control sobre el personaje no es bueno o la manera de interactuar con el juego no queda clara, los usuarios no quedaran satisfechos con la experiencia y buscaran otra.

7. Requisitos Hardware Y Software

Requisitos hardware y software necesarios para la realización del trabajo de fin de grado:

En cuanto a los programas usados para la realización de éste trabajo, se ha tenido siempre como objetivo el uso de herramientas libres y gratuitas. Exceptuando el sistema operativo, del que se tiene licencia, el resto de herramientas se encuentran disponibles en la red sin coste.

- Sistema operativo Windows 7.
- Conjunto de bibliotecas SDL 1.2.15 .
- Bibliotecas adicionales SDL_ttf 2.0.11 y SDL_mixer 1.2.12 .
- IDE de trabajo Eclipse IDE for C/C++ Juno Service Release 1.
- Editor de imágenes GIMP 2.8.4 .
- Procesador de texto LibreOffice 4.0.4.2 .
- Implementación de los compiladores GCC para la plataforma Win32 MinGW 4.7.0 .

Una herramienta fundamental para el desarrollo de este trabajo fue el acceso a internet, principalmente para la búsqueda de información y en menor medida para la búsqueda de recursos, imágenes, sonidos, etc... . En cuanto a los requisitos hardware, se contó con un PC de gama media, aunque en este aspecto no hay requisitos específicos.

Requisitos para los usuarios:

Para probar los prototipos es necesario un PC con sistema operativo Windows, se han probado y funciona en Windows 7 y Windows 8.

8. Pliego Condiciones Técnicas

Objeto

El Objeto del contrato, que regularán estas condiciones técnicas, es el desarrollo de dos prototipos de videojuego en 2D. El primero será un clon del conocido juego “Tetris” y el segundo será un videojuego tipo plataformas que cuente con las principales características de este tipo de juegos y además usará un editor externo para la creación de niveles.

Requisitos de fiabilidad

El desarrollo de la aplicación debe emprenderse de tal forma que su resultado garantice alta fiabilidad y comportamiento estable.

Especificaciones técnicas

Los adjudicatarios deben preparar un manual de usuario sin coste adicional alguno, con la suficiente información para que las personas interesadas puedan utilizar la aplicación sin ningún tipo de problema, especificando claramente todos los pasos a realizar en cada punto.

Plazo de Ejecución

El contratista se compromete a elaborar la aplicación de conformidad con lo establecido en este documento y con la oferta que hubiese presentado, así como a entregarlo al contratante. Dicha entrega deberá hacerse en el plazo máximo de 5 meses, a contar desde la fecha de la firma del contrato.

El software se considerará debidamente entregado cuando se haya recibido la plataforma y a su vez esté disponible para su inmediato funcionamiento y se haya facilitado el manual de usuario y la documentación técnica correspondiente.

Confidencialidad de la información

En cumplimiento de lo establecido en el articulo 12 de la Ley Orgánica 15/1999 de 13 de diciembre sobre protección de datos de carácter personal, junto con la formalización del contrato se exigirá la firma por parte del representante legal de la empresa adjudicataria del servicio, de una cláusula de compromiso sobre el tratamiento de datos de carácter personal que pudiera tratar o conocer el personal de su empresa con ocasión del cumplimiento del contrato.

El adjudicatario queda expresamente obligado a mantener absoluta confidencialidad y reserva sobre cualquier dato, especialmente los de carácter personal, que no podrá copiar o utilizar con fin distinto al que figura en este pliego.

Transferencia Tecnológica

Durante la ejecución de los trabajos objeto del contrato, el adjudicatario se compromete a facilitar en todo momento a las personas designadas por la empresa contratante a tales efectos la

información que ésta solicite para disponer de un pleno conocimiento de las circunstancias en que se desarrollan los trabajos, así como de los eventuales problemas que pueden plantearse y de las tecnologías, métodos y herramientas utilizados para resolverlos.

Entrega de la solución

Toda la funcionalidad del sistema debe estar completamente documentada de tal forma que se puedan desarrollar, por una tercera empresa, nuevos aplicativos e incluso nuevas funcionalidades del sistema, que en todo caso serán independientes del software mantenido por los adjudicatarios. El adjudicatario deberá generar la documentación necesaria para asegurar el despliegue exitoso del producto y servicio.

9. Normativa Y Legislación

El delito informático implica actividades criminales que los países han tratado de encuadrar en figuras típicas de carácter tradicional, tales como robos, hurtos, fraudes, falsificaciones, perjuicios, estafas, sabotajes. Sin embargo, debe destacarse que el uso de las técnicas informáticas han creado nuevas posibilidades del uso indebido de las computadoras lo que ha creado la necesidad de regulación por parte del derecho.

9.1 Normativa y regulación de la informática en el ámbito internacional.

En el contexto internacional, son pocos los países que cuentan con una legislación apropiada. Entre ellos, destacan, Estados Unidos, Alemania, Austria, Gran Bretaña, Holanda, Francia, España, Argentina y Chile.

Por esta razón a continuación se mencionan algunos aspectos relacionados con la ley en los diferentes países, así como con los delitos informáticos que persigue.

Estados Unidos

Este país adoptó en 1994 el Acta Federal de Abuso Computacional que modificó al Acta de Fraude y Abuso Computacional de 1986.

Con la finalidad de eliminar los argumentos hipertécnicos acerca de qué es y que no es un virus, un gusano, un caballo de Troya y en que difieren de los virus, la nueva acta proscribe la transmisión de un programa, información, códigos o comandos que causan daños a la computadora, a los sistemas informáticos, a las redes, información, datos o programas. La nueva ley es un adelanto porque está directamente en contra de los actos de transmisión de virus.

Asimismo, en materia de estafas electrónicas, defraudaciones y otros actos dolorosos relacionados con los dispositivos de acceso a sistemas informáticos, la legislación estadounidense sanciona con pena de prisión y multa, a la persona que defraude a otro mediante la utilización de una computadora o red informática.

En el mes de Julio del año 2000, el Senado y la Cámara de Representantes de este país, tras un año largo de deliberaciones, establece el Acta de Firmas Electrónicas en el Comercio Global y Nacional. La ley sobre la firma digital responde a la necesidad de dar validez a documentos informáticos, mensajes electrónicos y contratos establecidos mediante Internet, entre empresas (para el B2B) y entre empresas y consumidores (para el B2C).

Alemania

Este país sancionó en 1986 la Ley contra la Criminalidad Económica, que contempla los siguientes delitos:

- Espionaje de datos.

- Estafa informática.
- Alteración de datos.
- Sabotaje informático.

Austria

La Ley de reforma del Código Penal, sancionada el 22 de Diciembre de 1987,sanciona a aquellos que con dolo causen un perjuicio patrimonial a un tercero, influyendo en el resultado de una elaboración de datos automática a través de la confección del programa, por la introducción, cancelación o alteración de datos o por actuar sobre el curso del procesamiento de datos.

Además contempla sanciones para quienes comenten este hecho utilizando su profesión de especialistas en sistemas.

Gran Bretaña

Debido a un caso de hacking en 1991, comenzó a regir en este país la Computer Misuse Act (Ley de Abusos Informáticos). Mediante esta ley el intento, exitoso o no, de alterar datos informáticos es penado con hasta cinco años de prisión o multas. Esta ley tiene un apartado que específica la modificación de datos sin autorización.

Holanda

El 10 de Marzo de 1993 entró en vigencia la Ley de Delitos Informáticos, en la cual se penaliza los siguientes delitos:

- El hacking.
- El phreaking (utilización de servicios de telecomunicaciones evitando el pago total o parcial de dicho servicio).
- La ingeniería social (arte de convencer a la gente de entregar información que en circunstancias normales no entregaría).
- La distribución de virus.

Francia

En enero de 1988, este país dictó la Ley relativa al fraude informático, en la que se consideran aspectos como:

- Intromisión fraudulenta que suprima o modifique datos.
- Conducta intencional en la violación de derechos a terceros que haya impedido o alterado el funcionamiento de un sistema de procesamiento automatizado de datos.
- Conducta intencional en la violación de derechos a terceros, en forma directa o indirecta, en la introducción de datos en un sistema de procesamiento automatizado o la supresión o modificación de los datos que éste contiene, o sus modos de procesamiento o de transmisión.
- Supresión o modificación de datos contenidos en el sistema, o bien en la alteración del funcionamiento del sistema (sabotaje).

Chile

Chile fue el primer país latinoamericano en sancionar una Ley contra delitos informáticos, la cual entró en vigencia el 7 de junio de 1993. Esta ley se refiere a los siguientes delitos:

- La destrucción o inutilización de los de los datos contenidos dentro de una computadora es castigada con penas de prisión. Asimismo, dentro de esas consideraciones se encuentran los virus.
- Conducta maliciosa tendiente a la destrucción o inutilización de un sistema de tratamiento de información o de sus partes componentes o que dicha conducta impida, obstaculice o modifique su funcionamiento.
- Conducta maliciosa que altere, dañe o destruya los datos contenidos en un sistema de tratamiento de información.

9.2 Normativa y regulación de la informática en el ámbito europeo.

Hasta ahora, el principal esfuerzo europeo por regular el tema de los delitos informáticos dio como resultado el “Convenio sobre la Ciberdelincuencia", de 21 de noviembre de 2001. Este documento fue firmado por los representantes de cada país miembro del Consejo de Europa, aunque su eficacia depende de su posterior refrendo por los órganos nacionales de cada país firmante.

El “Convenio sobre la Ciberdelincuencia" permitió la definición de los delitos informáticos y algunos elementos relacionados con éstos, tales como “sistemas informáticos", “datos informáticos", o “proveedor de servicios".

Estos delitos informáticos fueron clasificados en cuatro grupos:

1- Delitos contra la confidencialidad, la integridad y la disponibilidad de los datos y sistemas informáticos.

- Acceso ilícito a sistemas informáticos.
- Interceptación ilícita de datos informáticos.
- Interferencia en el sistema mediante la introducción, transmisión, provocación de daños, borrado, alteración o supresión e estos.
- Abuso de dispositivos que faciliten la comisión de delitos

2- Delitos informáticos.

- Falsificación informática que produzca la alteración, borrado o supresión de datos informático que ocasionen datos no auténticos.
- Fraudes informáticos.

3- Delitos relacionados con el contenido.

- Delitos relacionados con la pornografía infantil.

4- Delitos relacionados con infracciones de la propiedad intelectual y derechos afines.

Conviene destacar que en el “Convenio sobre la Ciberdelincuencia" se encomienda a cada Parte que tome las medidas necesarias para tipificar como delito en su derecho interno cada uno de los apartados descritos en cada categoría.

En la Disposición 14221 del BOE núm. 226 de 2010, encontramos el Instrumento de Ratificación del Convenio sobre la Ciberdelincuencia, hecho en Budapest el 23 de noviembre de 2001.

9.3 Normativa y regulación de la informática en el ámbito nacional

Este país quizás sea el que mayor experiencia ha obtenido en casos de delitos informáticos, en Europa. Su actual Ley Orgánica de Protección de Datos de Carácter Personal (LOPDCP) aprobada el 15 de diciembre de 1999, la cual reemplaza una veintena de leyes anteriores de la misma índole, contempla la mayor cantidad de acciones lesivas sobre la información.

Se sanciona en forma detallada la obtención o violación de secretos, el espionaje, la divulgación de datos privados, las estafas electrónicas, el hacking maligno o militar, el phreacking, la introducción de virus, etc.; aplicando pena de prisión y multa, agravándolas cuando existe una intención dolosa o cuando el hecho es cometido por parte de funcionarios públicos.

Así mismo su nuevo Código Penal establece castigos de prisión y multas "a quien por cualquier medio destruya, altere, inutilice o de cualquier otro modo dañe los datos, programas o documentos electrónicos ajenos contenidos en redes, soportes o sistemas informáticos".

Norma General

- La Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de carácter personal (**LOPD**). [Documento disponible en pdf (BOE 298 de 14-12-1999) [51]] y [HTML (BOE 298 de 14-12-1999) [52]] Esta Ley Traspone al Ordenamiento Jurídico Español la Directiva Europea 95/46 CE de 24 de Octubre de 1995, relativa a la Protección de las Personas Físicas en lo que respecta al Tratamiento de Datos Personales y a la Libre Circulación de estos Datos .

- El Real Decreto 1720/2007, de 21 de diciembre , por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999 , de 13 de diciembre, Desarrollo de la Ley Orgánica de Protección de Datos de Carácter Personal. [Documento disponible en pdf (BOE 298 de 19-01-2008) [53]] y [HTML (BOE 298 de 19-01-2008) [54]]

Normas Sectoriales

- La Ley 34/2002 , de 11 de julio, de Servicios de la Sociedad de la Información y Comercio Electrónico (**LSSI-CE**). [Documento disponible en pdf (BOE 166 de 12-07-2002)[55]] y

51 https://www.agpd.es/upload/Canal_Documentacion/legislacion/Estatal/Ley%2015_99.pdf
52 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?coleccion=iberlex&id=1999/23750&txtlen=1000
53 http://www.boe.es/boe/dias/2008/01/19/pdfs/A04103-04136.pdf
54 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?coleccion=iberlex&id=2008/979
55 http://www.lssi.es/NR/rdonlyres/A14E0E90-BE74-4CAA-ADF6-

[HTML (BOE 166 de 12-07-2002)[56]].

- Ley 32/2003, General de Telecomunicaciones. (Arts 33-35).

- Ley 59/2003 , de 19 de diciembre, de Firma Electrónica [Documento disponible en pdf (BOE 304 de 20-12-2003)[57]] y [HTML (BOE 304 de 20-12-2003)[58]] .

Otras

- Real Decreto Legislativo 1/1996 , de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual (**LPI**), regularizando, aclarando y armonizando las disposiciones vigentes en la materia. [Documento disponible en HTML (BOE 97 de 22-04-1996)[59]] .

- Real Decreto 3/2010 , de 8 de enero, por el que se regula el Esquema Nacional de Seguridad en el ámbito de la Administración Electrónica .[Documento disponible en HTML (BOE 25 de 29-01-2010 páginas 8089 a 8138)[60]].

- Real Decreto 4/2010 , de 8 de enero, por el que se regula el Esquema Nacional de Interoperabilidad en el ámbito de la Administración Electrónica. [Documento disponible en HTML (BOE 25 de 29-01-2010 páginas 8139 a 8156)[61]].

9.4 Normativa y regulación que afecta al proyecto

A continuación se detallarán las licencias bajo las que se distribuyen las principales herramientas utilizadas en el desarrollo del proyecto.

GNU Lesser General Public License

La Licencia Pública General Reducida de GNU, o más conocida por su nombre en inglés GNU Lesser General Public License (antes GNU Library General Public License o Licencia Pública General para Bibliotecas de GNU), o simplemente por su acrónimo del inglés GNU LGPL, es una licencia de software creada por la Free Software Foundation que pretende garantizar la libertad de compartir y modificar el software cubierto por ella, asegurando que el software es libre para todos sus usuarios.

Esta licencia permisiva se aplica a cualquier programa o trabajo que contenga una nota puesta por el propietario de los derechos del trabajo, estableciendo que su trabajo puede ser distribuido bajo los términos de esta "LGPL Lesser General Public License". El "Programa", utilizado en lo subsecuente, se refiere a cualquier programa o trabajo original, y el "trabajo basado en el Programa" significa ya sea el programa o cualquier trabajo derivado del mismo bajo la ley de derechos de autor: es decir, un trabajo que contenga el Programa o alguna porción de él, ya sea íntegra o con modificaciones o traducciones a otros idiomas.

7C803B150DAA/0/1Ley34_02Consolidado_Enero2008.pdf

56 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?coleccion=iberlex&id=2002/13758
57 http://www.boe.es/boe/dias/2003/12/20/pdfs/A45329-45343.pdf
58 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?coleccion=iberlex&id=2003/23399
59 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?coleccion=iberlex&id=1996/08930
60 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?id=BOE-A-2010-1330
61 http://www.boe.es/aeboe/consultas/bases_datos/doc.php?id=BOE-A-2010-1331

Otras actividades que no sean copia, distribución o modificación no están cubiertas en esta licencia y están fuera de su alcance. El acto de ejecutar el programa no está restringido, y la salida de información del programa está cubierta sólo si su contenido constituye un trabajo basado en el Programa (es independiente de si fue resultado de ejecutar el programa). Si esto es cierto o no depende de la función del programa.

Eclipse Public License

La Licencia Pública Eclipse (EPL) es una licencia de software de código abierto utilizada por la Fundación Eclipse para su software. Sustituye a la Licencia Pública Común (CPL) y elimina ciertas condiciones relativas a los litigios sobre patentes.

La Licencia Pública de Eclipse está diseñado para ser una licencia de software favorable a los negocios y cuenta con disposiciones más débiles que las licencias copyleft contemporáneas, como la Licencia Pública General de GNU (GPL). El receptor de programas licenciados EPL pueden utilizar, modificar, copiar y distribuir el trabajo y las versiones modificadas, en algunos casos están obligados a liberar sus propios cambios.

La EPL está aprobada por la Open Source Initiative (OSI), y aparece como una licencia de "software libre" por la Free Software Foundation (FSF).

General Public License

La Licencia Pública General de GNU o más conocida por su nombre en inglés GNU General Public License (o simplemente sus siglas del inglés GNU GPL) es la licencia más ampliamente usada en el mundo del software y garantiza a los usuarios finales (personas, organizaciones, compañías) la libertad de usar, estudiar, compartir (copiar) y modificar el software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios. Esta licencia fue creada originalmente por Richard Stallman fundador de la Free Software Foundation (FSF) para el proyecto GNU (GNU project).

La licencia GPL puede ser usada por cualquiera, su finalidad es proteger los derechos de los usuarios finales (usar, compartir, estudiar, modificar). Esta es la primera licencia copyleft para uso general. Copyleft significa que los trabajos derivados sólo pueden ser distribuidos bajo los términos de la misma licencia. Bajo esta filosofía, la licencia GPL garantiza a los destinatarios de un programa de ordenador los derechos-libertades reunidos en definición de software libre (free software definition) y usa copyleft para asegurar que el software está protegido cada vez que el trabajo es distribuido, modificado ó ampliado. En la forma de distribución (sólo pueden ser distribuidos bajo los términos de la misma licencia) se diferencian las licencias GPL de las licencias de software libre permisivas (permissive free software licenses), de las cuales los ejemplos más conocidos son las licencias BSD (BSD licenses).

zlib License

La licencia zlib/libpng o simplemente la licencia zlib es una licencia de software libre que define los términos bajo los cuales zlib y libpng pueden ser distribuidos. La licencia es también

usada por otros paquetes de software libre.

Ha sido aprobada por la Free Software Foundation como una licencia de software libre, y por la Open Source Initiative como una licencia de código abierto. Es compatible con la GNU General Public License.

La licencia sólo tiene los siguientes puntos para tener en cuenta:

- Software es usado simplemente como tal. Los autores no son responsables de ningún daño provocado por el uso del mismo.

- La distribución de una versión modificada del software está sujeta a las siguientes restricciones:

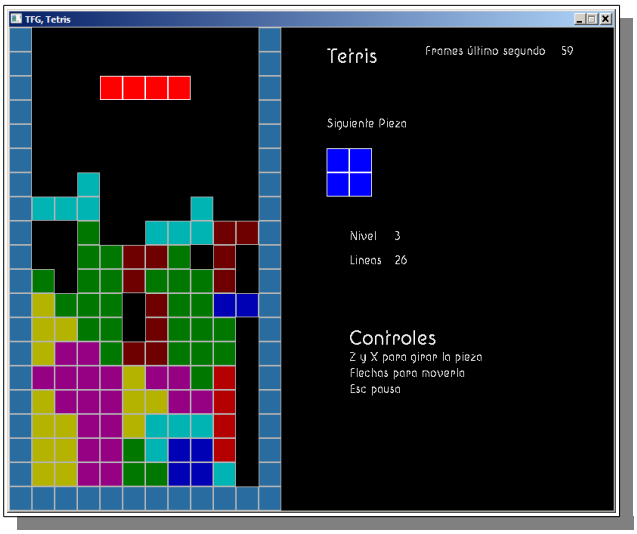
    1. No se debe indicar que se es el autor del software original.
    2. Versiones de fuentes alteradas no deben ser representadas como la versión original del software.
    3. El aviso de la licencia no debe ser eliminado de las distribuciones derivadas.

10. Manual De Usuario

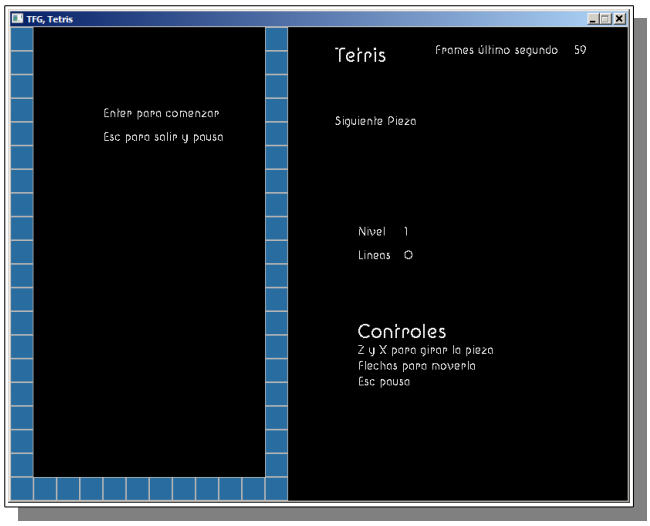
A continuación se explica el funcionamiento de los dos prototipos desarrollados en el trabajo, las características de cada uno, los controles y las diferentes opciones disponibles.

10.1 Tetris

El juego consiste en organizar las diferentes piezas que van cayendo en el tablero, intentando no dejar huecos entre ellas. Cada vez que se completa una línea, se elimina del mapa y aumenta el número de líneas completadas. Cada diez líneas completadas aumenta el nivel en el que nos encontramos, que se refleja en el juego en un aumento de la velocidad a la que caen las piezas.



Captura del juego.



Pantalla inicial.

En cuanto a los controles, con las flechas del teclado podemos mover la pieza hacia la izquierda, hacia la derecha y si pulsamos abajo la pieza caerá más rápido hasta que soltemos esta tecla. Con la tecla "z" giramos la pieza al contrario de las agujas del reloj y con la tecla "x" a favor. El giro hacia un lado u otro sólo se aprecia en algunas piezas. Cuando se inicia el juego, se espera que se pulse "Enter" para comenzar el juego o "Esc" para salir. Una vez iniciada una partida el jugador puede pausar el juego con la tecla "Esc" y reanudarlo desde ese punto con la tecla "Enter" o terminar la ejecución con "Esc". En todo momento podemos encontrar información sobre los controles en la pantalla.

10.2 Plataformas

El objetivo principal es esquivar o matar a los enemigos y superar los diferentes obstáculos para avanzar de nivel, para ello el usuario puede caminar, correr, saltar, subir escaleras y disparar para conseguir llegar a la casilla de salida. Durante su camino podrá recoger sacos de oro y una armadura que le protegerá ante los ataques de los enemigos. Si el jugador es alcanzado por un enemigo o cae en los pinchos morirá. Al ser un prototipo en una versión temprana, se ha añadido una opción para facilitar las pruebas, si pulsamos la tecla "r" durante el juego el personaje recuperará la vida y aparecerá con armadura, aunque hayamos muerto.

En la imagen superior, perteneciente al primer nivel, se pueden observar los diferentes objetos que el usuario puede recoger. También podemos ver uno de los obstáculos , los pinchos azules, que deberá superar el usuario para pasar los niveles y las escaleras pos las que puede subir y bajar. Se aprecia también la casilla de salida marcada con un cuadrado verde a la derecha de la imagen, una vez que el usuario entre en esta casilla el juego avanzará de nivel. Para acabar con los enemigos cuenta con la habilidad de lanzar proyectiles.

Para mover el personaje se usan las flechas del teclado, izquierda y derecha para movernos hacia los lados, arriba y abajo para movernos en las escaleras, espacio para saltar y el "Ctrl" izquierdo para disparar los proyectiles.

Captura del juego en la que se observa al jugador lanzando proyectiles y un enemigo explotando.

11. Bibliografía

Libros

- García Serrano A. . *Programación de videojuegos con SDL.*
- David Brackeen, Bret Barker y Laurence Vanhelsuwé. *Developing Games in Java.*
- Daniel Rodríguez Millán. *Programación de videojuegos en Java.*
- Andrew Davison. *Killer Game Programming in Java*

Sitios web dedicados al desarrollo de videojuegos.

- http://www.gamedev.net/
- http://www.gamasutra.com/
- Wiki sobre desarrollo de videojuegos http://content.gpwiki.org/index.php/Main_Page

Blogs y foros

- Higher-Order Fun Game Design & Game Programming, http://higherorderfun.com/blog/
- Tutoriales para el desarrollo de videojuegos y uso de SDL http://lazyfoo.net/
- Desarrollo de videojuegos y programación http://www.wiredweasel.com/users/serhid/blog/
- Discord Games http://discordgames.com/
- Blog dedicado a los temas legales relacionados con Tetris http://desiree47.wordpress.com/
- Página sobre desarrollo de videojuegos http://gameprogrammer.com/
- realtimecollisiondetection.net - the blog http://realtimecollisiondetection.net/blog/
- Foro oficial SDL http://forums.libsdl.org/index.php
- Foro de desarrollo para NES http://forums.nesdev.com/

Artículos

- Belli, Simone y López, Cristian (2008). Breve historia de los videojuegos. Athenea Digital, 14, 159-179.
- Gran cantidad de artículos relacionados con consolas http://www.museo8bits.com/
- http://psicologiasocial.uab.es/athenea/index.php/atheneaDigital/article/view/570179
- Anuario del Videojuego 2012, Asociación Española de Distribuidores y Editores de Software de Entretenimiento. http://www.adese.es/anuario2012/
- Sonic Physics Guide http://info.sonicretro.org/Category:Sonic_Physics_Guide
- Scrolling http://www.yaldex.com/games-programming/0672323699_ch08lev1sec12.html
- Doing gravity right http://www.niksula.hut.fi/~hkankaan/Homepages/gravity.html
- Instalación de MinGW http://www.mingw.org/wiki/Getting_Started

Documentación SDL

- Documentación oficial SDL http://www.libsdl.org/cgi/docwiki.fcg
- Tutorial muy completo tipo wiki http://wikis.uca.es/wikijuegos/w/index.php?title=Portada

Recursos

Sonidos

- http://www.freesound.org/
- http://soundbible.com/

Gráficos

- http://opengameart.org/
- http://www.spriteland.com/
- http://www.spriters-resource.com/