

UD7 - Videojuegos: motores y desarrollo



Q <https://www.flickr.com/photos/luchored/2452445733>. *videogames* (CC BY)

Parte I - Introducción a los motores de juegos.



Caso práctico

A Juan, el programador de BK Programación que ha estado aprendiendo a desarrollar Aplicaciones Android contigo, le acaba de llegar una oferta de trabajo para empezar a programar en una empresa que se dedica al desarrollo de videojuegos.

Juan no tiene experiencia en el mundo del videojuego, sin embargo, le gusta jugar a algunos videojuegos y sin duda es un tema que le apasiona. Tras realizar la primera entrevista con la empresa que le ha hecho la entrevista ha descubierto varias cosas:

- El desarrollo de software orientado a los videojuegos es un nicho importante de mercado en España, alcanzando altos volúmenes de negocio y empleando en España a más de 8.000 profesionales.
- El desarrollo de los videojuegos no tiene nada que ver con jugar a videojuegos. Es una industria que exige mucho a sus desarrolladores, por los continuos cambios de la tecnología y por las fuertes inversiones que se realizan en el desarrollo que esperan retornar beneficios.
- Existen muchos tipos de videojuegos, algunos de ellos no son lúdicos, sino que buscan otros objetivos. Son los conocidos como "**serious games**" empleados en asuntos tan importantes como la Educación, el Turismo, la Cultura o la Sanidad.



Q Soumil Kumar

(<https://www.pexels.com/es-es/foto/foto-de-persona-escribiendo-en-el-teclado-de-la-computadora-735911/>)
(Pexels)



Q [Ministerio de Educación y Formación Profesional.](#)

(Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

1.- Introducción a los videojuegos.

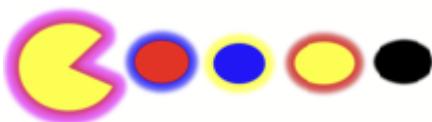


Caso práctico



Juan tiene muchas dudas a la hora de decidir si acepta o no su nueva oferta de trabajo: por un lado, le apasiona el sector de los juegos, la nueva oferta es buena y no quiere dejar pasar una oportunidad importante de trabajo; por otro lado, sigue aprendiendo muchas cosas en su actual empresa, donde se encuentra cómodo y valorado.

Así que antes de tomar la decisión de salir de su zona de confort va a estudiar un poco mejor la situación actual del mercado de los videojuegos en España. Para ello ha descubierto que hay una asociación [DEV](#) que publica anualmente información al respecto donde consigue ver datos muy interesantes del sector, que le animan cada vez más a dar el salto. Además, está investigando en Internet sobre cómo se organizan los equipos de trabajo que desarrollan los videojuegos para entender mejor su papel como desarrollador.



Q Moeez (CC BY-SA)

Todos tenemos una idea intuitiva más o menos clara de lo que es un videojuego. Como puedes deducir, hoy en día un videojuego no deja de ser un tipo de aplicación más que puede ejecutarse en un ordenador, dispositivo móvil o consola.

Podemos definir un videojuego como un juego electrónico que interactúa con uno o varios jugadores con el fin de que se consigan ciertos objetivos y que muestra visualmente una respuesta a sus acciones.

Un videojuego tiene que informar mediante algún mecanismo al jugador o jugadores del resultado de sus acciones, ya sea visualmente, acústicamente, mediante vibraciones, etc. Normalmente, esa realimentación se realiza mediante la proyección de una imagen de vídeo en una pantalla, de ahí el nombre: videojuegos.

Puede llamarte la atención que no hayamos incluido el requisito de que sea entretenido o divertido. Eso es correcto, aunque es deseable, un videojuego no tiene por qué ser divertido de jugar al igual que ocurre, por ejemplo, al ver una película. Al cine se puede ir a pasar miedo (películas de terror), a ver historias tristes (dramas) o para estar en tensión constante (acción). ¿Quiere decir que esas películas no son buenas? Nada más lejos: lo importante es que sea una experiencia agradable, que te deje con ganas de repetir. Lo mismo ocurre con los videojuegos, como veremos más adelante cuando analicemos los géneros.

1.1.- Orígenes de los videojuegos.

Es difícil poner una fecha exacta al origen de los videojuegos, pero podemos comenzar en 1947. Fue en ese año cuando se registró una patente en Estados Unidos acerca de un dispositivo de entretenimiento que consistía en disparar a aviones en una pantalla de tubo de rayos catódicos (CRT). Thomas T. Goldsmith, Jr. y Estle Ray Mann aparecen como sus autores. Bien pudieron ser los padres del concepto de videojuego.

El **Nimrod**, construido en 1951, se considera primer ordenador diseñado exclusivamente para jugar al **Nim**, un juego donde jugador y máquina se alternaban para retirar palitos de unos montones.

Poco después, en 1952, Alexander S. Douglas programó una versión del tres en raya (**Noughts and Crosses**) que se ejecutaba en el ordenador **EDSAC** de la Universidad de Cambridge en 1952. No tuvo mucha repercusión dado que solamente podía ejecutarse en el ordenador de dicha universidad.



Q [Bumm13](#) (Dominio público)

William Higinbotham podría considerarse como el creador del videojuego multijugador. Su "**tenis para dos**" o "**pong**" permitía a dos jugadores competir en el mismo juego. Se exhibió públicamente durante unas jornadas de puertas abiertas celebradas en su empresa. Aunque fue disfrutado por cientos de personas, no llegó al público adecuado y por tanto no condicionó la aparición de otros videojuegos.

En 1961, un estudiante del MIT de nombre **Stephen Russell** desarrolló un videojuego para dos jugadores llamado **Spacewar!** que funcionaba en ordenadores PDP-11. Posteriormente el juego fue mejorado por otros compañeros de Stephen, distribuyéndose como software de dominio público a través de la ARPAnet, la antecesora de Internet.

Pero no fue hasta comienzos de los años 70 cuando se extendió el uso de los videojuegos en las casas y en los salones de máquinas recreativas. Los responsables fueron **Ralph Baer** y **Nolan Bushnell**, el primero mediante el diseño y la comercialización de la primera consola de videojuegos: la **Magnavox Odyssey**, y el segundo por la fundación de una de las empresas de videojuegos más influyentes de la historia: **Atari**. Esta compañía creó innumerables máquinas recreativas y la primera consola que fue un éxito de ventas: la **Atari 2600**.



Q [computerhistory.org](#) (Uso educativo nc)

A partir de ahí, todo fue historia, marcándose mucha diferencia entre juegos de generaciones diferentes separados por apenas 10 años, lo cual nos da una idea de la velocidad con la que evoluciona la tecnología de los videojuegos.

Puedes echarte una partida al Spacewar! original de Stephen Russell si sigues este [enlace al simulador del juego en el PDP-1](#).

1.2.- Los videojuegos en la actualidad.

Mucho ha cambiado en el campo de la informática y la tecnología en general desde los tiempos de esos primeros videojuegos. Los de hoy en día pueden llegar a ser auténticas producciones multimillonarias que superan en presupuesto a las grandes películas de Hollywood e involucran a cientos de personas.

Hoy en día, la industria de los videojuegos está centrada en tres grandes plataformas: los ordenadores personales, las consolas y los dispositivos móviles. Hace unos años tendríamos que haber mencionado las máquinas recreativas, pero la democratización de los ordenadores y las consolas han terminado por relegarlas prácticamente al olvido.

Por lo general, los saltos tecnológicos aparecen primero en los ordenadores ya que están en permanente evolución y sus posibilidades de expansión permiten la inclusión de nuevos tipos de dispositivos. Eso sucedió en su momento con la introducción de las tarjetas de sonido, los gráficos 3D o las tarjetas de aceleración de cálculos físicos. Las novedades más demandadas pasan poco después al mundo de las consolas. A su vez, los dispositivos móviles son cada vez más potentes y ya tienen en muchos casos las mismas características técnicas y potencia que los otros dispositivos antes mencionados.

Programar para una de estas plataformas implica una serie de ventajas y de inconvenientes respecto a las demás. Veamos algunos aspectos interesantes:

- **Consolas:** Son dispositivos diseñados exclusivamente para ejecutar videojuegos. Su hardware evoluciona con cada iteración (que en ellas se llama "generación"), lo que suele suceder cada varios años. Generalmente, las capacidades de una consola en el momento del lanzamiento son punteras, aunque permanecen constantes durante la vida del producto. Eso implica que con el paso del tiempo se quedarán obsoletas, dando paso a una nueva generación. La gran ventaja desde el punto de vista de la programación de videojuegos es que el hardware no varía durante este tiempo lo que permite concentrar los esfuerzos en una única plataforma. De cara al usuario final, su uso es mucho más simple que el de un ordenador personal. La generación actual de consolas a la fecha de escribir este texto está formada por la Sony PlayStation 5, la Xbox Series S y la Nintendo Switch.
- **Ordenadores personales:** Dado que son dispositivos de propósito general, los ordenadores personales han sido uno de las primeras plataformas en permitir la ejecución de juegos. Dado que continuamente están saliendo al mercado nuevos procesadores, tarjetas gráficas, memorias, etc., se produce un crecimiento firme en las capacidades de los ordenadores. Por ello, los equipos de última generación casi siempre van a superar a las consolas en potencia y memoria ya que adaptan las nuevas tecnologías con mayor velocidad. La cruz de la moneda es que los videojuegos tienen que soportar un abanico muy amplio de hardware ya que las configuraciones de dos ordenadores personales pueden ser muy distintas y eso necesita una inversión extra de tiempo y dinero durante el desarrollo.



Q Libro Blanco DEV 2021 (Uso educativo nc)

- **Dispositivos móviles:** Los últimos en llegar han sido los dispositivos móviles. Si bien hace años que existen las videoconsolas portátiles (Nintendo Gameboy, Atari Lynx, Sega GameGear, Nintendo DS, Sony PSP, etc.) el concepto ha evolucionado para dar soporte a teléfonos móviles, tablets y a otros dispositivos similares (por ejemplo, reproductores de música). Concretamente, la aparición de los smartphones con grandes capacidades gráficas y con un sistema operativo de grandes prestaciones (iOS, Android, etc.) ha disparado el consumo de videojuegos en este tipo de dispositivos.

1.3.- Clasificación de los videojuegos.

No existe una clasificación unificada de los géneros de videojuegos, ni siquiera los expertos se ponen de acuerdo. De hecho, lo normal es que un juego moderno sea una mezcla de más de uno. Muchos de estos géneros han surgido como respuesta a un juego innovador que en su momento sentó escuela y cuya idea fue repetida y/o mejorada en desarrollos posteriores. A continuación, vamos a describir algunos de los más importantes, resaltando algunos de los juegos que dieron paso al género o bien supusieron su popularización:

- **Acción:** Los videojuegos de acción son aquellos donde se realizan movimientos y/o combates rápidos. Aquí podemos incluir los juegos de lucha, simulaciones de combate, juegos de disparo en primera persona, etc. Como representantes de los primeros juegos de acción podríamos nombrar Pong, Asteroids y Space Invaders.
 - ○ **Lucha:** En ellos, el jugador pelea contra otros jugadores o bien contra personajes controlados por el videojuego. Eso incluye los juegos de lucha uno contra uno o los juegos cooperativos donde el jugador o jugadores se enfrentan a hordas de enemigos mientras avanzan por un nivel. Como ejemplos podríamos nombrar: Double Dragon, Street Fighter, Mortal Kombat y Virtual Fighter.
 - ○ **Videojuegos de disparo en primera persona (FPS: First Person Shooter):** Los FPS son juegos donde la pantalla representa lo que vería el personaje a través de sus ojos. En ellos, el personaje suele ir fuertemente armado y dispara a los enemigos mientras recoge ítems desperdigados por el nivel. Los más conocidos fueron: Wolfenstein 3D, Doom, Duke Nukem 3D y Quake.
- **Aventura:** Son aquellos que sumergen al jugador en una historia que se va desarrollando conforme avanza en el juego. Suelen incluir puzzles y otros desafíos que se superan con astucia e intuición. Las más antiguas no tenían gráficos y funcionaban mediante descripciones del entorno en forma de texto. Entre ellas destacan Zork (en modo texto) y las series: King's Quest y Monkey Island. Se incluyen en este tipo los juegos de mundo abierto como la saga Final Fantasy, Kingdom Hearts, Zelda o Dragon Quest.
- **Plataformas:** Este tipo de juegos suele consistir en un personaje, controlado por el jugador, que avanza por un nivel mediante movimientos rápidos y saltos. Los niveles suelen recorrerse en horizontal o vertical. Se podría considerar Super Mario Bros y sus variantes. como el iniciador del género al que siguieron otros éxitos como Bubble Bobble o Rainbow Island.
- **Arcade:** El juego se divide en diferentes pantallas o niveles que cada vez presentan más dificultad. Para ello el jugador debe demostrar mayor destreza cada vez y puede necesitar resolver puzzles, laberintos y enigmas. Pac-Man, Tetris y SuperPang se incluyen dentro de esta categoría.
- **Estrategia:** Son juegos donde la planificación de las acciones suele tener bastante peso a la hora de obtener una victoria. Podemos clasificar los juegos en dos subgéneros: estrategia en tiempo real y estrategia basada en



Q Autor desconocido (Uso educativo nc)

turnos. La diferencia principal entre ambas es el tiempo de reacción ante los sucesos, ya que el juego no se detiene en el primer caso, obligando a realizar acciones de forma rápida. Uno de los juegos que sentó las bases de la estrategia en tiempo real fue The Ancient Art of War, aunque el género fue popularizado por Dune 2, Warcraft y Command & Conquer. En el caso de la estrategia por turnos Civilization fue el gran iniciador, al que siguieron otros títulos célebres como Master of Orion o X-Com. Otros destacados serían Fire Emblem y League of Legends.

- **Puzzles:** Estos juegos permiten al jugador aplicar estrategias y deducciones lógicas con el fin de conseguir ciertos objetivos. Suelen trabajar mucho con formas geométricas y combinaciones de elementos y colores que pueden ser movidos, intercambiados o rotados. También pueden incluir un tiempo limitado o bien introducir una competición con otro jugador para darle más emoción. El caso más claro de este tipo de juegos es Tetris (ya mencionado en los juegos de Arcade).
- **Simuladores:** Como su nombre indica, intentan recrear un sistema del mundo real. Desde el funcionamiento de una ciudad (SimCity), acciones cotidianas (la saga de Los Sims o Nintendog, que acompañó el lanzamiento de la Nintendo DS), simulación de un avión (Microsoft Flight Simulator), de un negocio de transportes (Transport Tycoon), de una nave espacial (X-Wing) o de un coche de carreras (eaSport F1).
- **Juegos de rol (RPG: Role Playing Game):** Es la versión electrónica de los juegos de rol tradicionales que se juegan con papel y dados. Los jugadores tienen unas estadísticas (fuerza, sabiduría, dinero, etc.) que se van incrementando conforme aumenta su experiencia. Además, pueden adquirir y usar objetos y armas que van recopilando durante el juego. Al contrario que sus equivalentes en papel, aquí no es necesario echar a volar la imaginación para situarse en el juego; además, las reglas están implícitas, lo que evitar tener que memorizarlas. Las series Ultima, Baldur's Gate o Final Fantasy son algunos ejemplos de este tipo de videojuegos.

- ○ **Juegos de rol multijugador masivos (MMORPG: Massively**

Multiplayer Online Role Playing Game): Son juegos de rol cuyo principal interés es que el desarrollo de la partida se produce en un escenario donde coinciden cientos o miles de otros jugadores. En sus orígenes eran juegos de texto y se llamaban MUD (Mazmorras multiusuario). Suelen ser juegos en los que se paga una suscripción mensual para acceder. Algunos ejemplos representativos son: Ultima Online y World of Warcraft. La saga de Assassin's Creed es un buen ejemplo de este tipo de formatos.

- **Carreras:** El objetivo de estos juegos es llegar el primero a la meta. En muchos de ellos se intenta recrear lo mejor posible las sensaciones de conducir un vehículo en una competición. El primero en convertirse en éxito de ventas fue Pole Position, eaSport F1, etc.
- **Deportivos:** Estos videojuegos tratan de transmitir la emoción de practicar un deporte real. Esto incluye desde deportes individuales, a dobles o en equipo. El juego Summer Games o las series PC Fútbol, Pro Evolution Soccer, FIFA, NBA2KXX.
- **Musicales:** El jugador tiene que seguir o acompañar el ritmo dado por el juego ya sea mediante la pulsación de botones, movimientos o entonación al

cantar. El juego clásico es el Dance Dance Revolution, donde el jugador tiene que saltar sobre unas flechas que hay en el suelo siguiendo las indicaciones de la pantalla. Otros juegos que popularizaron el género son las series Singstar, Guitar Hero y Rock Band.

- **Minijuegos:** Son muchos juegos en uno, generalmente de muy corta duración y con unas reglas simples. Pueden ser parte de otro tipo de juego, ofreciéndose los minijuegos como una recompensa o como un requisito para seguir avanzando. En otros casos, el juego en sí consiste en la superación de un conjunto de minijuegos (por ejemplo, Mario Party).

- **Tradicionales:** Los equivalentes electrónicos de los juegos de toda la vida. El juego Noughts and Crosses (OXO) puede considerarse uno de los primeros juegos de este género, pero aquí también se incluyen los juegos de cartas, de mesa (ajedrez, damas), etc.

- **Educativos:** Su misión es que el jugador aprenda mientras se divierte. El aprendizaje puede producirse mediante la necesidad de memorización de ciertos datos para conseguir cierto objetivo o bien mediante la observación y repetición de ciertas tareas o comportamientos. Algunas de las series más conocidas fueron la de Carmen Sandiego y la saga The Incredible Machine.

- **Serios:** Los juegos serios son un género muy importante, de hecho, se calcula que en 2021 el 21% del total de juegos pertenecen a esta categoría. Están diseñados no solo para entretener sino para que al jugar repetidamente se adquieran ideas que entran dentro del ámbito de la ética, la realidad social o el trabajo en equipo. La gamificación llega con fuerza a sectores como la educación, la sanidad y el turismo.



Q Poiuyt Man (Dominio público)

1.4.- La industria del videojuego.

El hecho de que un usuario final disfrute de un videojuego es el resultado del esfuerzo de muchos elementos que han de trabajar coordinados. Lo primero que podemos preguntarnos es ¿de dónde sale la inversión inicial para pagar el desarrollo? Normalmente procede de un editor o productor que adelanta parte del dinero que espera recibir por las ventas del videojuego. Dado que es la entidad que paga, el editor suele tener gran poder de decisión en el desarrollo.

El desarrollador (puede ser una empresa o un particular) es el encargado de diseñar e implementar el videojuego. Suele estar especializado en unos determinados géneros y/o plataformas (por ejemplo, juegos de rol multijugador en ordenadores personales). Respecto al desarrollador podemos distinguir tres casos según los proyectos que desarrollean:

- Un desarrollador interno es el que trabaja siempre para el mismo editor o fabricante de hardware.
- Un desarrollador de terceros firma contratos con los editores para llevar a cabo un proyecto concreto en particular. De hecho, la misma empresa desarrolladora puede tener a la vez proyectos con distintos editores.
- Un desarrollador independiente es un equipo pequeño, puede que incluso unipersonal. Suele autopublicar sus propios juegos y, al no estar atado a un editor, tienen plena libertad creativa. Eso permite que surjan ideas innovadoras que nunca recibirían dinero de un editor que las percibiría como una inversión arriesgada. La falta de recursos económicos suelen suplirla con soluciones y productos ingeniosos. A la hora de escribir estas líneas hay un repunte de este tipo de equipos debido a que los costes de distribución que existen en los dispositivos móviles son bastante bajos.

La etapa de distribución es la encargada de hacer llegar el videojuego a las estanterías de las tiendas o a las tiendas de descargas digitales. Es aquí donde se genera el beneficio para el desarrollador y/o el editor. Hay que tener en cuenta que el distribuidor se lleva parte de esas ganancias.



Q Ministerio de Educación (Uso

educativo nc)



Para saber más sobre la situación actual de la industria en España te recomendamos que eches un ojo al interesante

(<https://dev.org.es/images>

/stories

/docs/libro%20blanco%20del%20desarrollo%20espanol%20de%20videojuegos%202021.pdf)

(Uso educativo nc)

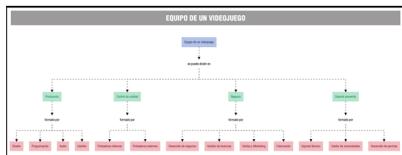
 Libro Blanco del Desarrollo Español de Videojuegos 2021 .

Presentado en mayo de 2022, promovido por DEV, la Asociación española de empresas productoras y desarrolladoras de videojuegos y software de entretenimiento, y apoyado por ICEX España (Exportación e Inversiones), el Libro Blanco se dirige a los estudios desarrolladores, a los profesionales actuales y futuros del sector, a las entidades públicas y, también, a inversores privados nacionales e internacionales, siendo una inmejorable herramienta para conocer en profundidad la industria y el mercado del videojuego en España y recabar toda la información posible para la toma de decisiones y para poder elaborar los planes de inversión y apoyo público de cara a los próximos años.

El videojuego es un sector en constante crecimiento que proporciona grandes oportunidades de negocio.

1.5.- El equipo de desarrollo.

Si bien en los años 70 y 80 los videojuegos eran diseñados e implementados por una persona o un grupo reducido, **hoy en día lo normal es que un proyecto se lleve a cabo por decenas de personas**. Como se puede deducir, en ellos no solamente hay programadores, artistas y diseñadores gráficos, sino que también intervienen editores, productores, distribuidores, escritores, probadores, publicistas, gestores económicos, soporte técnico, etc. Tan solo tienes que fijarte en los títulos de créditos de un videojuego moderno, ¡aparece tanta gente como en una película!



Q. Ministerio de Educación (Uso educativo nc)

Observa en la imagen el gráfico sobre la estructura de un equipo de desarrollo derivado del propuesto en "Game Development and Production" de **Erik Bethke**.

Todas esas secciones forman el equipo típico de desarrollo de un videojuego moderno de gran presupuesto.

Cada persona que participa posee un área de especialización, aunque no siempre la aplica trabajando en lo más obvio. Por ejemplo, un programador puede estar echando una mano en gestión de calidad o en soporte técnico ya que sus conocimientos benefician el funcionamiento de esa parte del proyecto.

En el caso de la producción, podemos dividir al personal en otros grupos según la función que desempeñan: diseño general, programación, diseño artístico o 3D, audio y gestión. En este módulo profesional nos centraremos en la parte de producción, y dentro de ella en los equipos de diseño y programación, especialmente en esta última.

La parte del equipo dedicada al diseño son las personas encargadas de determinar la idea, la mecánica del juego, los diseñadores de niveles o de misiones y los escritores de la historia y los diálogos. Suelen incluir un diseñador principal (en inglés, "Lead Designer"), que es quien coordina a los demás.

Por otro lado, y dado que el producto final es software, debe haber programadores que se encarguen de la creación del código específico, del motor de juegos y de todas las demás herramientas relacionadas (editores de niveles, instaladores, etc.). Los equipos de programadores suelen construirse alrededor del programador principal (en inglés: "Lead Programmer"), que suele ser el más experimentado. Si el tamaño del equipo es grande, puede aparecer el rol de director técnico ("Technical Director"), que tiene un papel menos directo en la programación porque dedica tiempo a gestionar el equipo; si existe, es él quien se comunica directamente con el director de proyecto.



Autoevaluación

El tipo de segmento para el que más videojuegos se producen en la actualidad es el de los juegos para consola. ¿Verdadero o Falso?

Verdadero Falso

Falso

Falso, es el segmento de los SmartPhone Games.

2.- Motores de juegos.



Caso práctico

El siguiente paso que ha dado Juan para valorar su nueva oferta, tras haber estado investigando estos días sobre cómo se desarrollan los videojuegos es conocer qué es y cómo funcionan los llamados "motores de videojuegos". Son un software de diseño y desarrollo que facilita la creación de los mismos.

Juan ya tiene una idea y quiere programar un juego, pero se pregunta ¿por dónde empiezo? ¿Qué componente es el primero sobre el que debo trabajar? Pues dado que la elección del motor de juegos determinará cómo será el código específico y el formato de los recursos (dos de los componentes principales en la arquitectura de un videojuego) lo lógico es comenzar seleccionando un motor de juegos. Es una decisión que no debe tomarse a la ligera ya que cambiar el motor de juego a mitad de un proyecto puede tomar mucho tiempo. Algunos de ellos son gratuitos, pero hay diversos aspectos a tener en cuenta.

En este caso Juan se decanta por intentar trabajar con uno de los motores de videojuegos que la empresa que le ha hecho la oferta está usando actualmente. De esta forma se hará una idea más real de cómo será su nuevo trabajo, si es que decide aceptarlo. Es sorprendente la cantidad de motores que existen actualmente.

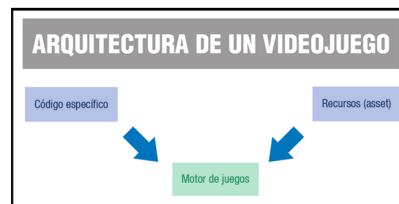


Q Suludan Diliyaer

(<https://www.pexels.com/es-es/foto/controlador-de-juego-led-en-la-mesa-596750/>) (Pexels)

Hoy en día la gran mayoría de los videojuegos presentan tres claros componentes a grandes rasgos: **el código específico del juego, el motor de juegos y los recursos**.

- El **código específico** es donde se implementa la lógica de ese juego en particular, es decir, el sitio donde se definen las reglas, el comportamiento de los elementos del juego (por ejemplo, la pelota, los enemigos, etc.), cómo se reacciona ante las posibles acciones del jugador, las condiciones para ganar, etc.
- El **motor de juegos** contiene la funcionalidad sobre la que se apoya el código específico para realizar esas actividades. Como veremos más adelante, un buen motor de juegos es independiente del tipo de juego que se está implementando y sólo incluye conceptos genéricos como la captura de eventos del teclado/ratón, la gestión de recursos, la visualización de los objetos, las comunicaciones por red, la reproducción de sonidos, etc. El motor de juegos nos permite abstraernos del hardware del equipo y el sistema operativo, aunque usan sus servicios para llevar a cabo sus funciones. Esto es muy interesante, veamos la razón con un ejemplo: si



Q Ministerio de Educación (Uso educativo nc)

nuestro motor de juegos soportara tres clases de consola y tres sistemas operativos distintos de ordenadores personales, podemos tener un juego funcionando en nueve plataformas con muy poco esfuerzo adicional.

- El tercer componente, los **recursos** (o **assets**, en inglés), es el compendio de los contenidos del juego. Entre esos contenidos podemos encontrar música, efectos de sonido, modelos 3D, fondos de pantalla, tipos de letra, niveles, etc. El motor de juegos cargará los recursos según le indique el código específico. De la creación de los contenidos suelen encargarse los diseñadores artísticos, ingenieros de sonido, escritores, diseñadores de niveles, etc., aunque su desarrollo puede estar supervisado por programadores.

En los siguientes apartados haremos una revisión de los distintos componentes que tiene un motor de juego típico y veremos cuáles son sus características más importantes.

2.1.- Clasificación de motores de juegos.

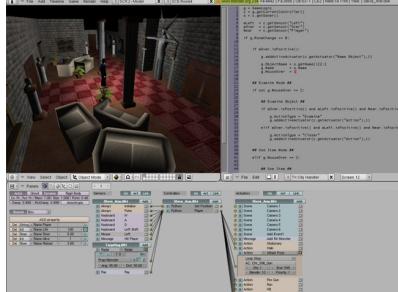
Un motor de juegos expone una interfaz de programación de aplicaciones (API) que permite al código específico utilizar su funcionalidad. Sin embargo, no todos los motores ofrecen las mismas características.

Según el nivel de abstracción cubierto por el motor de juegos, podemos distinguir tres tipos de complejidad creciente:

- Motores diseñados únicamente para facilitar la representación en pantalla y el acceso al hardware en general (audio, dispositivos de entrada, red, etc.). De hecho, muchos de ellos son considerados como librerías más que como motores de juegos. Algunos ejemplos: **SDL**, **LWJGL** o **Allegro**.
- Motores que, además de lo anterior, añaden un soporte total o parcial a la lógica del juego. Podemos destacar: **JGame** y **Ogre3D**.
- Motores que, además de todo lo anterior, incluyen plataformas para la creación, modificación o integración de contenidos. Algunos de ellos: **Blender Game Engine**, **jMonkeyEngine** o **XNA Game Studio**.

Otro aspecto a considerar es el del coste económico y de la licencia de uso. Para proyectos personales o no comerciales es posible encontrar motores que no nos supongan ninguna inversión, lo que puede ser determinante. La licencia de uso de otros motores puede impedirnos vender el resultado final o limitar el número de unidades vendidas antes de tener que pagar.

Teniendo todo esto en cuenta, podemos clasificar los motores en dos grandes grupos según el tipo de licencia de uso:

- **De código abierto:** No es necesario pagar por su uso e incluyen el código fuente del motor. Hay que prestar atención al tipo de licencia concreto, pues algunas restringen el uso del motor a la creación de videojuegos que también sea software libre (por ejemplo, la licencia GPL). La documentación y el soporte suele ser llevado a cabo por la comunidad de usuarios.

Q: [blender community \(GNU/GPL\)](#)
- **Propietario:** Son desarrollados por empresas que cobran por la comercialización de juegos basados en sus motores. En algunos casos pueden ofrecerse de forma gratuita si se cumplen determinadas condiciones. El coste final puede depender de muchos factores: número de empleados, de copias vendidas o del dinero entregado por los editores para su realización. Suelen estar excelentemente documentados y, por lo general, incorporan herramientas adicionales que facilitan la gestión de los contenidos. Las empresas ofrecen normalmente asistencia técnica de sus productos. Por todo ello, las grandes producciones suelen basarse en este tipo de motores de juego. A la hora de escribir estas líneas, los más reconocidos son **Construct 3**, **Gamebryo**, **Gamemaker Studio**, **GameSalad**, **HeroEngine**, **Leadwerks**, **RPG Maker** y **Unity**.

Los motores de uso gratuitos (que no libres) en este momento son los siguientes: **Amazon Lumberyard**, **Arcade Game Studio**, **Cocos**, **CryEngine**, **GDevelop**, **Godette (antes Godot)**, **HaxeFlixel**, **jMonkey**, **Mugen**, **Panda3D**, **Phaser**,

Ren'Py, Scratch, Solar2D, Source 2, Stencyl, Unreal Engine.

2.2.- Programación de un motor. APIs básicas.

Cada motor soporta solamente determinados lenguajes de programación para el código específico del juego, así que debemos elegirlo con cuidado. Es importante destacar que algunos motores usan un lenguaje propio para especificar el comportamiento del juego. El cómo se integra el código específico del juego con el motor puede variar considerablemente de un motor a otro, ya que no todos los motores comparten la misma estructura interna ni filosofía de programación. La mejor forma de empezar es estudiar los tutoriales que suelen incluirse con el motor.

Un motor de juegos está especializado solamente en determinados aspectos, no todos incluyen las mismas características. De hecho, es posible combinar distintos motores de manera que cada uno aporte parte de la funcionalidad necesaria para completar los requisitos del juego. En esos casos, el código específico se comunicará con ambos motores para integrarlos.

Muchos motores de juegos se apoyan en librerías ya existentes que les proporcionan parte de su funcionalidad básica. Por ejemplo, Vulkan o DirectGraphics suelen utilizarse para mostrar objetos en 2 y 3 dimensiones, siendo estas librerías las que acceden directamente a los controladores de la tarjeta gráfica. Esto permite al motor no estar atado a un determinado hardware. Más aún, algunas de esas librerías están disponibles para múltiples plataformas (por ejemplo, Vulkan) lo que facilita la portabilidad del motor a múltiples sistemas. Las plataformas destinos son sin duda uno de los parámetros a repasar antes de elegir el motor.

Las librerías no tienen por qué ser exclusivamente para acceder al apartado gráfico: existen muchas otras librerías para otros usos. Por ejemplo, OpenAL o FMOD hace lo mismo con el audio y OpenCL con las operaciones de computación intensiva.



Para saber más

En la versión inglesa de la Wikipedia hay una comparativa de motores de juegos de gratuitos y de código abierto: [Comparativa de motores de juegos gratuitos y de software libre](#). 

2.3.- Ventajas de la utilización de motores.

¿Qué pasaría si no usáramos un motor de juegos? De hecho, los primeros videojuegos no los utilizaban. Cuando esto ocurre, el código específico es el encargado de todas las tareas: comunicarse directamente con el sistema operativo y el hardware, dibujar en la pantalla, interceptar los dispositivos de entrada, conectarse con otras instancias a través de la red en el caso multijugador, implementar la "inteligencia" de los enemigos, simular la gravedad, detectar cuando dos objetos colisionan, etc.

A priori, puede parecer una idea buena porque sólo se implementa aquello que se va a necesitar, con lo que el resultado será más óptimo. Además, el código estará más integrado.

Imaginemos que realizamos el proyecto así. **¿Qué problemas podríamos encontrarnos?** Analicemos algunos escenarios posibles:

Ventajas del uso de un motor de juegos		
Escenarios	Sin un motor de juegos	Con un motor de juegos
El juego tiene tanto éxito que nos encargan una continuación.	Tendremos que revisar todo el código del proyecto anterior y diferenciando lo que nos sigue sirviendo de lo que no. El motivo es que no hay una separación clara entre lo genérico y lo específico del juego.	El código del proyecto ya está separado en código específico del juego y el motor (código genérico).
Hay que migrar el juego a una nueva plataforma.	<p>No nos queda más remedio que revisar todo el código del proyecto anterior y analizar qué parte es dependiente de la plataforma antigua para sustituirlo por código nuevo. Por supuesto, tendríamos que tener cuidado de no estropear nada.</p> <p>Es posible que los recursos ya no sean válidos para la nueva plataforma y haya que adaptarlos.</p> <p>Además, si detectamos un error en el juego o mejoramos alguna herramienta habrá que</p>	En el momento que el motor de juegos soporte la nueva plataforma bastará con recompilar el proyecto.

Ventajas del uso de un motor de juegos

Escenarios	Sin un motor de juegos	Con un motor de juegos
	cambiar el código por separado en ambos proyectos.	
Llega un nuevo programador a la empresa y queremos formarlo para trabajar en nuestros proyectos.	Habrá que explicarle la estructura del código, que será distinta para cada proyecto en el que vaya a trabajar y plataforma soportada.	Bastará con enseñarle a usar el motor de juego, que será el mismo en muchos proyectos. Luego se le señalarán las particularidades de cada proyecto concreto en el que vaya a trabajar.
Queremos incorporar a nuestro juego el efecto gráfico X de un juego de la competencia.	Será necesario investigar cómo funciona la característica X, implementarla en todas las plataformas del proyecto y hacer las pruebas correspondientes antes de usarla.	Seguramente haya un motor de juego que implemente la característica X de serie en todas las plataformas, con lo únicamente tendremos que usarla.

2.4.- Componentes de un motor de juegos.

Como hemos visto antes, no todos los motores de juegos tienen las mismas funcionalidades, aunque sí existen algunos aspectos que son comunes entre muchos de ellos. Vamos a ver cuáles son los componentes más habituales que componen un motor de juegos:

- Motor gráfico 2D
- Motor gráfico o de renderizado 3D
- Detector de colisiones
- Motor de físicas
- Motor de inteligencia artificial (IA)
- Motor de sonidos
- Gestor de comunicaciones en red

En los siguientes apartados desarrollaremos en qué consiste cada uno de estos componentes y lo que aporta a un videojuego.

2.4.1.- Motor gráfico 2D.



Q WhiteTimberwolf (GNU/GPL)

La palabra "vídeo" proviene del latín y significa "yo veo". Es por ello que los videojuegos tienen una componente visual tan importante y que la sucesión de imágenes es la forma principal de comunicación con el jugador. El número de imágenes por segundo que se crean es un número llamado framerate y se mide en cuadros por segundo (FPS). Para que el ojo no perciba parpadeo, el framerate debe ser superior a 46 FPS.

Antes de detallar las características de los motores gráficos debemos introducir el concepto de 2D y 3D.

Los juegos 2D son aquellos en los que los dibujos tienen dos dimensiones, por ejemplo, ancho y alto, pero no profundidad. Para entendernos, son formas planas que se mueven por la pantalla. Podemos dibujarlas más grandes para parecer que están más cerca y más pequeñas para parecer que se alejan, pero siguen siendo planos. Un motor 2D permite dibujar de forma sencilla figuras geométricas (líneas, rectángulos, círculos, etc.) con lo que se evita tener que implementar los algoritmos en el código específico.

Normalmente, un juego 2D consiste en una o varias capas de imágenes en el fondo sobre las que se dibujan objetos 2D como pueden ser el jugador, los enemigos, monedas, obstáculos, disparos, etc. A esos objetos que son representativos en el juego se les denomina sprites .

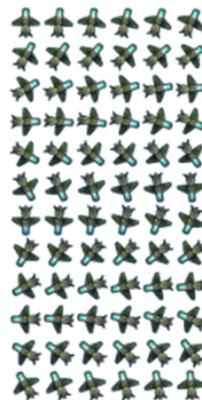
El motor controla todos los que estén activos en un momento dado, almacenando la posición en la que están, si hay que dibujarlos rotados, si están o no animados, etc.

Las animaciones en los juegos 2D se producen cambiando el dibujo del sprite rápidamente de manera que haya pequeñas diferencias entre uno y otro, lo que produce en el cerebro una sensación de continuidad. Observa la imagen de los aviones y verás cómo cada dibujo está un poco más girado que el anterior: si se dibujara uno tras de otro (de izquierda a derecha y de arriba a abajo) en la misma posición el efecto que captarías es que estaría girando en el sentido de las agujas del reloj.

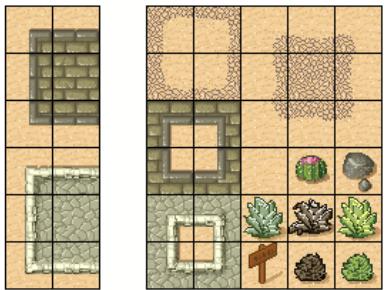
Los motores gráficos 2D dan soporte a los sprites, ofreciendo herramientas para escalar, rotar y mover objetos por la pantalla con comodidad. En la unidad de trabajo siguiente se verán estos conceptos con más detenimiento. También la información que se superpone en la pantalla (como el número de vidas o la puntuación).

Cuando dos sprites se superponen el resultado final dependerá del orden en que se dibujen. Dicho orden normalmente se denomina plano o capa. Si el motor lo soporta, hay que elegir el plano de cada sprites cuidadosamente para que se produzca el efecto que deseamos.

En algunos motores gráficos podemos representar mundos que sean varias



Q Killy Overdrive (CC BY)



Q Ministerio de Educación (Uso educativo nc)
fondos, mapas y niveles.

veces más grandes que el tamaño de la pantalla. El motor se encargará de mostrar solamente aquella parte que le indique el código específico. Esto permite, por ejemplo, seguir al jugador a lo largo y ancho de un nivel.

Otro elemento gráfico que poseen muchos motores 2D es el de **tilemap** (celosía), que es una composición de imágenes pequeñas del mismo tamaño. Este recurso es muy útil para dibujar



Autoevaluación

¿Cómo se realiza el efecto de animación en los motores gráficos 2D?

- Mediante la aceleración de la GPU y, en su caso, de la PPU.
- Mediante el efecto parpadeo del fondo de pantalla.
- Mediante una realimentación acústica.
- Mediante la sucesión rápida de dibujos con pequeños cambios.

Incorrecto

Incorrecto

Incorrecto

Opción correcta

Solución

1. Incorrecto (#answer-453_144)
2. Incorrecto (#answer-453_11298)
3. Incorrecto (#answer-453_11301)
4. Opción correcta (#answer-453_11304)

2.4.2.- Motor gráfico o de renderizado 3D.

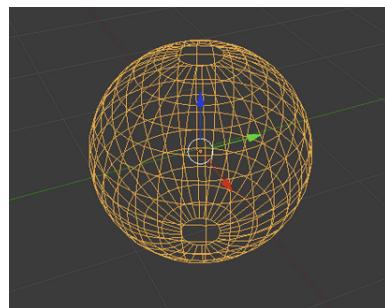
Por otro lado, en los juegos 3D los objetos tienen tres dimensiones y, por tanto, pueden moverse en el espacio con total libertad. Esto intenta imitar la visión del ser humano, que también es tridimensional. El problema es que la gran mayoría de los monitores sólo son capaces de representar imágenes 2D. Para solucionarlo podemos realizar una operación llamada proyección, que transforma esa imagen 3D en una 2D. Ese proceso de generar las imágenes recibe el nombre de renderizado y es tarea del motor gráfico.

Normalmente, los objetos 3D están constituidos por vértices que juntos forman polígonos, generalmente de tres o cuatro lados (figura 1).

En la figura 2 se observa una esfera 3D. Los vértices son los puntos donde confluyen las líneas. La flecha roja indica el eje X, la verde el eje Y y la azul el eje Z.

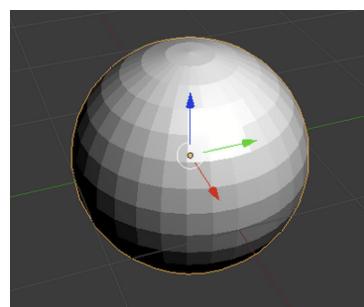
Los polígonos formados por los vértices se observan mejor en la figura 3. Fíjate que algunos se ven en un color más claro que otros. Todo ello es debido a la iluminación de la escena.

Figura 1



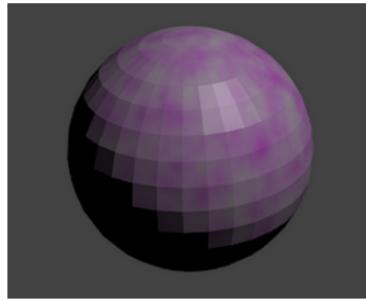
Q Ministerio de Educación (Uso
educativo nc)

Figura 2



Q Ministerio de Educación (Uso
educativo nc)

Figura 3



Q Ministerio de Educación (Uso
educativo nc)



Q SteveBaker (GNU/GPL)

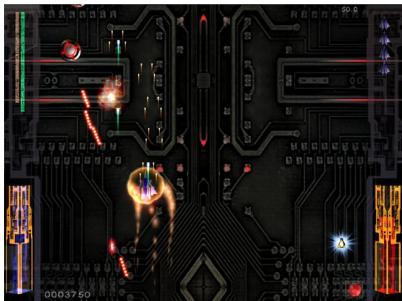
Sobre los polígonos se pueden aplicar texturas para darle un aspecto más realista. Simplificando mucho, podemos considerarlas como unas pegatinas que se pegan a su superficie. Las veremos con más detalle más adelante.

Un motor gráfico 3D simplificará el trabajo con los objetos tridimensionales permitiendo al código específico realizar operaciones con ellos

independientemente del hardware sobre el que se esté ejecutando el juego. Entre estas operaciones está la de crear objetos, moverlos, escalarlos, rotarlos, deformarlos, animarlos, cambiarles las texturas, etc. Además, deberá soportar la creación de luces que iluminen dichos objetos y cámaras que permitan observar el mundo renderizado desde la perspectiva que necesitemos en cada momento.

El conjunto de todos los objetos de este mundo virtual junto con las cámaras, las luces y demás elementos forman la escena. Internamente, la escena se suele almacenar mediante el denominado grafo de escena.

2.4.3.- Detector de colisiones.



Q Mark B. Allan (The Artistic Licence)

Tanto en el motor gráfico 2D como en el 3D tenemos objetos que se mueven por la pantalla. Una de las acciones que realizaremos más frecuentemente es comprobar si dos objetos colisionan entre sí. Por ejemplo: nos interesaría saber si una bala ha chocado contra un enemigo o si nuestro coche ha llegado a la meta. Ahí es donde entran los detectores de colisiones.

¿Es realmente necesario que se encargue un motor? ¿No podemos comprobarlo nosotros mismo desde el código específico de una forma sencilla? Resulta que detectar colisiones no es nada fácil, especialmente en ciertas circunstancias. En el caso 2D, dos objetos no han colisionado hasta que tienen un punto (píxel) en común. Eso implica tener que comprobar punto a punto de las dos imágenes buscando que ambas coincidan en algún sitio. Si tenemos en cuenta que puede haber decenas de objetos en pantalla de forma simultánea y que el objeto puede ser de gran tamaño, las cosas se complican. Podríamos simplificar el proceso de detección de una colisión suponiendo que hay un círculo o una caja que envuelve cada objeto y comprobar si dichas figuras geométricas se superponen: matemáticamente es sencillo de comprobar. **El problema es que pueden darse falsos positivos** de colisión, veamos un ejemplo a través de la imagen de los aviones.

En la parte izquierda de la imagen puedes comprobar como hay un falso positivo: las cajas de la izquierda están superpuestas pero los objetos en realidad no se tocan. En la parte derecha sí lo hacen. Si el motor hace bien su trabajo, cuando las figuras geométricas colisionen realizará un estudio más cuidadoso en la zona superpuesta para saber si efectivamente han chocado; **de esta forma podemos solucionar los falsos positivos.**



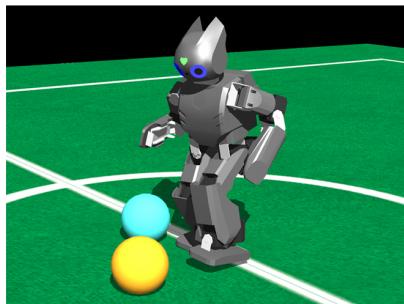
Q Luis Ramón López (CC BY)

Algunos detectores de colisiones clasifican los objetos en tipos de manera que sólo comprueban ciertas combinaciones. Por ejemplo: las colisiones de los objetos tipo "bala" con los tipo "enemigo" sí se comprueban, pero las colisiones de los objetos tipo "enemigo" con otros tipo "enemigo" no. **Esto permite ahorrar mucho tiempo de cálculo.**

Si el caso de los objetos 2D no es sencillo de solucionar, con los objetos 3D será más difícil aún porque los objetos tienen volumen y no todo está en el mismo plano.

2.4.4.- Motor de físicas.

Muchos de los videojuegos intentan dotar a los objetos que aparecen de un comportamiento creíble. Esto es, si empuja un objeto hacia el borde una mesa, el jugador espera que el objeto caiga al suelo. Si se golpea un cristal, esperamos que se haga añicos. O si lanzo golpeo una pelota, espero que siga una trayectoria natural y al llegar a la pared rebote de una forma natural.



Q Fabien Rohrer (CC BY-SA)

Todos estos comportamientos son calculados por el motor de físicas y pueden ir desde dotar de un efecto de gravedad al escenario donde estamos jugando hasta el cálculo de la dirección que tomarán dos objetos al chocar, calcular por donde discurre un líquido derramado, por donde se romperá un objeto al golpearlo o que al tirar de una cuerda se mueva un objeto que está atado al otro extremo. De

hecho, en entornos 3D suele ser el motor de físicas el encargado de detectar las colisiones entre objetos. Para poder realizar todas estas actividades el motor asignará una serie de propiedades físicas a cada objeto (velocidad, aceleración, masa, etc.) y aplicará las ecuaciones de la **física newtoniana**.

Los cálculos de este tipo de motores requieren un uso intensivo del procesador. Es por ello que hace unos años surgieron tarjetas aceleradoras de física, que permitían dejar libre la CPU para otras tareas. Su funcionamiento era similar a las GPU de las tarjetas gráficas 3D, pero su procesador (PPU) estaba dedicado solamente a realizar cálculos de simulaciones físicas. Poco después se trasladó esa funcionalidad a las propias GPU de las tarjetas gráficas, ya que éstas son capaces de realizar millones de cálculos por segundo en paralelo.

Por tanto, es importante conocer que algunos motores de físicas son capaces de descargar la CPU y realizar los cálculos en la GPU de la tarjeta gráfica o en la PPU de la tarjeta aceleradora de física.



Para saber más

Puedes aprender más sobre la mecánica newtoniana que es utilizada en la mayoría de los motores de física visitando este enlace de la Wikipedia: [Mecánica newtoniana](#).

2.4.5.- Motor de inteligencia artificial (IA).

Los jugadores esperan que los enemigos que aparezcan en el videojuego le supongan un desafío. De lo contrario, rápidamente se perdería el interés por jugar. Es ahí donde entra el motor de inteligencia artificial (IA) cuya misión es la de proveer a algunos de los elementos del juego de un comportamiento que parezca ser inteligente. Dada la complejidad de algunos de los problemas que soluciona el motor de IA y teniendo en cuenta que **la toma de decisiones tiene que hacerse en muy poco tiempo**, la mayoría de las veces se opta por usar técnicas simples y rápidas.

Por ejemplo, para encontrar de forma eficiente el trayecto que tendría que seguir un guardia de seguridad para llegar a un punto en un mapa se usan algoritmos de búsqueda de caminos. El más conocido, es el A*.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Q dbenzhuser (CC BY-SA)

situación. Por ejemplo: un guardia puede estar vigilando una zona y, en caso de detectar al jugador, dejaría inmediatamente esa tarea para proceder a perseguirlo. Si lo perdiera de vista, volvería a su tarea original. Al personaje del juego que no está bajo control directo del ordenador se le denomina NPC (personaje no jugador) y suele ser controlados por la IA.

En algunos juegos el comportamiento inteligente se produce mediante la ejecución de código específico condicionado a ciertos eventos. No es extraño que ese código esté en lenguaje de guiones (**scripting**) formando parte de los recursos; así no será necesario recomilar todo el proyecto para hacer cambios en el comportamiento del juego.

Cualquier técnica en el campo de la inteligencia artificial podría aplicarse aquí: **lógica difusa, redes neuronales, redes bayesianas, algoritmos genéticos**, etc. Eso sí, hay que considerar que no deben ser muy costosas computacionalmente hablando para que el juego no se ralentice.

En la figura, **el algoritmo A*** ha encontrado el camino óptimo (en rojo) para llegar al cuadro azul desde el cuadro verde, teniendo en cuenta que los cuadrados grises son obstáculos.

Un recurso muy utilizado para simular comportamientos sería el uso de **máquinas de estados finitos**. Su aplicación permite que un objeto se comporte de forma distinta según la

2.4.6.- Motor de sonidos.

Es el encargado de controlar la reproducción de música y efectos de sonido de manera sincronizada con el resto del juego. El código específico puede disponer de esta funcionalidad de forma independiente al hardware de la plataforma. Esto quiere decir que si, por ejemplo, el hardware no soportara la reproducción de más de un sonido a la vez, el motor podría realizar la mezcla él mismo para producir el mismo efecto sin tener que cambiar el código específico.

Tanto la música como los efectos de sonido son parte de los recursos del juego.

Ambos pueden incluirse como partitura (usando archivos MIDI) o bien de forma digitalizada, ya sea grabada directamente del mundo real o modificada con algún editor de audio. En los juegos también **pueden incluirse voces** que se reproducirán de forma sincronizada con el motor de gráficos con el fin de dar la sensación de que el modelo está moviendo los labios.

Los motores de sonido pueden ser muy complejos: algunos tienen conexiones con el motor de físicas para cambiar las características del sonido según a situación. Por ejemplo, podría producir un **efecto doppler** (el que hace que sepamos si una ambulancia está acercándose o alejándose simplemente por el sonido que emite). También, otros pueden distorsionar el sonido para **simular la acústica** de una sala o de un corredor, dotando de más realismo al juego.

Por último, algunos motores permiten generar sonido posicional en 3D. Eso quiere decir que pueden localizar espacialmente el sonido en aquellos sistemas que soporten audio multicanal (por ejemplo, 5.1) o bien simularlo en la salida de auriculares. Si el motor de sonidos está integrado con el motor gráfico 3D, una forma de conseguirlo es asociando el sonido a uno de los objetos de la escena; el resultado se obtiene automáticamente.

El motor de sonidos puede interactuar con otros motores:

- Con el motor gráfico para generar audio posicional.
- Con el motor de físicas para modificar las características del sonido según el entorno.
- Con el código específico, para iniciar un sonido cuando el jugador haga una acción.

2.4.7.- Gestor de conexiones en red.

El último componente de los motores que vamos a estudiar es el gestor de conexiones en red. **Este gestor nos permite que nuestro juego no esté aislado y pueda comunicarse con otros sistemas.** El uso más obvio es la creación de juegos multijugador, pero no es el único.

Otros escenarios que pueden resolverse con este componente son:

- Subida de la puntuación de la partida a un servidor Web o a una red social.
- Descarga de actualizaciones del juego.
- Implementación de salas para poder seleccionar contrincantes.
- Comunicaciones de voz y/o de texto entre varios jugadores.

Todo ello, como siempre, sin que el código específico tenga que saber nada de la infraestructura de red, el sistema operativo o la plataforma en general.

Muchos de los gestores de conexiones en red facilitan la creación de juegos multijugador permitiendo que el estado y las propiedades de los objetos (por ejemplo, su posición, su nivel de salud, etc.) se sincronice con todas las demás instancias del juego a través de un servidor. En estos casos es posible que el código específico no necesite apenas cambios para adaptarse al entorno multijugador.

La forma de implementar el multijugador varía de un motor a otro. Algunos usan la arquitectura cliente/servidor arquitectura cliente/servidor, de manera que todas las instancias de juego se conectan a un mismo servidor, que recibe los datos de cada cliente y los reenvía a los demás. Otros usan una arquitectura distribuida o **P2P** donde todos se comunican directamente con todos, aunque sólo suele usarse en redes de área local donde eso no supone una penalización de rendimiento.

La separación entre los roles de cliente y servidor es la más utilizada. De hecho, algunos juegos como Quake 2 o Quake 3, siguen este modelo incluso para el modo de un jugador, simplemente la máquina se conecta consigo misma. El servidor guarda en memoria el estado de la partida (objetos activos, vida restante de cada jugador, ítems, munición restante, etc.) y los clientes sirven solamente para enviar los movimientos o acciones (saltar, disparar, etc.) y para mostrar por pantalla la proyección de lo indicado por el servidor. El haberlo diseñado así permite extender soportar sin apenas cambios los modos multijugador.

Para finalizar, mencionar que un gestor de conexiones de red puede resolver los siguientes problemas:

- Permitir que varias personas jueguen en el mismo juego sin estar en el mismo ordenador.
- Buscar el camino óptimo para llegar a un punto del mapa teniendo en cuenta la posición de cada jugador.
- Posicionar los sonidos en función de la situación de los otros jugadores.
- Dibujar a los personajes que están conectados por la red.
- Permitir que los jugadores conversen entre sí, ya sea mediante texto o audio.

2.5.- Librerías que dan soporte a los motores.

Internamente los motores también se apoyan en otros componentes. Por ejemplo: el sistema operativo, librerías de representación gráfica o librerías de reproducción de sonidos. Algunas de esas librerías son multiplataforma mientras que otras sólo funcionan en ciertos sistemas operativos o un hardware en concreto.

Veamos algunas de las librerías más utilizadas respecto al aspecto gráfico:

- **DirectGraphics** (DirectDraw y Direct3D): Son librerías de Microsoft que se encuentran disponibles en sus productos, como el sistema operativo Microsoft Windows y en su consola Xbox360 y XboxOne. DirectDraw soporta funciones para dibujar en 2D, mientras que Direct3D se usa para dibujar escenas 3D a partir de primitivas sencillas. En otros sistemas operativos puede ser emuladas usando Wine, un entorno que permite ejecutar programas Windows. Las dos librerías forman parte de la colección de APIs DirectX. La versión 12 de DirectX está incorporada de serie en Windows 10, aunque puede también instalarse en otras versiones de Windows.
- **Vulkan**: Es la librería gráfica 2D y 3D más utilizada. Se soporta en múltiples sistemas operativos y plataformas. Está gestionada por el consorcio sin ánimo de lucro Khronos Group y está en constante revisión. La última versión a la hora de escribir estas líneas es la 1.3.206. Proviene de OpenGL.
- **Metal**: Apple también cuenta con su propia librería gráfica, y su desarrollo lo inició para mejorar enormemente el rendimiento gráfico en iOS. Permiten sacar el máximo partido a la GPU de 64 bits que incluyen los últimos iPhones.

Respecto al audio:

- **DirectSound y DirectSound 3D**: También forman parte de DirectX. La primera de ellas, especializada en audio 2D fue dividida posteriormente en dos librerías (XAudio2 y XACT3). Presentan el mismo problema que DirectGraphics ya que sólo funcionan en software o hardware de Microsoft. Cuenta con la API WASAPI para interactuar con ella.
- Otras alternativas: **ASIO, Core Audio**.

Y por último, librerías que ofrecen funcionalidad combinada:

- **SDL** (Simple DirectMedia Layer): Ofrece una interfaz de programación multiplataforma muy sencilla para acceder a los dispositivos gráficos (2D) y de sonido. Además, permite gestionar los dispositivos de entrada, como joysticks, gamepads, teclados, pantallas táctiles y ratones. Suele combinarse con Vulkan si se desea funcionalidad 3D. Si no se necesitan otros motores específicos, SDL puede ser la base sobre la que construir un videojuego.
- **Allegro**: Similar a SDL, aunque incorpora soporta 3D (sin aceleración, de forma limitada).

Parte II - Desarrollo de un videojuego.



Caso práctico

Juan está decidiendo su futuro profesional a corto plazo. Para ello ha decidido probar usando uno de estos motores de videojuegos sobre los que tanto puede leerse en Internet. En la empresa donde ha realizado la entrevista le han hablado de GDevelop; al parecer es uno de los motores usados para juegos sencillos. El primer paso a seguir por parte de Juan será instalar el motor en su ordenador para intentar construir un juego por sí mismo y entender cómo sería su trabajo en caso de aceptar la oferta.



Q [Mikhail \(https://www.pexels.com/es-es/foto/empresario-hombre-persona-gente-6592661/\). \(Pexels\)](https://www.pexels.com/es-es/foto/empresario-hombre-persona-gente-6592661/)

Leyendo sobre el motor GDevelop descubre que es sencillo de aprender y que cuenta con varios elementos interesantes como la capacidad de uso de la Inteligencia Artificial para algunos eventos como el cálculo de caminos hacia objetos en la escena del videojuego.

Además, se trata de un motor gratuito y que ofrece un montón de posibilidades para introducirse en este mundo de creación de juegos. Otra de las ventajas es que es multiplataforma, es decir desde su página web existen instaladores/ejecutables para Windows, Linux y MacOSX.



Q [Ministerio de Educación y Formación Profesional.](#)
(Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

1.- La herramienta de desarrollo GDevelop. Introducción.



Caso práctico

Se trata de un motor de videojuegos de código abierto para dispositivos móviles y ordenadores en géneros como plataformas, puzzles, estrategia, disparos y más.

Cuenta con un sistema de eventos con los que es posible determinar la lógica de tu juego: podrás añadir comportamientos predeterminados a tus objetos de juego, o expandir y crear nuevos comportamientos que actúen de manera intuitiva.

Incluye sprites con múltiples animaciones, emisor de partículas, máscaras de colisión personalizadas y generador de patrones entre otros recursos. También es posible expandir el motor de juego mediante JavaScript e incluso incluir anuncios para obtener una fuente de ingresos por medio de tu proyecto.

El primer paso es la instalación del motor y el segundo paso la creación de un nuevo proyecto.



Q [GDevelop en Wikipedia](#)

(<https://es.wikipedia.org/wiki/GDevelop#/media/Archivo:GD-logo-big.png>). (Dominio público)

Para llevar a cabo nuestro videojuego, vamos a utilizar una potente, pero a la vez sencilla herramienta de código abierto llamada GDevelop.

GDevelop permitirá crear juegos HTML5 para ser jugados en cualquier navegador web o mediante aplicaciones. Según el propio desarrollador, los juegos son exportables a las plataformas iOS, Android, Steam, Facebook Gaming, Itch.io, Newgrounds, the Microsoft Store y otras sin necesidad de tener que usar, o siquiera conocer un lenguaje de programación específico. Dispone de una organización y distribución de componentes bastante intuitivo y donde la potencia principal de la herramienta reside en un logrado y completo sistema de gestión y control de eventos.

Versiones anteriores del motor ofrecían juegos para entornos Escritorio Windows y Linux en formato C++, pero esta característica está obsoleta.

La [arquitectura de GDevelop](#), según la fuente indicada en el enlace, viene distribuida en una serie de directorios cuya finalidad está descrita en esta lista:

- Core: La biblioteca principal de GDevelop, que contiene herramientas comunes para implementar el IDE y trabajar con los juegos de GDevelop.
- GDJS: El motor de juego, escrito en TypeScript y que usa PixiJS (WebGL) para crear juegos todos los juegos GDevelop.
- GDevelop.js: Enlaces de Core, GDJS y Extensiones a JavaScript (con WebAssembly), utilizados por el IDE.
- newIDE: El nuevo editor del juego, escrito en JavaScript con React, Electron

y PixiJS.

- Extensions: Extensiones para el motor del juego, provee objetos, comportamientos, eventos y otras características.

1.1.- Instalación de la herramienta de desarrollo GDevelop.

Podemos obtener GDevelop para instalarlo en nuestro ordenador de través de la página: <https://gdevelop.io/download>. Se accede a la página mostrada en la figura 1.

Si clicamos sobre el enlace Download, se abrirá la página mostrada en la figura 2, con enlaces a la instalación sobre los sistemas operativos más utilizados, e incluso es posible crear juegos sin ni siquiera instalar la herramienta, utilizando directamente el navegador web.

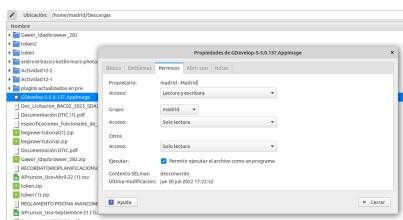
Figura 1. Página GDevelop.



Figura 2. Clic en Download.



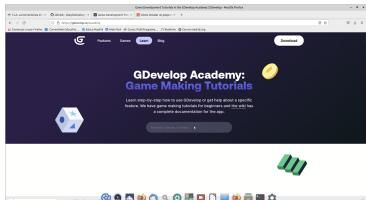
Se deja a elección del alumno la versión a instalar, aunque se recomienda la más reciente. Puede haber cambios con respecto a lo que aquí se muestre, pero en todo caso, será bastante similar. En este caso el juego se ha desarrollado con la versión 5.5.0.137, en un sistema operativos MAX 11.5 (Madrid Linux), una distribución basada en Ubuntu 20.04. En caso de tener un sistema operativo Windows el proceso de instalación es el mismo que con cualquier otro tipo de aplicación Windows.



Ministerio de Educación (Uso
educativo nc)

En el caso de Linux, la aplicación se distribuye en forma de AppImage. Una vez descargada y descomprimida es necesario dotar al fichero de imagen (.AppImage) de permisos de ejecución y para ejecutarlo basta con ejecutar en un terminal el nombre del paquete desde la carpeta donde se encuentra descomprimido. En este caso desde la carpeta /home/madrid/Descargas ejecutaremos ./GDevelop-5.5.0.138.AppImage

1.2.- Tutoriales y procedimiento de elaboración de nuestro juego.



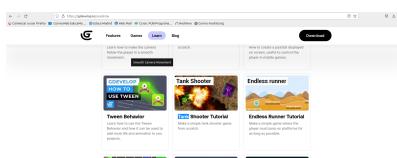
Q GDevelop.io (

Todos los derechos reservados)

ambos de tipo 2D,
<https://gdevelop.io/academy>

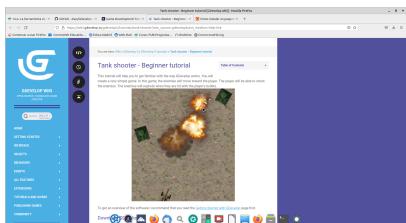
Volviendo a la página web principal de GDevelop, podemos observar que en la parte superior existe un botón Learn que habilita el acceso a la zona de tutoriales para crear unos juegos de temáticas diferentes y que con un esfuerzo relativamente pequeño nos permitirán crear en poco tiempo unos juegos,

con resultados bastante satisfactorios:



Q GDevelop.io (Todos los derechos reservados)

Si nos desplazamos hacia abajo en la página, encontraremos un tutorial sobre la creación del juego "Tank Shooter Game", el enlace nos lleva a un manual que explica cómo crear un juego donde un cañón orientable por el jugador y que dispara proyectiles de manera controlada también por el jugador, se defiende del avance dirigido hacia éste, de múltiples tanques que tratan de colisionar contra dicho cañón.



Q GDevelop.io (Todos los derechos reservados

)

Si abrimos el tutorial accederemos a las instrucciones y a una serie de recursos como son sprites de los personajes, imágenes estáticas, sonidos, fuentes para textos, etc. Te recomendamos que uses este manual para los casos en que no encuentres una solución a cómo hacer los distintos puntos de la creación del juego.

2.- Videojuego de tanques.



Caso práctico



🔍 [Mikhail](#)

(<https://www.pexels.com/photo/a-man-in-white-long-sleeve-shirt-6592370/>). (Pexels)

controla y con la que puede ofrecer más alternativas a las empresas que le contratan.

Definitivamente, Juan ha decidido su futuro a corto plazo. Aunque el mundo de los videojuegos es muy tentador, la ausencia de necesidad de escribir código, al menos en estos motores sencillos, hace que se decante por seguir apostando por el desarrollo de aplicaciones Android, donde la importancia de los conocimientos técnicos de informática y de programación es vital. En este campo, Juan se siente competente y puede marcar la diferencia frente a otros profesionales que no han estudiado el ciclo de grado superior de Desarrollo de Aplicaciones Multiplataforma, así que por ahora continuará su aprendizaje trabajando junto a sus compañeros en su actual empresa.

El próximo reto de Juan es... ¡aprender Kotlin! Pero el tuyo, por ahora, diseñar este videojuego de tanques.



Autoevaluación

Elije las respuestas adecuadas sobre GDevelop...

- Los juegos resultantes están en HTML5.
- La versión actual de GDevelop ofrece juegos en formato C++ para juegos nativos de Windows y Linux.
- El IDE está desarrollado con React, Electron y PixiJS.



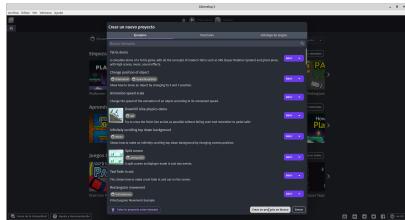
El motor está escrito con TypeScript.

Mostrar retroalimentación

Solución

- 1.** Correcto (#answer-490_139)
 - 2.** Incorrecto (#answer-490_11432)
 - 3.** Correcto (#answer-490_11434)
 - 4.** Correcto (#answer-490_11436)
-

2.1.- Creación del proyecto.



Q Elaboración propia a partir de GDevelop ([MIT License](#))

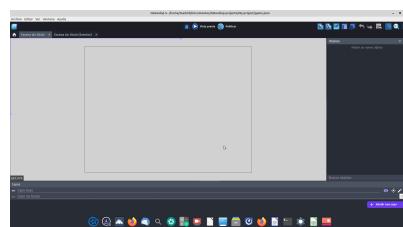
descargar alguno de los proyectos existentes (opción muy interesante para aprender), o crear uno desde cero.

A continuación seleccionamos un nombre para nuestro proyecto y una ruta donde se almacenarán todos los ficheros del juego.

Nota importante: antes de seguir, se recuerda al alumno la conveniencia de ir guardando todos los cambios que se vayan haciendo en el proyecto. No se guardan automáticamente los cambios, por lo que no salvarlos sistemáticamente puede jugarnos una mala pasada.

Se creará entonces una escena y quedará preparada para la incorporación de componentes del juego con las siguientes áreas principales:

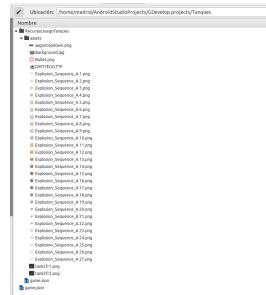
- Una escena está compuesta por varias capas, como puedes ver en la parte inferior.
- En la parte derecha está la zona de creación de objetos.
- En la pestaña de la derecha, "Escena sin título (Eventos)" es donde podremos programar el comportamiento de nuestro juego.



Q Elaboración propia a partir de GDevelop ([MIT License](#))

2.2.- Incorporación de recursos.

A continuación incorporaremos los recursos del juego. Concretamente podrán descargarse del siguiente enlace:



Q Elaboración propia a partir de

GDevelop ([MIT License](#))

<https://wiki.compilgames.net/lib/exe/fetch.php/gdevelop5/tutorials/beginner-tutorial.zip>

Tendremos que descomprimir dicho fichero e incorporar el contenido en la misma carpeta del proyecto. En nuestro caso hemos renombrado la carpeta de recursos a "RecursosJuegoTanques".

2.3.- Creación del objeto cañón e incorporación a la escena.

A continuación incorporaremos el cañón con el cual nos defenderemos del avance y amenaza de los tanques. Consistirá en un objeto cuya imagen se encuentra entre los recursos proporcionados. Para ello sobre la zona derecha de la pantalla, en la zona de Objetos clicaremos en "+" para insertar un nuevo objeto, apareciendo la ventana desde la que elegiremos el tipo de objeto "Sprite" (figura 1), donde indicaremos el nombre "Canon" (figura 2).

Sobre dicha ventana tendremos que agregar la imagen del cañón. Para ello, pulsamos "Añadir una animación" y en la siguiente ventana pulsaremos al botón Añadir y seleccionaremos la imagen del cañón marrón de la carpeta de los elementos descargados en el apartado anterior, quedando como se muestra en la figura 3.

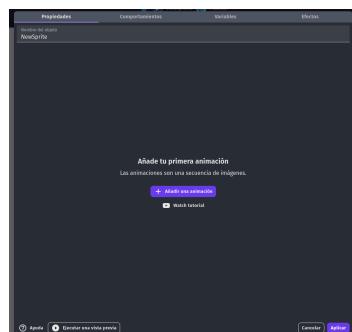
Figura 1. Insertar objeto.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

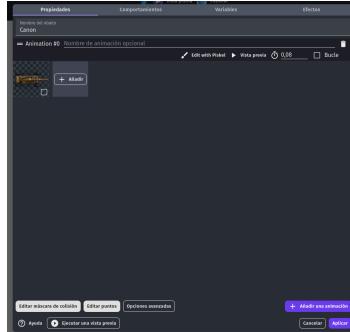
Figura 2. Objeto sprite "Canon"



Q Elaboración propia a partir de GDevelop

[\(MIT License\)](#)

Figura 3. Añadir animación.



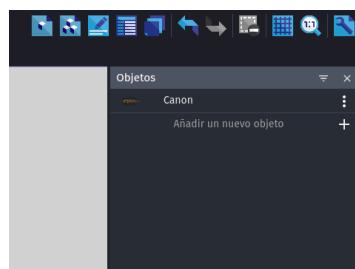
Q Elaboración propia a partir de GDevelop

([MIT License](#))

Al cerrar la ventana en la parte derecha habrá quedado creado el objeto "Canon" tal y como se muestra en la figura 4 y podemos arrastrarlo hasta la escena principal para ubicarlo, donde podemos cambiarlo de posición o redimensionarlo (figura 5).

También es posible girarlo haciendo clic sobre el punto de la imagen (figura 6).

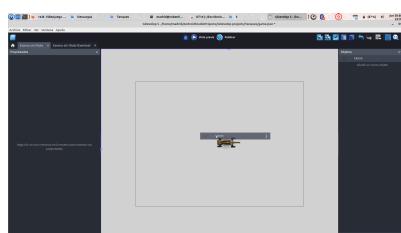
Figura 4. Objeto "Canon"



Q Elaboración propia a partir de GDevelop

([MIT License](#))

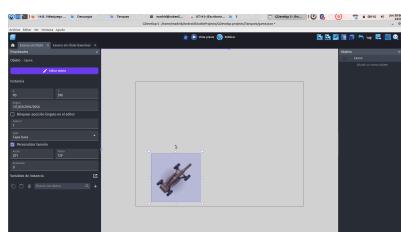
Figura 5. Reposicionar o redimensionar.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 6. Girar.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

2.4.- Incorporando los primeros eventos.

Ahora es momento de incorporar los primeros eventos. En nuestro caso, haremos que el cañón gire orientándose hacia donde se sitúe, en la escena del juego, el cursor de ratón.



Para ello tenemos varias opciones. La más rápida es pulsando sobre la pestaña superior "Escena sin título (Eventos)". Aquí pulsamos sobre "Insertar un nuevo Evento" quedando como se muestra en la imagen.

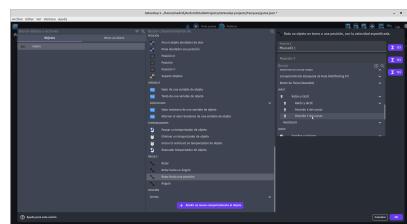
Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Ahora agregaremos una acción (pulsamos sobre el texto "Añadir una acción" en la parte derecha), apareciendo la ventana donde seleccionamos el objeto "Canon" y nos desplazamos en la parte derecha hacia abajo hasta encontrar "Rotar hacia una posición" en el grupo ÁNGULO donde podemos especificar los siguientes valores que indican que el objeto girará hacia la posición del ratón:

- Posición X: MouseX()
- Posición Y: MouseY()
- Velocidad Angular: 0 (inmediata)

NOTA: Podemos escribir directamente observando que el IDE muestra el color rojo en caso de que el valor introducido no sea correcto sintácticamente o pulsar también el botón morado "E 123" y seleccionar en INPUT los eventos del ratón "Posición X del cursor" e Y. Observa que los valores y las variables son sensibles a mayúsculas.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Al dar a Ok, ya se quedará en primer lugar en el desplegable de acciones.

Como para dicha acción no hemos especificado condición (sin condiciones), el cañón refrescará su posición aproximadamente 60 veces por segundo, de forma que prácticamente de manera inmediata, cada vez que mueva el cursor,

el cañón se moverá apuntando al cursor.

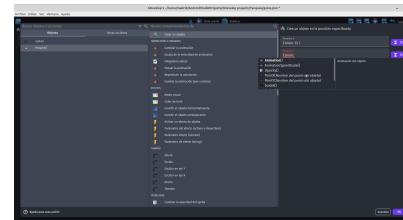
Ahora sería el momento de comprobar que efectivamente el cañón obedece al evento de giro descrito. Para ello probaremos lo que hace hasta el momento mi aplicación pulsando el botón superior de Vista Previa para comprobar que el cañón gira cuando movemos el ratón.

2.5.- Más eventos del cañón. Disparos.

En este punto del desarrollo, haremos que nuestro **cañón dispare balas o proyectiles al pulsar el botón principal del ratón**. Para ello, lo primero será incorporar el objeto característico o representativo de un proyectil, siguiendo los mismos pasos que cuando creamos el Cañón (objeto "Canon"), es decir, sobre la zona derecha de la pantalla, en la zona de Objetos clicaremos en "+" para insertar un nuevo objeto, apareciendo la ventana desde la que elegiremos el tipo de objeto "Sprite", donde indicaremos el nombre "Proyectil".

A continuación, sobre dicha ventana tendremos que agregar la imagen del proyectil (bullet.png). Para ello, pulsamos "Añadir una animación" y, en la siguiente ventana, pulsaremos al botón Añadir y seleccionaremos la imagen del proyectil  de la carpeta de los elementos descargados anteriormente.

El siguiente paso consistirá en crear un nuevo evento desde la pestaña superior derecha (escena sin título "Eventos") y en este segundo evento creado pulsamos en la parte derecha, encima del título "Añadir Acción". Después seleccionamos en la izquierda el tipo de objeto que queremos crear: proyectil. En la derecha indicamos el comportamiento que queremos, en este caso seleccionamos Crear un Objeto. Cuando lo hacemos a la derecha se abren las propiedades de la creación, en este caso necesitamos decirle dónde deseamos (en qué coordenadas de la escena) que sea creado el proyectil.



Q Elaboración propia a partir de GDevelop (
[MIT License](#))

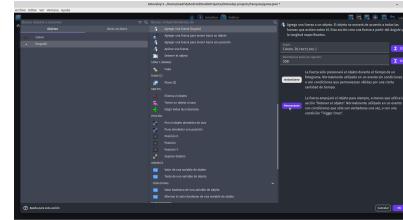


Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Observa que la posición donde queremos que el proyectil sea creado es variable, depende de las coordenadas del cañón, que puede girar. Para indicar esto usamos las funciones Canon.X() y Canon.Y() que devuelven la posición del cañón.

Esto sólo situará el proyectil en dicha posición (además ni siquiera será en la boca del cañón). Ahora añadiremos otra acción, debajo de esta acción ya creada, pero dentro de este mismo segundo evento.

Con esta nueva acción trataremos de dotar de movimiento al proyectil con la misma dirección que tiene el cursor en el momento de ser disparado el proyectil. Para ello seleccionaremos el tipo de objeto proyectil en la parte izquierda, en el medio seleccionamos el grupo **MOVIMIENTO USANDO FUERZAS -->**

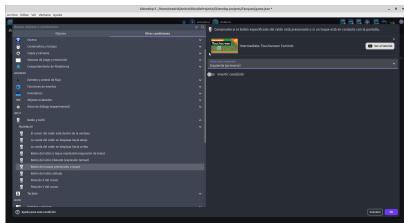


Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Agregar una fuerza (ángulo) y en la parte derecha indicamos estas propiedades:

- Ángulo: Canon.Direction() que indica que aplicará la misma dirección al proyectil que ángulo esté girado el cañón.
- Velocidad: 350 (píxeles por segundo).

- Permanente: Marcado.



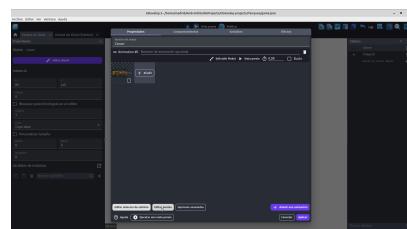
Q Elaboración propia a partir de GDevelop (
MIT License)

Ahora debemos asociar a estas dos acciones de este segundo evento una condición en la parte izquierda para que se desencadenen. En nuestro caso, haremos que al clicar sobre el botón izquierdo del ratón se realicen dichas acciones de creación del proyectil y su movimiento dirigido. Para ello, sobre este

segundo evento añadiremos una condición ("Añadir condición" en la parte izquierda). Elegiremos "Ratón y táctil" > "Botón del ratón presionado o toque" > y en el campo "Botón para comprobar" dejamos "Izquierda (Primario)", o bien lo seleccionaremos a través del navegador de botones a su derecha.

Ya podremos probar el resultado, viendo que efectivamente dispara, pero con el inconveniente de que los proyectiles no salen de la boca del cañón, ya que hemos indicado la coordenada del cañón en general (todo el objeto) pero no de la boca del cañón.

Para solucionarlo debemos cambiar el punto donde se crea el proyectil, pasando a indicar que ese punto es la boca del cañón. Para indicarlo, vamos a crear un punto con coordenadas en el objeto "Canon" que llamaremos "Boca" y que cuyas coordenadas usaremos después.



Q Elaboración propia a partir de GDevelop (
MIT License)

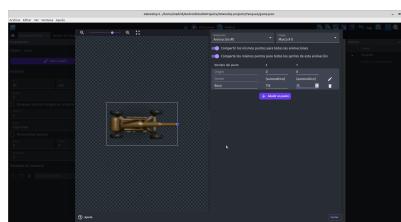
Para ello, seleccionamos arriba a la izquierda la pestaña "Escena sin título". Esto despliega a la derecha la lista de los objetos creados hasta ahora ("Canon y Proyectil"). Hacemos doble clic el objeto "Canon" y clicamos el botón "Editar Puntos" en la parte inferior.

Ahora añadimos un punto en la derecha y especificamos las coordenadas adecuadas, que pueden variar dependiendo de si has cambiado el tamaño del cañón. Observa que el punto debe quedar justo en la boca del cañón, como se muestra en la figura 1, y observa que le hemos dado el nombre al punto de "Boca".

Toca reescribir las coordenadas de creación del proyectil para que, en lugar de crearse en el punto "Canon.X()" x "Canon.Y()", se creen en los puntos "Canon.PointX("Boca")" x "Canon.PointY("Boca")", tal y como muestra la figura 2.

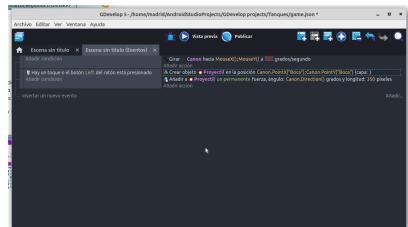
De esta forma el proyectil saldrá de la boca del cañón como se aprecia en la figura 3.

Figura 1



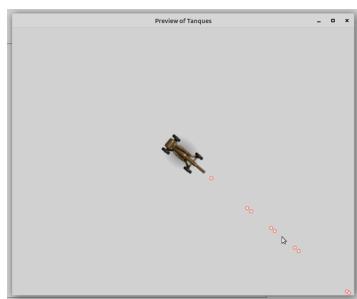
Q Elaboración propia a partir de GDevelop (

Figura 2



Q Elaboración propia a partir de GDevelop ([MIT License](#))

Figura 3

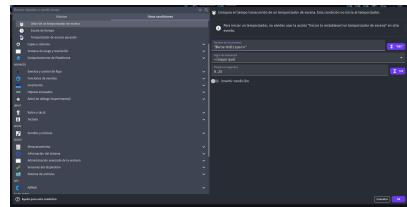


Q Elaboración propia a partir de GDevelop ([MIT License](#))

Además, si queremos, podemos incorporar un pequeño retardo desde que clico el botón izquierdo del ratón por primera vez y el proyectil se dispara. Quizá en este juego este comportamiento no sea necesario, pero en otro tipo de juego puede que sí. Veamos cómo hacerlo.

Se tratará de incorporar una condición basada en un temporizador, una especie de alarma para que hasta que no suceda no permita crear los proyectiles. Esto retardará la creación de los disparos.

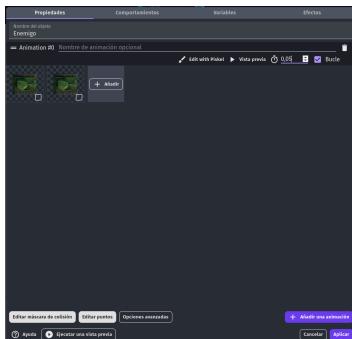
Para ello, sobre este segundo evento, agregaremos una segunda condición yendo a Cronómetros y tiempo --> Valor de un temporizador de un temporizador de escena. Aparecerán tres campos que rellenaremos: por un lado, el nombre que daremos a este temporizador: "RetardoDisparo"; por otro lado, el tiempo medido en segundos: 0.25; y por otro lado el signo de evaluación: ">". Esto significa que hasta que no pasen 0.25 segundos no se podrá hacer un siguiente disparo.



Q Elaboración propia a partir de GDevelop ([MIT License](#))

Si ahora lo probamos, veremos que aún no se produce el retardo. Se debe a que, como puedes ver en la imagen anterior, es necesario iniciar el temporizador, algo así como activar la alarma. Veremos cómo hacerlo en próximos apartados.

2.6.- Creación de los objetos enemigos. Tanques como imágenes animadas.



Q Elaboración propia a partir de GDevelop (MIT License)

Ahora trataremos de **crear los enemigos del cañón**, que básicamente serán múltiples tanques que irán apareciendo de manera aleatoria en posición (solamente exigiremos que aparezcan por la parte de arriba) cada cierto tiempo (puede ser a intervalos fijos o aleatoriamente en el tiempo) y **que se acercarán a nuestro cañón**. Para modelizar el comportamiento de cada uno de estos tanques, crearemos el objeto de tipo "Enemigo" de modo similar al Cañón, pero la diferencia será que agregaremos dos imágenes con idea de crear una imagen animada provocada por la alternancia sucesiva de esas dos imágenes. En la animación dichas imágenes se intercambiarán cada 0.05 segundos, de modo que dará sensación de que las cadenas del tanque enemigo se mueven.

Una vez incorporadas las imágenes pulsamos sobre la caja de texto, donde está el ícono del cronómetro, el tiempo que dura cada imagen antes de cargar la otra, en este caso 0.05 segundos.

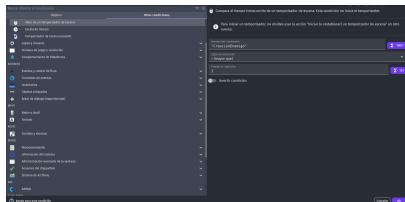
Además, como queremos que las imágenes se alternen y se vuelvan a realizar continuamente en forma de bucle, marcaremos la casilla "Bucle".

2.7.- Fijar dirección de los tanques. Inteligencia artificial.

En este apartado buscamos crear objetos de tanques enemigos constantemente (en intervalos de 1 segundo) y que se dirijan hacia el cañón con ánimo de chocarse contra él.

Para ello añadiremos en el Editor de eventos un tercer evento. En principio, la dirección que deben tomar los tanques debe ser directa hacia al cañón.

En nuestra primera versión, el cañón no se traslada, solo rota moviendo el ratón. Esto significa que los tanques únicamente tendrán que calcular su dirección una vez, cuando inician tras su instanciación el movimiento. Pero si el cañón tuviese incorporado movimiento de traslación, los tanques debieran corregir su dirección para dirigirse a la nueva posición del cañón. Es decir, dispone de ciertas funciones propias de la inteligencia artificial que permiten en todo momento recalcular su ruta atendiendo a factores externos. El método que usaremos de desplazamiento de los tanques tendrá en cuenta esta posibilidad y, al final del proyecto, cuando incorporemos traslación, comprobaremos que este fenómeno está perfectamente contemplado.



Q Elaboración propia a partir de GDevelop ([MIT License](#))

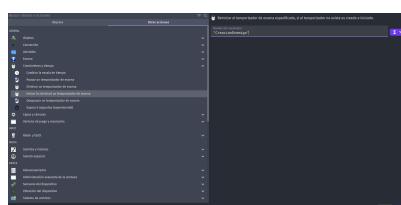
En primer lugar, deben crearse tanques que se dirigirán hacia el cañón. Esto lo haremos, por ejemplo, cada segundo. Para ello iremos al editor de eventos y crearemos un nuevo evento, el tercero. Sobre dicho evento añadiremos una condición (haciendo clic en "Añadir condición" a la izquierda). Seleccionamos

la condición "Valor de temporizador de escena" que configuramos de esta forma, llamando a este temporizador "CreacionEnemigo" (debe ir entre comillas).

Además, observa que, al crear el temporizador, el propio interfaz advierte que es necesario iniciar/restear el temporizador desde un evento. Es decir, hay que añadir una acción para iniciar e ir reseteando el contador, de forma que cada vez que pase un segundo, se lleva a cabo la acción, paramos el tiempo y volvemos a activar la cuenta para que en el próximo segundo volvamos a hacer la acción repetidamente.

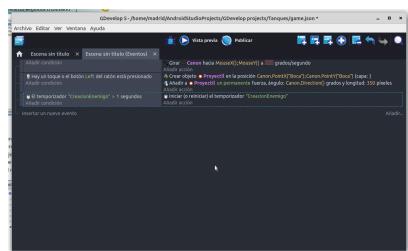
Para ello damos a "Añadir acción" en la derecha y seleccionamos en el grupo de "Cronómetro y tiempo" > "Iniciar (o reiniciar) un temporizador de escena". Ahora configuramos el Nombre del cronómetro, el mismo que indicamos en el evento anteriormente, es decir, "CreacionEnemigo" (figura 1). Damos a Ok quedando como muestra la figura 2.

Figura 1



Q Elaboración propia a partir de GDevelop (

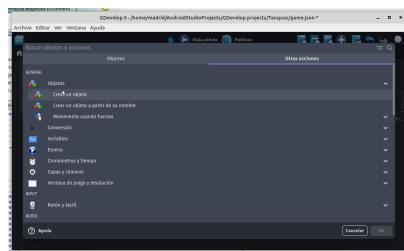
Figura 2



Q Elaboración propia a partir de GDevelop (

MIT License)

Además, ahora añadiremos otra acción para crear un objeto "Enemigo". Para ello, daremos de nuevo, para este tercer evento, en la columna de acciones, "Añadir acción" > Pestaña "Otras acciones" > Objetos > "Crear un objeto", escribiendo como primer parámetro (Objeto a crear): "Enemigo", como segundo parámetro (Posición X, del objeto a crear): Random(800), de forma que el objeto se creará con un valor aleatorio en la componente X, que cubrirá más o menos el ancho de la ventana del juego, mientras que el tercer parámetro (Posición Y, del objeto a crear): -50, de forma que los tanques se crearán fuera de la ventana de juego (por la parte superior, teniendo en cuenta que el valor 0 del eje Y corresponde justo al marco horizontal superior, para que así el jugador no tenga la sensación de que los tanques de repente "aparecen".



Q Elaboración propia a partir de GDevelop (

MIT License)



Q Elaboración propia a partir de GDevelop (

MIT License)



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

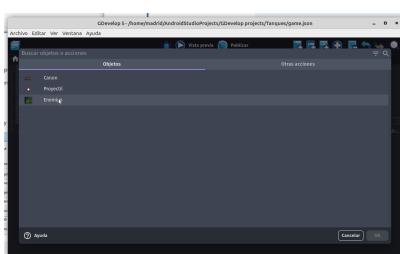
Esto hará que los tanques se creen, sin embargo, no tienen atribuido todavía movimiento. Para ello debemos crear un nuevo evento (cuarto) pero sin ninguna condición, ya que los tanques siempre se moverán.

Ahora sobre este cuarto evento, añadiremos una acción (figura 3): "Añadir acción", marcamos el objeto "Enemigo" (esto aplicará sobre todas las instancias de esta clase de objetos, todos los enemigos), seleccionamos "Aregar una fuerza para mover hacia un objeto", y rellenaremos los campos de la forma:

- Objetivo: Canon (ya que los enemigos se moverán hacia donde esté el cañón).
- Velocidad (en pixeles por segundo): 1 (puede que sea un bug del motor en el momento de hacer esta documentación, pero con 150px que es lo esperadamente adecuado va demasiado deprisa).
- Permanente: Marcado.

Dando OK el resultado quedaría tal y como muestra la figura 4.

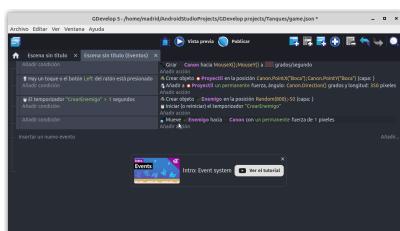
Figura 3.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

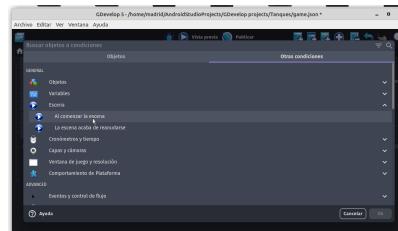
Figura 4.



Q Elaboración propia a partir de GDevelop (

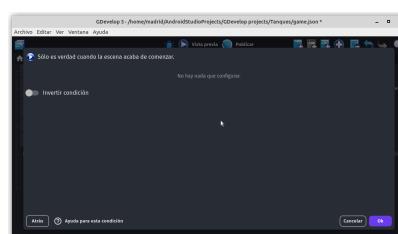
[MIT License](#))

Para que los enemigos sean creados es necesario arrancar el temporizador que vaya generando eventos cada segundo. La idea es arrancarlo al empezar el juego y cada vez que se detecte que ha pasado un segundo, crear al enemigo y volver a reiniciar el contador. Esto es justo lo que hacemos creando el siguiente evento que, aunque al inicio estará en la parte de abajo, luego lo arrastraremos hacia la parte de arriba, para que quede más claro que se lanza al inicio de cada partida.



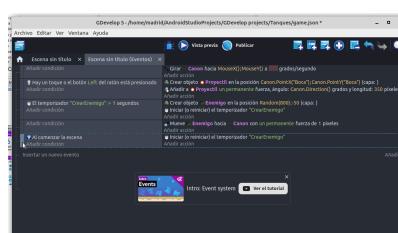
Q Elaboración propia a partir de GDevelop (

[MIT License](#))



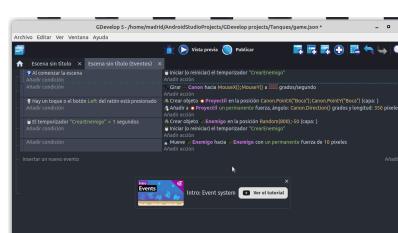
Q Elaboración propia a partir de GDevelop (

[MIT License](#))



Q Elaboración propia a partir de GDevelop (

[MIT License](#))



Q Elaboración propia a partir de GDevelop (

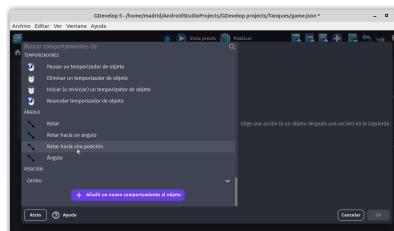
[MIT License](#))

Si hacemos una prueba mediante la vista previa, vemos que los tanques se crean e incluso se aproximan al cañón según lo programado, pero llama la atención que los tanques enemigos no tienen la orientación de manera adecuada, es decir, la boca de su cañón, no está apuntando al cañón del jugador, provocando un

efecto bastante irreal.

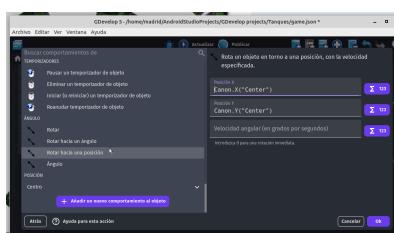
Para solucionar esto, añadiremos al (ahora quinto) evento una segunda acción detrás de la acción de "Mueve Enemigo...". Para ello pulsamos en "Añadir acción", seleccionamos el objeto "Enemigo", buscamos en el grupo "ÁNGULO" > "Rotar hacia una posición", y rellenaremos los parámetros de la forma:

- Posición X: Canon.X("Center") (se referirá a cada tanque)
- Posición Y: Canon.Y("Center") (se referirá a cada tanque)
- Velocidad angular (en grados por segundo) (0 para una rotación inmediata): 0



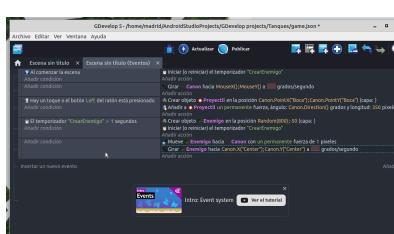
Q Elaboración propia a partir de GDevelop (

[MIT License](#))



Q Elaboración propia a partir de GDevelop (

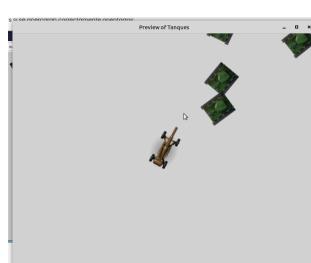
[MIT License](#))



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Si ahora probamos de nuevo, vemos que ahora los tanques sí se acercarán correctamente orientados:



Q Elaboración propia a partir
de GDevelop ([MIT License](#))

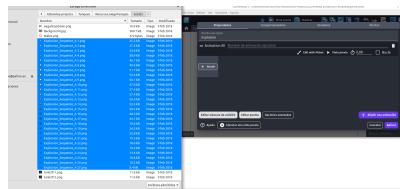
2.8.- Más animaciones. Explosiones.

A continuación trataremos de añadir explosiones para que tras el impacto de un proyectil procedente de cañón sobre el tanque, se muestre una explosión. Para ello añadiremos un nuevo Sprite al cual llamaremos "Explosion".

Para ello, vamos al Editor de objetos del panel derecho superior y hacemos clic sobre "Añadir un nuevo objeto", seleccionamos Sprite y en la pantalla de Propiedades del objeto, indicamos el nombre: "Explosion" y a continuación pulsamos el botón "+ Añadir" desde donde podemos seleccionar y cargar todas las imágenes que tenemos en nuestra carpeta de recursos cuyo nombre son Explosion_X (figura 1).

Además, para especificar el tiempo entre cada una de esas imágenes, especificaremos el valor 0,03 (será medido en segundos) en el ícono del cronómetro en la parte superior, quedando como muestra la figura 2.

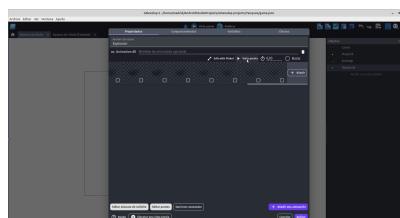
Figura 1



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 2

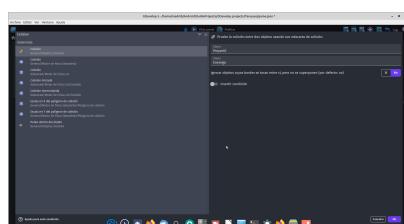


Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Nota: Los nombres que traen los ficheros originales, hacen que por ordenarse alfabéticamente, se cambie el orden de la secuencia de imágenes. Por ello conviene cambiar nombre de forma que el "1" pase a ser "01" y así hasta el "9" por "09".

Podemos comprobar la secuencia sin salir de esta ventana pulsando sobre "Vista previa" junto al lugar dónde has especificado el tiempo entre las imágenes de la secuencia.



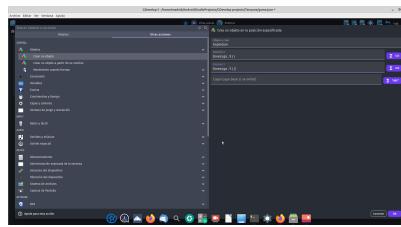
Tras esto, el objeto de la explosión (Explosion) estará listo. Cerraremos la ventana de edición del Sprite y nos iremos a la pestaña de "Escena sin título (Eventos)" para crear un nuevo evento, al que añadiremos una condición basada en

Q Elaboración propia a partir de GDevelop ([MIT License](#))

filtrar las condiciones disponibles, en este caso escribimos "Colisión" y configuramos el evento para que se lance cuando colisione un objeto "Proyectil" con un objeto "Enemigo".

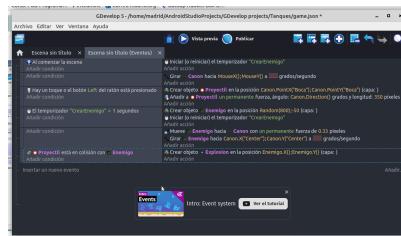
El siguiente paso es añadir la Acción en la parte derecha de la pantalla, que consiste en crear un objeto de tipo explosión, de forma que el evento queda programado para que cada vez que un proyectil impacte o colisione con un enemigo se cree una explosión. Configuraremos las coordenadas dónde se crea el objeto "Explosion" con las coordenadas ocupadas por el Enemigo tal y como se muestra en la figura 3, quedando el evento configurado como se muestra en la figura 4.

Figura 3



Q Elaboración propia a partir de GDevelop ([MIT License](#))

Figura 4

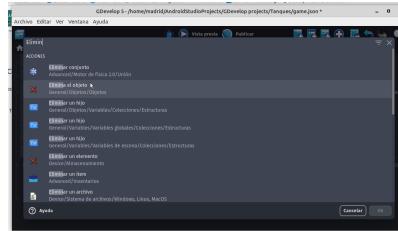


Q Elaboración propia a partir de GDevelop ([MIT License](#))

2.9.- Efecto final de explosión (I). Desaparición del tanque.

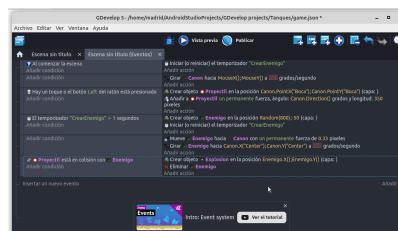
Además, haremos que los tanques enemigos, tras ser impactados y desencadenarse la explosión, desaparezcan. Del mismo modo haremos desaparecer el proyectil que ha impactado con el enemigo, para que no continúe su camino, sino que simulemos que ha provocado la explosión y desaparezca junto al tanque eliminado.

Para ello incorporaremos una segunda y una tercera Acción de tipo "Elimina el objeto" para este mismo evento como se muestra en las siguientes imágenes:



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

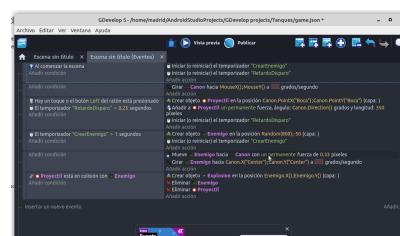


Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Si previsualizamos, veremos que ya funciona, provocándose las explosiones y desapareciendo los tanques impactados y los proyectiles.

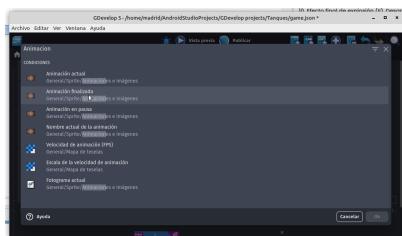
NOTA: Observa que puedes añadir una acción más al inicio de la escena para inicializar el Temporizador de retardo de disparo. Esta acción es opcional y provoca que haya que esperar un cuarto de segundo entre un dispara y otro para dotar a la escena de un poco más de dinamismo.



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

2.10.- Efecto final de explosión (II). Desaparición de estela.



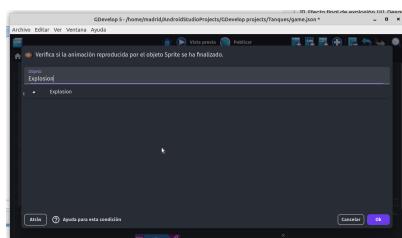
Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Ahora, tras jugar un rato, vemos que la estela o humo de cada explosión queda de manera permanente en la pantalla, dejando toda la escena negra, es decir, no desaparece, simplemente porque el objeto de la explosión realmente no se ha eliminado, sino que simplemente se ha quedado en su último estado (humo negro). Dependiendo de nuestro criterio,

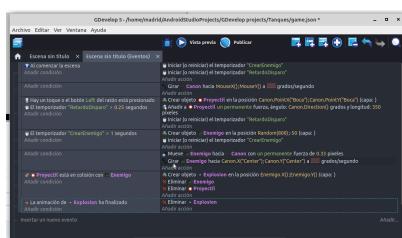
podemos decidir que dicha estela finalmente desaparezca, es decir, como si el humo pasado cierto tiempo se "dispara" quedando el fondo "limpio" de nuevo. Además, hay que tener en cuenta que al no desaparecer o finalizar dichos objetos, penalizará, si se juega durante un cierto tiempo, el rendimiento del juego, por mantenerse gran cantidad de objetos en la escena.

Para resolverlo haciendo que desaparezcan, debemos añadir un nuevo evento y sobre él una nueva condición de "Animación Finalizada".

Para crear la Acción de "Eliminar objeto" Explosión. Se realiza del mismo modo que en el apartado anterior, quedando de la siguiente forma:



Q Elaboración propia a partir de GDevelop (
[MIT License](#))



Q Elaboración propia a partir de GDevelop (
[MIT License](#))



Autoevaluación

Para poder repetir una acción cada X tiempo en GDevelop...

- Añadiremos una condición que compare el estado de un temporizador con una cantidad de tiempo.
- Añadiremos una acción que compare el estado de un temporizador con una cantidad de tiempo.
- Es necesario iniciar / reiniciar el temporizador mediante una condición.
- Es necesario iniciar / reiniciar el temporizador mediante una acción.

[Mostrar retroalimentación](#)

Solución

- 1.** Correcto (#answer-482_139)
 - 2.** Incorrecto (#answer-482_12576)
 - 3.** Incorrecto (#answer-482_12578)
 - 4.** Correcto (#answer-482_12580)
-

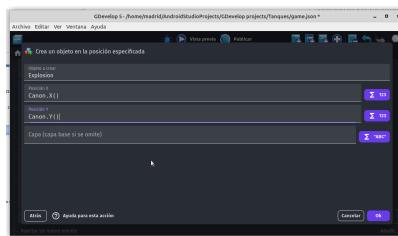
2.11.- Más colisiones. Tanque contra cañón: final del juego.

Ahora además de las colisiones de los proyectiles con los tanques, habrá que tener en cuenta las colisiones de los tanques con el propio cañón. Vamos a programar que **si se produce una colisión de un "Enemigo" con el objeto Cañón ("Canon") lanzaremos un objeto de tipo explosión** (por sencillez utilizaremos el mismo). Además, consideraremos que en ese momento el juego debe finalizar, lanzándose en el siguiente apartado el clásico mensaje de GameOver.

En primer lugar añadiremos el evento y sobre él agregaremos una condición de Colisión, donde rellenaremos los campos "Objeto" con *Enemigo* y de nuevo "Objeto" con el *Canon*. Además, habrá que crear para dicho evento dos acciones:

- La primera consistirá en crear la explosión con una acción de tipo "Crear un objeto en la posición especificada", haciendo donde esté situado el cañón configurando los campos "Posición X" como *Canon.X()* y "Posición Y" como *Canon.Y()* (figura 1).
- La segunda consistirá en eliminar el objeto *Canon*. Para ello usaremos una acción de "Eliminar el objeto" seleccionando *Canon* quedando los eventos tal y como se muestra en la figura 2.

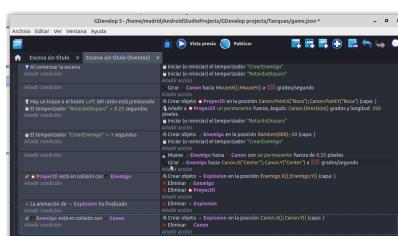
Figura 1



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 2



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

2.12.- Incorporación de texto GameOver.

Vamos ahora a incorporar el objeto que contiene el **texto GameOver** para **mostrarlo cuando se haya producido el impacto con el Cañón**. Para ello creamos un nuevo objeto, seleccionando el tipo > Texto (figura 1).

Damos a Ok y aparecerá una ventana donde podemos dar nombre al objeto de texto, concretamente se llamará "GameOver" y donde podemos modificar su texto, cambiando el tamaño a 48 y el color de la fuente a rojo (figura 2).

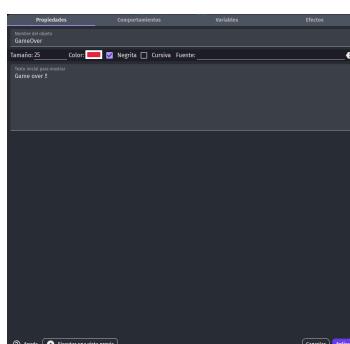
A continuación, arrastraremos con el clic izquierdo, el objeto GameOver desde el Editor de objetos hasta la escena. De esta forma el texto quedará superpuesto al fondo y al propio cañón (figura 3).

Figura 1



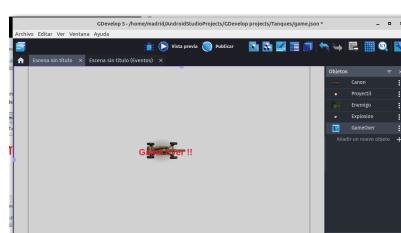
Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Figura 2



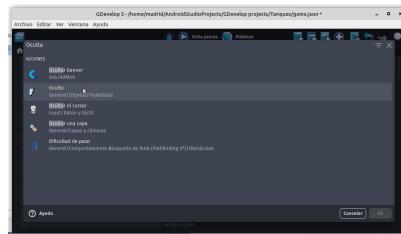
Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Figura 3

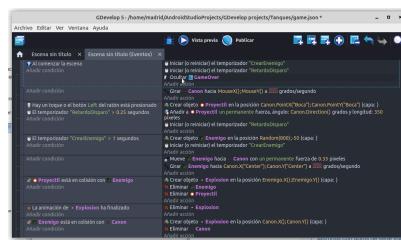


Q Elaboración propia a partir de GDevelop (
[MIT License](#))

Con esto, el texto quedará siempre visible. Lo que haremos será ponerlo oculto en el evento inicial, "Al comenzar la escena". Para ello añadimos una acción de tipo "Oculto", quedando:



Q Elaboración propia a partir de GDevelop (
[MIT License](#))



Q Elaboración propia a partir de GDevelop (
[MIT License](#))

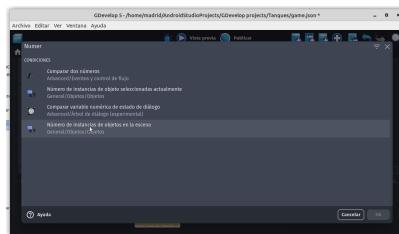
2.13.- Evento de control de fin de partida.

El momento de poner visible el texto de final de partida será en el momento en el que tras la colisión entre un "Enemigo" y el "Canon", este último haya desaparecido de la escena. Simplemente bastará con contabilizar cuantos cañones hay para poder determinar si ha de finalizar o no (habrá un cañón mientras el jugador no haya perdido, y 0 en el momento que el jugador pierda debido a la colisión de un tanque con el cañón).

Para llevarlo a cabo, añadiremos un nuevo evento y sobre él añadiremos una condición de tipo "Número de instancias de objetos en la escena" (figura 1).

Establecemos la configuración de la condición para que se lleve a cabo la acción asociada al evento en base a que el número de elementos "Canon" sea "igual a" 0, es decir, haya desaparecido por que haya colisionado con un "Enemigo" (figura 2).

Figura 1



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 2

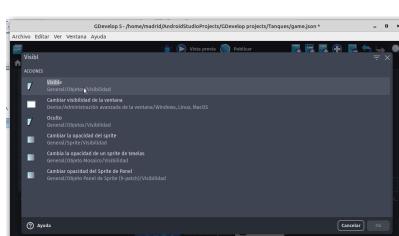


Q Elaboración propia a partir de GDevelop (

[MIT License](#))

La Acción asociada no será otra que hacer visible el texto que había sido ocultado al inicio de la partida (figura 3). Quedando de esta forma configurada como se muestra en la figura 4.

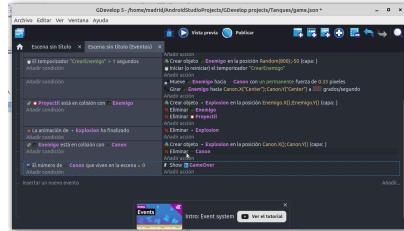
Figura 3



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 4



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

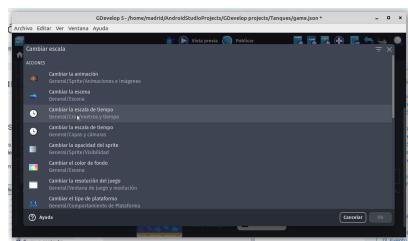
Nota: Si se desea puede añadirse un retardo de entre 0.5 y 1 segundo antes de mostrar el texto de Fin de Partida, del mismo modo a cómo se ha realizado en otros eventos del juego.

2.14.- Bloqueo del juego tras GameOver.

Por último, un efecto también necesario es finalizar el juego una vez que se ha producido el *GameOver*. Esto se debe principalmente al indeseable efecto que provoca el que sigan generándose tanques a pesar de haber finalizado el juego. Para ello, sobre el último evento, el que muestra el texto del Final de Partida, agregaremos una nueva acción. Esta debe ser de tipo "Cambiar la escala de tiempo de la escena" (figura 1).

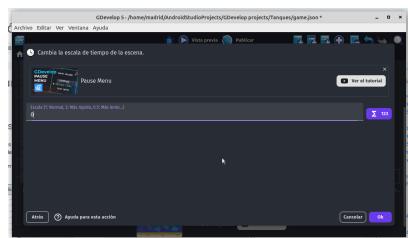
En el campo "Escala (1: Normal, 2: Más rápido, 0,5: Más lento...)" pondremos 0, con lo cual, el juego se parará (figura 2).

Figura 1



Q. Elaboración propia a partir de GDevelop (
[MIT License](#))

Figura 2



Q. Elaboración propia a partir de GDevelop (
[MIT License](#))

2.15.- Optimización de rendimiento. Finalización de hilos innecesarios.

Para optimizar el rendimiento del juego, debemos tratar de finalizar todos los hilos que se generen y que ya no tengan interés, porque son del todo innecesarios y una vez fuera de la vista del usuario lo único que hacen es sobrecargar el juego y penalizar su rendimiento. Concretamente, cuando el cañón dispara cada proyectil, si este sale fuera de los límites de la ventana seguirá recorriendo su trayecto de manera indefinida, aunque esté fuera de los límites de la ventana. Pero incluso con los proyectiles que impactan sobre un tanque sucede igual, ya que en lo que hemos programado hasta ahora dichos proyectiles, a pesar de haber provocado una explosión, siguen recorriendo su trayectoria, pudiendo darse incluso la circunstancia de que un mismo proyectil puede explosionar sobre dos o más tanques si estos se encuentran en la misma trayectoria. Lo cierto es que esto no sería muy real. Por ello vamos a tratar de "poner fin" o imponer una finalización a los proyectiles. En primer lugar, vamos a hacer que estos finalicen si salen de la ventana que ve el usuario. Para ello podríamos incorporar un evento que destruya los proyectiles una vez se encuentren a una distancia que consideremos adecuada, pero en lugar de eso, vamos a utilizar otra técnica que consistirá en incorporar un comportamiento sobre un objeto. Concretamente incorporaremos un comportamiento sobre un el objeto Proyectil.

Sobre el objeto Proyectil, en la parte derecha, damos clic derecho --> Editar comportamientos (figura 1). Aparecerá una ventana sobre la cual marcaremos el ítem "Destruir cuando sale fuera de pantalla" (figura 2).

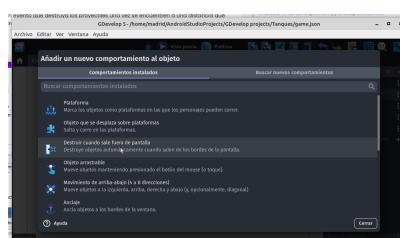
Figura 1



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 2



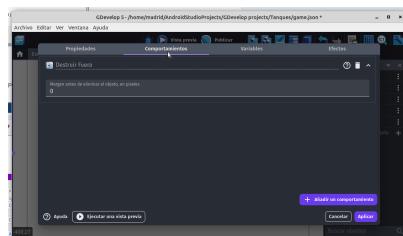
Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Dejamos el valor 0 (figura 3), quedando finalmente este evento tal y como se

muestra en la figura 4.

Figura 3



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Figura 4



Q Elaboración propia a partir de GDevelop (

[MIT License](#))

Con esto ya tendríamos nuestro videojuego de tanques finalizado.

Parte III - Desarrollo de videojuegos con Unity

1. Instalación de Unity

Unity, antes llamado Unity 3D, es una herramienta que nos permite crear videojuegos en 2D y 3D, éstos últimos con un increíble realismo, con gran facilidad. La calidad del de los juegos que podemos desarrollar es excelente, y el uso del programa es realmente sencillo. Estas características convierten a Unity en una excelente opción para los desarrolladores independientes y los estudios de videojuegos, cualquiera que sea su presupuesto.

Unity es una herramienta completa para el desarrollo de videojuegos. No solo es un motor de juegos de gran calidad y con numerosas características, también tiene un buen editor de mundos y trabaja con diferentes lenguajes de programación. Su uso es sencillo, y cualquier persona que tenga conocimientos básicos sobre programación podrá desarrollar un juego sencillo.

Unity está disponible para Windows, Mac y desde 2019 para Linux. Las herramientas necesarias para el desarrollo son:

- Unity 2020.3.30f1 o última versión LTS
 - Editor de código C# (Visual Studio Code o Visual Studio)
 - Aplicación Unity Remote para Android o iOS.
-

1.1 Usuario Unity

Lo primero que necesitamos para poder trabajar con Unity es una cuenta de usuario Unity.

En la dirección <https://unity.com/es> hacemos clic en la parte superior derecha de la ventana.



Una vez creado el usuario accedemos a Settings donde podemos cambiar el idioma al Español.

1.2 Descargar Unity

Antes de instalar Unity debemos saber si nuestro equipo cumple con los requisitos mínimos necesarios. Para la versión que vamos a utilizar en las prácticas deberemos consultar los requisitos en el siguiente enlace:

<https://docs.unity3d.com/2020.3/Documentation/Manual/system-requirements.html>

Unity nos permite dos tipos de descarga, por un lado podemos descargar la versión concreta que nos interesa o descargarnos la herramienta denominada Unity Hub con la cual podemos elegir qué versión descargar de una manera muy intuitiva.

Nosotros vamos a utilizar la herramienta Unity Hub que la podemos descargar de <https://unity3d.com/es/get-unity/download>

Debemos saber que podemos tener más de una versión de Unity instalada en el Hub.

Descargar Unity

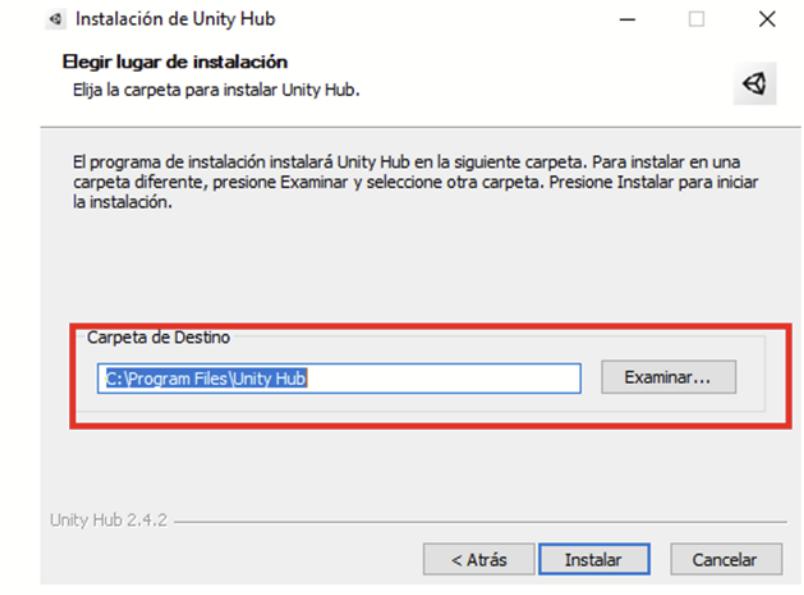
¡Bienvenido! Está aquí porque desea descargar Unity, la plataforma de desarrollo más popular del mundo para crear juegos multiplataforma y experiencias interactivas 2D y 3D.

Antes de descargar, elija la versión de Unity que sea adecuada para usted.

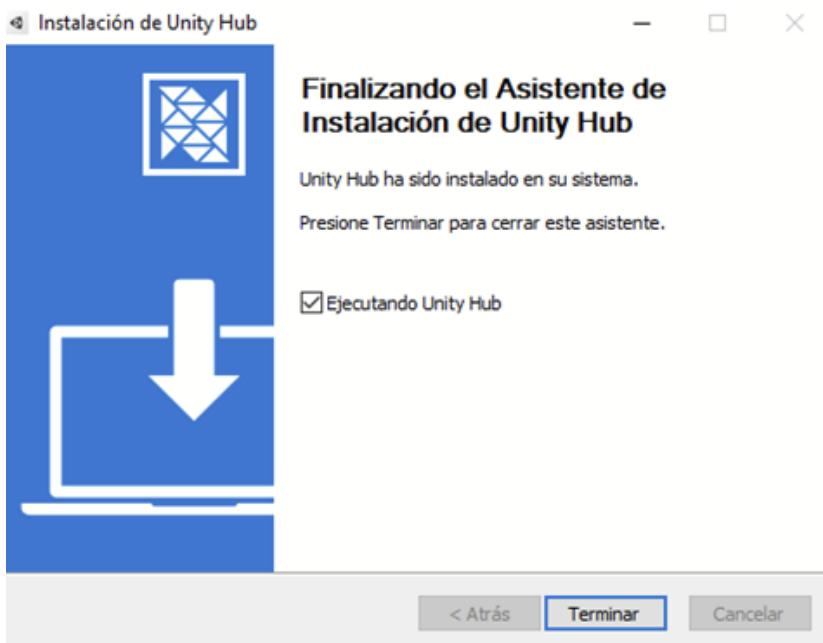
[Elige tu Unity + descargar](#) [Descarga Unity Hub](#)

[Descubrir más acerca del nuevo Unity Hub aquí.](#)

Lo primero que nos pregunta es dónde queremos instalarlo.

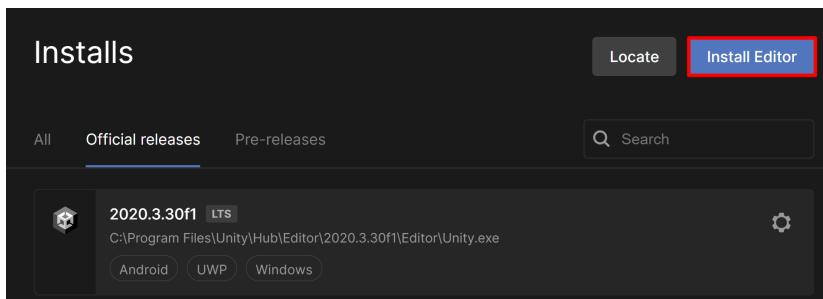


Se finaliza la instalación con una pantalla como la que se muestra en la siguiente figura.



Una vez se abre el Hub tenemos que iniciar sesión con la cuenta de Unity que hemos creado anteriormente.

Una vez que hemos iniciado sesión pasamos a instalar Unity. En el menú de la izquierda hacemos clic en Installs. En esta sección nos aparecerán las versiones de Unity que tenemos instaladas, en nuestro caso debería estar vacío. Para instalar una versión de Unity debemos hacer clic en Install Editor.



A continuación, deberemos seleccionar la versión a instalar. Para nuestro proyecto instalaremos la versión Unity 2020.3.30f1.

Cuando seleccionemos la instalación tenemos que asegurarnos que instalamos al menos los siguientes componentes. En cualquier caso se pueden instalar después desde el Hub.

- Android Build Support

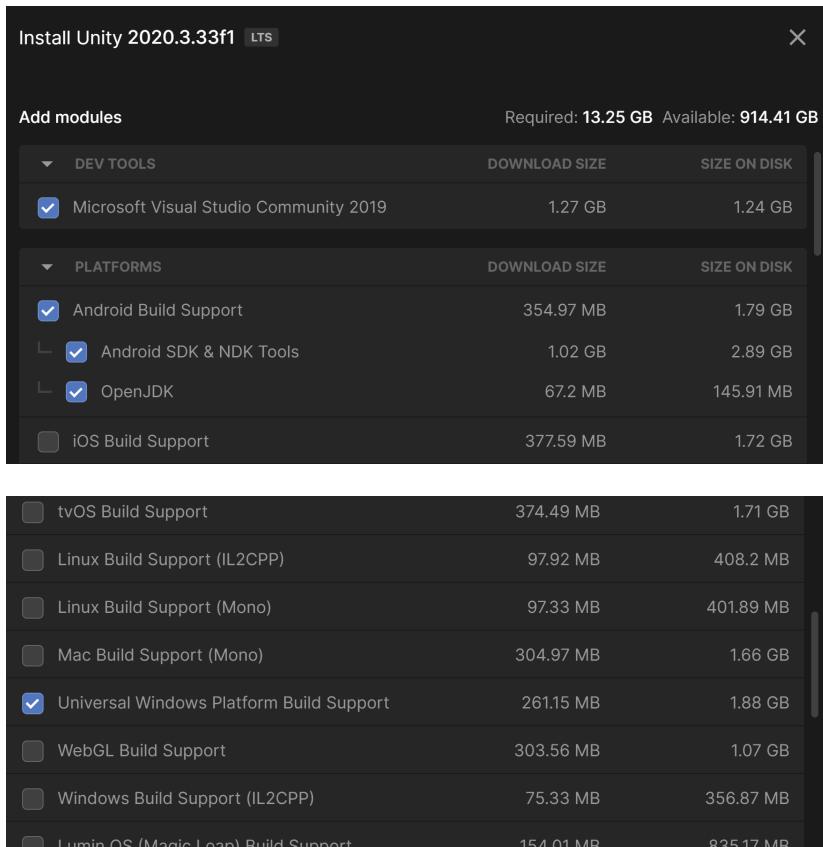
Los siguientes son opcionales, aunque recomendados:

- Android SDK & NDK tools. Si tenemos instalado Android Studio siempre podemos usar los que ya estén instalados, aunque habrá que configurarlo manualmente dentro de Unity.
- OpenJDK. Si ya tenemos instalada otra JDK de Java tampoco es necesario
- Microsoft Visual Studio Community. Si está instalado otro editor de C#, no es necesario (si, por ejemplo, ya lo tienes instalado para el módulo de Desarrollo de Interfaces).

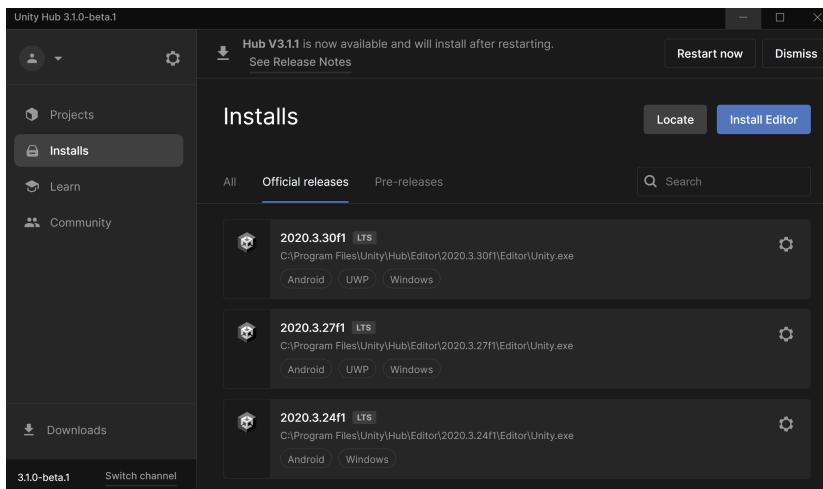
Nosotros vamos a seleccionar los siguientes módulos para instalar:

- **Herramientas de Desarrollo.** Necesitamos Microsoft Visual Studio para el desarrollo de los scripts (salvo que ya lo tengas instalado o trabajes con otro editor de C#)

- **Plataformas.** Nosotros vamos a escoger Android Build Support, Universal Windows Platform Build Support y Documentation.



Ahora tendremos que aceptar las licencias de cada elemento instalado. Una vez instalados todos los componentes nuestro Unity Hub será similar al de la siguiente figura.

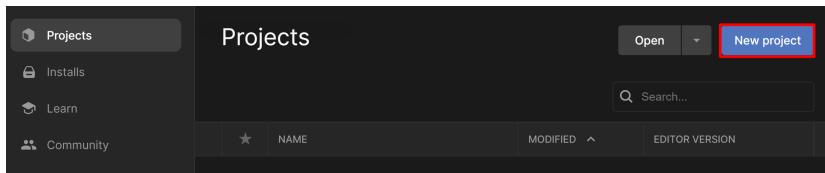


El siguiente paso es activar la licencia. En la opción de Configuración en el menú lateral (ícono engranaje) seleccionamos la opción Licenses y hacemos clic en Activate New License. Para realizar el proyecto de este módulo utilizaremos la licencia personal y la opción I don't use Unity in a professional capacity.

Ya tenemos Unity disponible para empezar a crear nuestros proyectos.

1.3 Creando nuestro primer proyecto

Nuestro último paso crear nuestro proyecto. En el menú de la izquierda seleccionamos Projects y luego hacemos clic en el botón New Project en la parte superior derecha.



Los templates elegibles serán similares a estos:

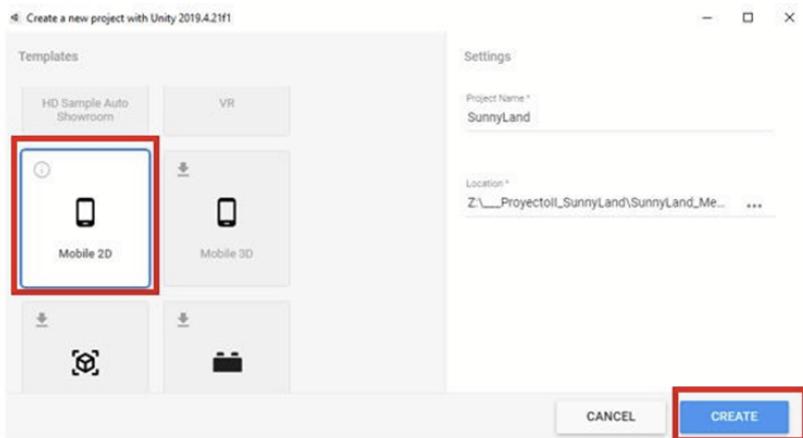
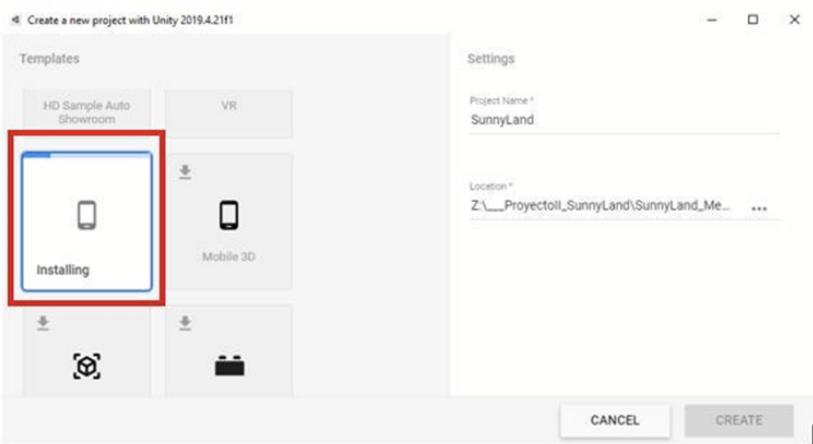
- 2D
- 3D
- 3D with Extras
- High Definition RP
- Universal Reder Pipeline
- 2D Platformer Microgame
- FPS Microgame
- Karting Microgame
- HD Sample Auto Showroom
- VR
- Mobile 2D
- Mobile 3D
- AR
- LEGO Microgame

El tipo de juego que vamos a desarrollar es Mobile 2D por lo que tendremos que descargar el template asociado si no lo tenemos disponible.

Mobile 2D

This project template provides a starting point for 2D mobile development in Unity.

[CANCEL](#) [INSTALL](#)



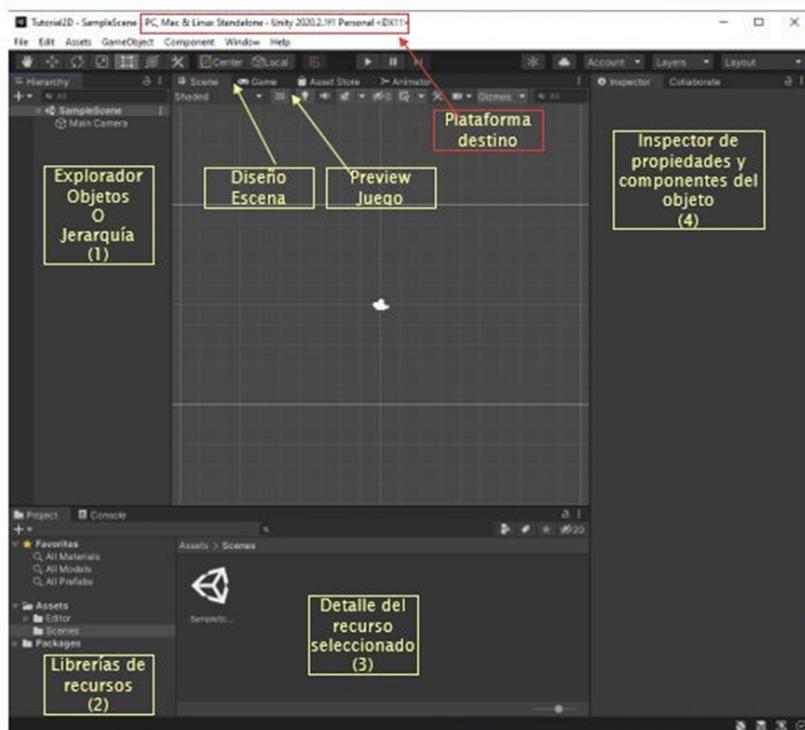
Ahora nos queda esperar a que se cree el proyecto.

2. Conociendo la interfaz

La interfaz gráfica de Unity está dividida en 5 secciones principales que se conoce como Vistas, nos encontramos con las siguientes:

- Vista de Escena
- Vista de Videojuego
- Vista de Jerarquía
- Vista de Proyecto
- Vista de Inspector

En la siguiente figura podemos ver la interfaz gráfica.



Al crear un proyecto lo primero que hace Unity es cargar los recursos. Una vez finalice la carga se verá el mundo del videojuego que estamos creando. Por defecto sólo habrá en él una cámara principal, se podrá ver como el único elemento en la lista de Vista de Jerarquía, su nombre es Main Camera.

En Vista de Jerarquía se encontrará todos los objetos de nuestro juego. Los objetos pueden ser seleccionados tanto haciendo clic sobre ellos en la Vista de Escena como en la Vista de Jerarquía. Si seleccionamos un objeto en la Vista de Escena podremos comprobar si hemos seleccionado el objeto correcto porque se mostrará iluminado también en la Vista de Jerarquía.

2.1 Vista de proyecto

La vista de proyecto es esencialmente una librería de assets para el proyecto de nuestro juego.

Todos los componentes del juego que creemos desde el editor y todos los objetos que importemos como modelos 3D, texturas, efectos de sonido, música etc. se guardarán ahí.

Como este panel contiene todos los assets del juego, y no solo los que están en la escena actual, es importante mantener una buena estructura.

Se pueden crear carpetas y colocar los objetos dentro de esas carpetas para crear un jerarquía de carpetas. Puesto que un proyecto de juego completo contendrá muchos assets (en algunos juegos una cantidad muy grande de assets) es una buena idea crear una estructura que sea fácil de usar por el equipo que trabaja en el juego antes de empezar a desarrollar el juego.

La vista de proyecto es muy sencilla y funciona como cualquier gestor de ficheros. Eso no quiere decir que podamos crear la estructura de nuestras librerías o movemos assets utilizando el explorador de archivos del sistema operativo, ya que Unity podría perder enlaces y dependencias importantes. Como práctica general debemos usar las herramientas de organización de la vista de proyecto para crear la estructura y organizar los assets puesto que Unity seguirá su localización.

En la parte superior de la vista de proyecto está el botón Create, que mostrara una lista desplegable con varias opciones de creación. Podremos crear carpetas, scripts, shaders, animaciones y otros tipos de objetos usando este panel.

Si hacemos doble clic sobre un ítem en la librería nos permitirá renombrarla.

Podemos hacer clic y arrastrar carpetas e ítems para organizar la estructura.

Podemos importar assets como archivos de audio, texturas, modelos etc. haciendo clic derecho y seleccionar “Import Asset”.

2.2 Vista de escena

La escena es el área de construcción de Unity donde construimos visualmente cada escena de nuestro juego.

Los juegos creados en Unity están divididos en escenas al igual que muchos motores, por ejemplo Unreal Engine.

La vista de escena es un entorno 3D para crear cada escena. Trabajar con la vista de escena, en la forma más sencilla, sería arrastrar un objeto desde la vista de proyecto a la vista de escena que colocará el objeto en la escena, entonces podremos posicionarlo, escalarlo y rotarlo sin salir de la vista de escena.

La vista de escena es también el lugar donde se editan los terrenos (esculpiéndolos, pintando texturas y colocando elementos), colocamos luces y otros objetos.

Cuando trabajamos con la vista de escena es útil usar el espacio para agrandar la vista de escena hasta completar el editor completo.

2.3 Vista de juego

En la vista de juego se obtiene una previsualización de nuestro juego. En cualquier momento podemos reproducir nuestro juego y jugarlo en esta vista.

2.4 Vista de jerarquía

La vista de jerarquía contiene todos los objetos en la escena actual. Cualquier objeto que coloques en la escena aparecerá como una entrada en la jerarquía.

La jerarquía también sirve como método rápido y fácil para seleccionar objetos en la escena. Si queremos por ejemplo, seleccionar un objeto de la escena pueden seleccionarlo desde la jerarquía en lugar de movernos por la escena, encontrarlo y seleccionarlo.

Cuando un objeto es seleccionado en la jerarquía también lo es en la vista de escena, donde podemos moverlo, escalarlo, rotarlo, borrarlo o editarlo. El inspector también mostrará las propiedades del objeto seleccionado, de esta forma la jerarquía sirve como una herramienta útil para seleccionar rápidamente objetos y editar sus propiedades.

2.5 Vista de inspector

El inspector es esencialmente un panel de propiedades. Si seleccionamos objetos entonces se mostrarán las propiedades del objeto donde podemos personalizar las características del objeto. El inspector cambia según el objeto que seleccionemos.

Por ejemplo, si seleccionamos una luz o cámara, el inspector nos permitirá editar varias propiedades de la luz o de la cámara. Además, el inspector sirve como panel de herramientas para ciertos tipo de objetos, por ejemplo si seleccionamos un terreno el inspector nos mostrará las opciones de terreno y también el editor con herramientas como esculpir, texturizar, etc.

3. Scripting en Unity

Los motores de videojuegos deben permitir al programador crear sus propios scripts porque es muy común que muchas funcionalidades deseadas no estén previamente implementadas en el motor y sea necesario añadirlas por medio de scripts o a veces, el programador simplemente desea ajustar el comportamiento de algunas configuraciones por defecto. Esta herramienta diversifica las posibilidades de desarrollo.

La mayoría de motores gráficos utilizan lenguajes de script, muchos de ellos están basados en Javascript o similares. Para las personas que no saben programar, Javascript es un buen lenguaje para comenzar porque tiene un bajo nivel de iniciación y una estructura sólida. Nosotros nos decantaremos por C# en el desarrollo de los scripts en Unity, por ser similar a Java, lenguaje que conocemos.

El scripting indica a nuestros GameObjects cómo comportarse; son los scripts y los componentes añadidos a los GameObjects, y cómo interactúan entre sí, lo que crea la jugabilidad. Ahora bien, el scripting en Unity difiere de la pura programación. Si pensamos en una aplicación en ejecución, debemos comprender que en Unity no se necesita crear el código que ejecuta la aplicación, ya que Unity lo hace por nosotros. En lugar de ello, nos centramos en la jugabilidad en los scripts.

Unity ejecuta un bucle enorme. Lee todos los datos que hay en la escena de un juego. Por ejemplo, lee las luces, las mallas, cuáles son los comportamientos, y procesa toda esta información.

Le indicamos a Unity con las instrucciones que escribimos en los scripts que tiene que hacer, y Unity las ejecuta frame tras frame tan rápido como puede.

Lograr una velocidad de frames alta no solo significa que tu juego se verá más fluido, sino que tus scripts se ejecutarán también con más frecuencia, haciendo que los controles sean más receptivos.

Un script debe estar vinculado con un GameObject en la escena para poder ser invocado por Unity. Los scripts se escriben en un lenguaje especial que Unity puede entender. Y, es a través de este lenguaje que podamos hablarle al motor y darle nuestras instrucciones.

El lenguaje que se usa en Unity se denomina C# (se pronuncia C sharp). Todos los lenguajes que Unity emplea son lenguajes de scripting orientados a objetos. Al igual que cualquier lenguaje, los lenguajes de scripting tienen sintaxis, o partes de diálogo, y las partes principales son las clases, dentro de estas tendremos los atributos que contendrán valores y referencias a objetos y los métodos que permiten trabajar sobre los atributos de las clases. También existen otros métodos que se ejecutan automáticamente en Unity, estos son: Awake, Start, Update, FixedUpdate y LateUpdate.

El scripting consiste principalmente en comparar estos objetos y sus estados y valores actuales. Se basa en la determinación lógica de un resultado o resolución.

3.1 Clases

Las clases son colecciones de atributos y métodos. Estos conceptos ya deben ser conocidos por nosotros de otros módulos del ciclo. Por ejemplo, este script es una clase:

```
6 public class DemoScript: MonoBehaviour {
7
8     public Light myLight;
9
10    void Awake () {
11        int myVar = AddTwo(9,2);
12        Debug.Log(myVar);
13    }
14
15    void Update () {
16        if (Input.GetKeyDown ("space")) {
17            MyFunction ();
18        }
19    }
20
21    void MyFunction () {
22
23        myLight.enabled = !myLight.enabled;
24    }
25
26    int AddTwo (int var1, int var2) {
27        int returnValue = var1 + var2;
28        return returnValue;
29    }
30
31    }
32
33}
34 }
```

El nombre de la clase debe coincidir con el nombre de archivo del script C# para que funcione. Y después de anexarla a un GameObject, tiene que derivarse de otra clase llamada MonoBehaviour.

En Unity, si creamos una clase personalizada, como en el ejemplo a continuación, hay que serializarla. Esto significa que se convertirá a datos simples que Unity puede examinar en el inspector.

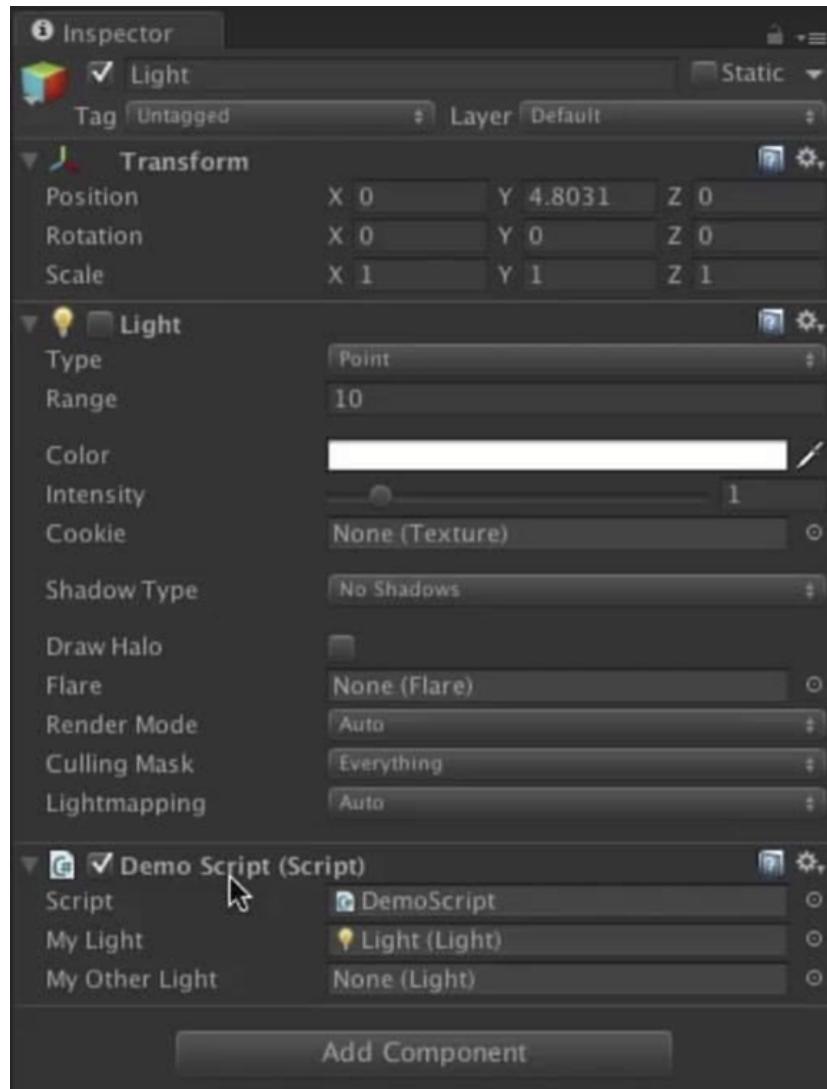
```

1 using UnityEngine;
2 using System.Collections;
3
4 [System.Serializable]
5 public class DataClass {
6     public int myInt;
7     public float myFloat;
8
9 }
10
11 public class DemoScript : MonoBehaviour {
12
13     public Light myLight;
14     public DataClass myClass;
15
16     void Awake () {
17         int myVar = AddTwo(9,2);
18         Debug.Log(myVar);
19
20     }
21
22     void Update () {
23         if (Input.GetKeyDown ("space")) {
24             MyFunction ();
25
26         }
27
28     }
29
30     void MyFunction () {
31         myLight.enabled = !myLight.enabled;
32     }

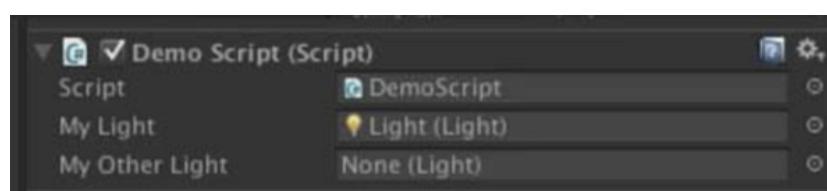
```

Un aspecto importante de los atributos es el tipo. Un tipo define qué tipo de valor mantiene el atributo en la memoria, puede ser un número, texto, o tipos más complejos, como los que se observan en la imagen a continuación: Transform, Light y Demo Script son, en efecto, referencias a Componentes. Unity necesita saber qué tipo de objeto es y cómo manejarlo.

Para nombrar a los atributos debemos recordar que al asignar un nombre a las variables no pueden comenzar por un número, y que no pueden contener espacios. Por lo tanto, hay un estilo para escribir los nombres. En C#, la convención de nomenclatura es camelCase: se comienza con una minúscula y añades palabras, sin espacios, comenzando por una mayúscula, por ejemplo “miAtributo”.



Cuando Unity compila el script, hace que las variables públicas sean visibles en el editor, no así las privadas. Observa la imagen a continuación del inspector.



A la hora de definir los métodos debemos recordar que los métodos comienzan con el valor que devuelve, seguido por el nombre del método, y después los parámetros entre paréntesis (si fuese el caso). Los nombres de los métodos comienzan por una mayúscula y el cuerpo de éste va entre llaves. Este es un ejemplo de cómo escribir un método:

```
13  
14     void MyFunction () {  
15  
16     }  
17 }
```

Debemos recordar que para llamar a un método debemos hacerlo con un objeto de la clase en la que están definidos.

3.2 Funciones automáticas

Los scripts manipulan los objetos a través de funciones. Hay una serie de funciones que se ejecutan automáticamente en Unity. A continuación se describe que hace cada una de ellas.

- **Awake** se invoca solo una vez cuando se menciona ese componente. Si un GameObject está inactivo, entonces no podrá invocarse hasta que se le active. Sin embargo, Awake se invoca incluso si el GameObject está activo pero el componente no está habilitado (con la pequeña casilla de verificación junto a su nombre). Puedes usar Awake para inicializar todos los objetos a los que necesitemos asignar un valor.
 - **Start** al igual que Awake, Start se invocará si un GameObject está activo, pero solo si el componente está habilitado.
 - **Update** se invoca una vez por frame. Aquí es donde colocamos el código para definir la lógica que se ejecuta continuamente, como animaciones, AI y otras partes del juego que tienen que actualizarse continuamente.
 - **FixedUpdate** asociado a la física del juego.
 - **LateUpdate** es una función que es similar a Update, pero LateUpdate se invoca al final del frame. Unity analizará todos los objetos de juego, encontrará todas las Updates, e invocará las LateUpdates. Esto es bueno para cosas como la cámara. Digamos que si queremos mover un personaje en el juego. Y entonces, él tropieza con otro personaje y termina en otra posición. Si movemos la cámara al mismo tiempo que el personaje, habrá una sacudida, y la cámara no va a estar donde tendría que estar. Así que, básicamente, es un segundo loop que resulta muy práctico.
-

Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0 (<http://creativecommons.org/licenses/by-sa/4.0/>)