UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Realización de un entorno de desarrollo para programas 2D basado en Phaser

Francisco José Parreño Rodríguez
Tutor: Carlos Aguirre Maeso

Mayo 2019

Realización de un entorno de desarrollo para programas 2D basado en Phaser

AUTOR: Francisco José Parreño Rodríguez
TUTOR: Carlos Aguirre Maeso

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo 2019

Resumen (castellano)

En la actualidad, el sector de los videojuegos está cobrando relevancia a nivel mundial convirtiéndose en una potencia dentro del mercado económico. Este auge es debido no solo a que han dejado de ser juzgados por ser elementos únicamente de ocio, y de un sector marginal de la población, debido a la moda actual del "gaming" y "gamer". Ya que además, se ha comenzado a trasladar su visión a una herramienta útil y transmedia, que permite al usuario adquirir conocimientos de una forma indirecta y sencilla, mediante su uso, con los llamados *serious game*, y permitir un aprendizaje más interactivo para cualquier persona, en edad y sector social.

A su vez, el continuo crecimiento de los lenguajes como JavaScript, y los estándares web de HTML5, han permitido extrapolar el desarrollo web en múltiples direcciones, abarcando diversos sectores de la programación como puede ser la programación gráfica, con los videojuegos por ejemplo; aportando la capacidad de multiplataforma, a día de hoy indispensable debido a que la informática ha dejado de ser exclusiva de un usuario sentado en su escritorio, a pasar a estar en el bolsillo de gran parte de la población en el Smartphone.

Con esta premisa, nos centraremos en los Frameworks JavaScript, que permiten el desarrollo de videojuegos. En concreto, nos enfocaremos para este trabajo en "Phaser", Framework diseñado para el desarrollo de videojuegos 2D, multiplataforma.

Este Trabajo de Fin de Grado abordará el tema de la realización de una GUI, *Graphic User Interface*, para dicho Framework, con el fin de permitir a cualquier usuario, a nivel profesional y no profesional, poder desarrollar videojuegos de una manera fácil e intuitiva.

Para ello, se ha creado mediante el uso de HTML5, CSS3, y JavaScript, un entorno de desarrollo web que permita al usuario realizar:

- El diseño de niveles de un videojuego 2D.
- Modificación del "mundo" que conformará el nivel.
- La inserción y posicionamiento de elementos dentro del nivel mediante técnicas de Drag&Drop.
- Modificación de las propiedades básicas tanto del nivel como los elementos del mismo.
- Inserción de físicas básicas en los elementos introducidos.

El resultado final, es un entorno de desarrollo, intuitivo y sencillo con un estilo visual similar a algunas aplicaciones del mercado como Photoshop. Y que mejora la experiencia del usuario, además de acercar la programación de videojuegos a personas fuera del sector, tanto como herramienta profesional como amateur.

Abstract (English)

Nowdays, the video game sector, is increasing its relevance worldwide becoming a power within the economic market. This boom is due not only to the fact that they have ceased to be judged as only leisurely elements of a marginal sector of the population. That's due to the current gaming and gamer fashion. In addition, it has begun to change its vision to auseful and transmedia tool, which allows the user to acquire knowledge in an indirect and simple method, through its use, and enable a more interactive learning for anyone who use it, from any age and social sector.

On the other hand, the continuous growth of languages such as JavaScript and HTML5 web standards, have made it posible to extrapolate the web development in multiple directions, occupying some sectors of programming such as graphic programming, like the video games. Contributing the multiplatfrom capacity, nowdays indispensable beacuse the computer has stopped being exclusive from a user sitting front his desk, to happen to be in the pocket of the main part of the poblation in the Smartphone.

With this premise, we'll focus on the JavaScript Frameworks, which allow us the development of video games. Specifically we will focus for this work on "Phaser", a Framework designed for development 2D videogames, and multiplatform.

This End-of-Grade Paper will address the issue of the creation of a GUI, Graphic User Interface, from this Framework, in order to allow any user, professional and non-professional, to develop video games in an easy and intuitive way.

For this, we have created a web development environment in HTML5, CSS3, and JavaScript, that allows the user to perform:

- Level design of a 2D videogame.
- Change the "world" that will make up the level.
- The insertion and positioning of elements throught the level using Drag & Drop techniques.
- Change the basic properties of the level and elements within.
- Insertion of basic physics into these elements.

The final outcome is a development environment, intuitive and simple, with a visual style that remember to some applications in the market like Photoshop. And that improves the user experience, in addition to bringing video games programming to people inside and outside the sector, as a tool both profesional and amateur.

Palabras clave (castellano)

GUI, Phaser, Framework, HTML5, CSS3, JavaScript, videojuegos, Serious Game, web, asset, motor gráfico, físicas, propiedades, 2D, require.js, Drag&Drop, IDE

Keywords (inglés)

GUI, Phaser, Framework, HTML5, CSS3, JavaScript, video games, Serious Game, web, asset, engine, physics, properties, 2D, require.js, Drag&Drop, IDE

Agradecimientos

Me gustaría dedicar este trabajo de fin de grado a todos los que aquellos que han estado conmigo estos años. En especial medida a mis padres que no han dejado de animarme a continuar y no rendirme durante este tiempo. Y me han apoyado cada día en este largo viaje.

Gracias especialmente, a Mary por estar estos años conmigo, soportándome y haciéndome mucho más amena la carga de la carrera, dándome su apoyo incondicional y ayudarme en los momentos más duros como en esas tardes de programación que se hacían eternas.

Gracias igualmente también a mi mejor amigo, Alejandro por estar siempre ahí.

Gracias a Carlos Aguirre por ayudarme en la selección del TFG, y su atención para todo el papeleo cosa que no es mi fuerte y que me salvó en el último momento de seleccionar TFG.

Por último querría agradecer a mis compañeros de grado que siguen estando o no… Moisés, Wizi, Chache, Tavi, Pablo, Borja, Alberto… Porque han hecho que este camino sea mucho más divertido y transitable.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

El constante crecimiento de las tecnologías está permitiendo que los videojuegos dejen de ser exclusivos de aparatos domésticos, llegando a los pequeños dispositivos, caracterizados por ser de fácil acceso y usabilidad.
Gracias a esto, los videojuegos han evolucionado hasta llegar a ser una parte inherente de la sociedad, siendo utilizados por gente de todas las edades y estratos sociales, dado su capacidad de adaptación (De jugabilidad, y ambientación) y aplicación (Usos médicos, educativo, de ocio…), así como la gamificación de la vida diaria.

Además, el sector *gamer*, está sufriendo una regresión a los clásicos de los videojuegos (Juegos en 8 bit, 2D…) que sirve de puente generacional entre los jugadores que crecieron en la era del pixel, y los nuevos jugadores. Hecho que podemos corroborar con grandes éxitos modernos como Shovel Knight, ganador de mejor juego del año en 2014, Minecraft como elemento educativo en países como Finlandia, la franquicia Mario Bros, y la resurrección de viejos clásicos como Mega-Man con Mighty No.9.

Aparte de esto, uno de los lenguajes más potentes del momento podría considerarse JavaScript, debido a su increíble capacidad de escalabilidad, modularidad y portabilidad al poder usarse en medios webs.

Por todo esto, se ha decidido desarrollar un entorno gráfico que permita de una manera más sencilla, y menos artesanal, el desarrollo; o al menos los primeros pasos, de videojuegos en 2D para entornos web, utilizando el Framework de código libre, Phaser.

1.2 Objetivos

El objetivo de este trabajo, será la implementación de un entorno gráfico de desarrollo para el Framework Phaser, que permita la creación de niveles de manera sencilla mediante técnicas de Drag&Drop.

Se busca que este entorno proporcione un acceso a la mayor cantidad posible de propiedades y capacidades de los elementos de un juego, y su entorno. Así como obtener la mayor modularidad posible de las partes para su futuro mantenimiento y escalabilidad.

Tras el análisis del proyecto, se optó por realizar un patrón MVC, donde la vista sería la propia GUI, realizada mediante los lenguajes HTML5 y CSS3 y JavaScript, en concreto la librería JQuery.js, que mostrará el mundo, los elementos, y las propiedades de estos. El modelo de datos, que estará dividido entre variables de control dentro del propio HTML, como variables JSON donde tendremos la información de los assets que podremos añadir al mundo. Mientras que el controlador estará íntegramente realizado en JavaScript, junto a la librería Require.js, para permitir la modularidad del mismo.

1.3 Organización de la memoria

La memoria consta de 6 capítulos, que son descritos en detalle a continuación:

- **Estado del arte:** En este capítulo se presentará el Framework de Phaser, así como modelos de GUI que resuelven el mismo problema que este TFG.
- **Análisis y diseño:** En este capítulo abordaremos el análisis y el diseño del controlador y el modelo.
- **Phaser:** En este capítulo haremos una pequeña introducción al Framework Phaser.
- **Desarrollo de la vista:** En este capítulo hablaremos del desarrollo del apartado visual de la GUI.
- **Desarrollo del modelo:** En este apartado, hablaremos de cómo se han realizado los diferentes métodos que conforman el controlador.
- **Integración, pruebas y resultados:** En este punto, se hablará de los resultados obtenidos mediante ensayos con los diferentes métodos.
- **Conclusiones y Trabajo futuro:** En este último capítulo, trataremos el tema de las conclusiones finales tras la realización del trabajo, así como las futuras posibles mejoras.

2 Estado del arte

Este *boom* de la programación gráfica, así como las facilidades que se tienen a día de hoy para la distribución, gracias a los medios digitales, ha desembocado en un pensamiento colectivo de que realizar un videojuego es una manera fácil y rápida de ganar dinero, llegando al punto de que gente ajena al sector de la informática se planteen ganarse la vida de este modo, realizando juegos en los que se premia el apartado gráfico, a una mecánica, jugabilidad o historia trabajada.

Estos hechos han propiciado un incremento de empresas pequeñas, o particulares, que tratan de hacerse un hueco en el mercado, creando videojuegos a base de programas como Gamemaker o Three.js como herramientas quizá con un público más amateaur, pero no por ello menos elaborados, y que tratan de emular el sistema de ventanas y módulos llevado a cabo por GUIs más clásicas como podrían ser Unity, Unreal o CryEngine, e incluso el más novedoso Lumber Yard. Por ello, de cara al desarrollo de nuestra propia GUI hemos decidido realizar un estudio previo de estas GUIs.

2.1 Entornos Amateaur

2.1.1 Gamemaker

Este entorno y motor de desarrollo de videojuegos llevado a cabo por YOYO Games, lo primero que nos encontramos es que nos ofrece diversas herramientas destinadas al desarrollo, desde un sistema de programación mediante enlaces con Drag&Drop de elementos parecido a los Blueprints que veremos más adelante en Unreal, y que nos permite entre otras cosas la codificación de propiedades del juego, físicas y comportamientos, sin tener un conocimiento elevado de programación, ni la restricción de un lenguaje. Además nos proporciona un sistema de Debug, así como un editor de código en tiempo real.

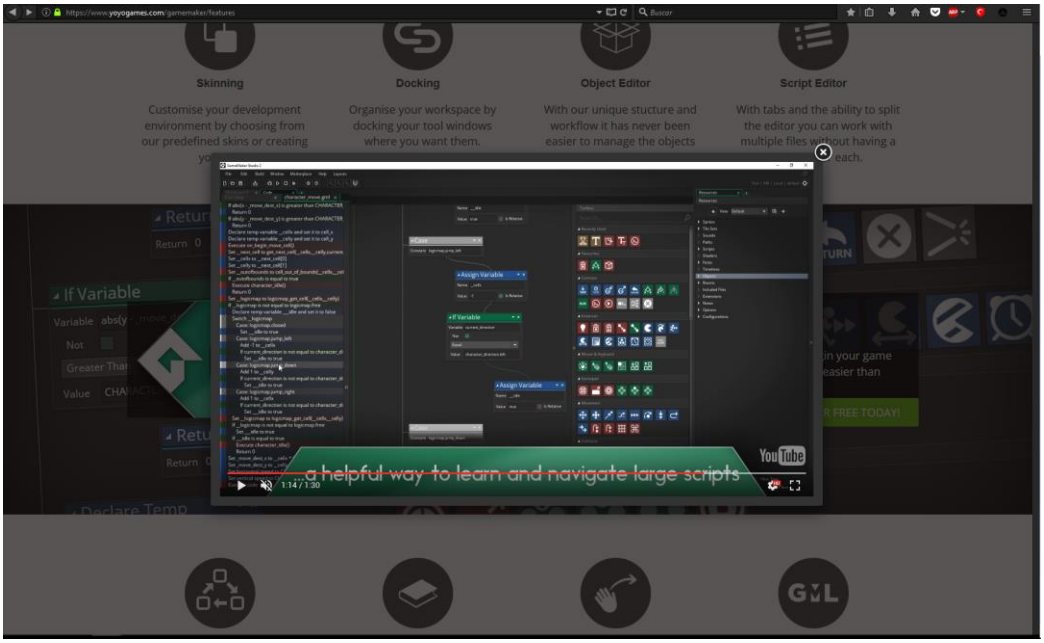


Ilustración 1 Detalle editor código Gamemaker.

En segundo lugar, tenemos la herramienta de edición del mundo, donde podemos observar los elementos que lo pueblan, recolocarlos, y modificar propiedades tales como tamaño, orientación, opacidad o profundidad de estos. Nos permite un sistema de gestión de ficheros que nos permite manejar con carpetas los Assets que utilicemos. Y además nos proporcionar un pequeño editor de código pensado para las colisiones e interacciones de los objetos en “caliente”. Que junto al editor de TileSets proporciona una manera rápida y sencilla de preparar el mundo, incluso con generación aleatoria de entornos.

Ilustración 2 Detalle editor de nivel Gamemaker.

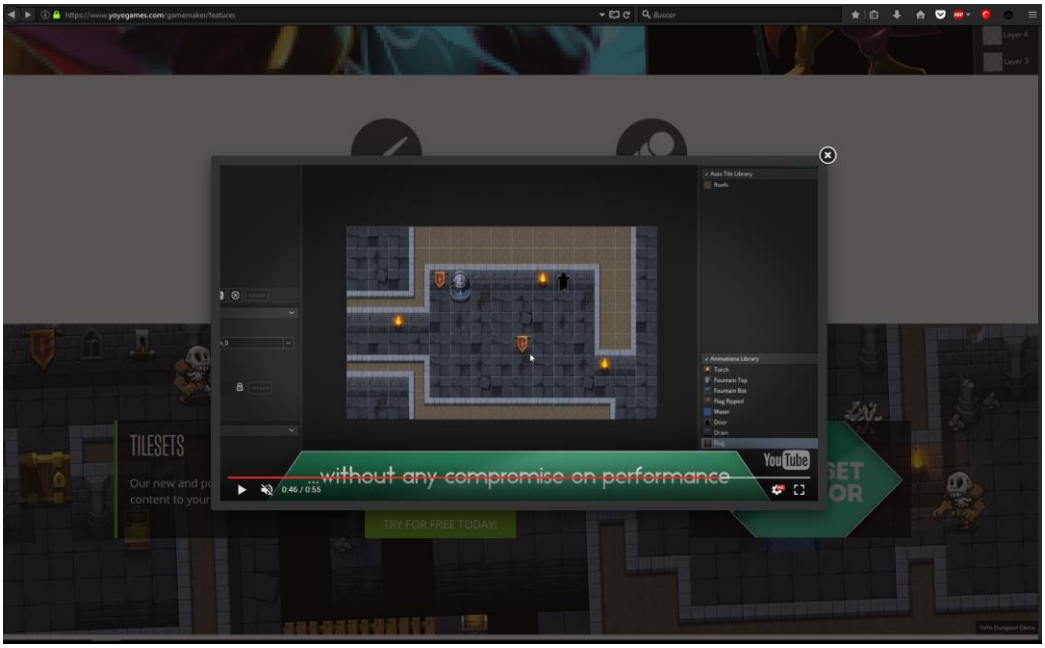


Ilustración 3 Detalle editor de TileSets Gamemaker.

Otras de las herramientas que dispone esta GUI es un pequeño pero potente editor de imágenes que nos permite realizar modificaciones, crear e incluso animar nuevos Assets, en tiempo real, desde el mismo programa sin necesidad de recurrir a terceros programas durante el desarrollo, hecho que agiliza el desarrollo.

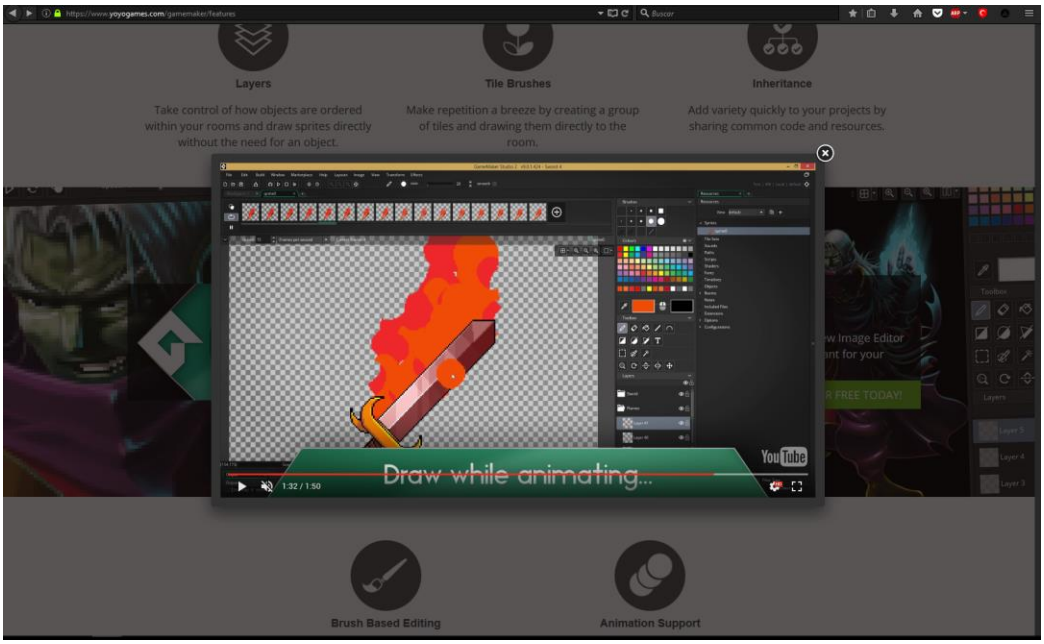


Ilustración 4 Detalle del editor de animaciones y dibujo Gamemaker.

Por último, no podemos olvidar el gestor de físicas que al igual que el editor de código nos brinda la posibilidad de un desarrollo Drag&Drop de elementos, así como un sistema de iluminaciones y sombras en la segunda versión del mismo. Que confiere mayor dinamismo y profesionalidad a nuestro juego.

Además posee una store propia donde los usuarios pueden subir contenido para poder ser utilizado por terceros en sus propios juegos.

2.1.2 ThreeJS Editor

Este entorno web está diseñado para el desarrollo de juegos con el Framework de JavaScipt, Threejs. Para empezar podemos observar como en él tan solo hay una ventana que se compone a su vez de diversos módulos. Uno principal donde tiene lugar la vista del mundo y donde llevaremos a cabo la mayor parte del desarrollo, y una barra lateral donde tenemos acceso a diversos factores y propiedades tanto del mundo, aquí llamado “scene”, como de los elementos que lo pueblan. Nos proporciona además la selección de método de renderizado e incluso la capacidad de la portabilidad a VR. También tenemos un control de la cámara, iluminación y efectos como niebla, así como de las propiedades de los materiales, pero sin embargo aunque si se traten las propiedades básicas como rotación, escala y posición, no tenemos forma alguna de acceder a la gestión de físicas ni comportamientos del juego. Lo que limita drásticamente el desarrollo que deberá continuarse en un sistema ajeno a este. Por lo que es más utilizado como un editor para contenido 3D.

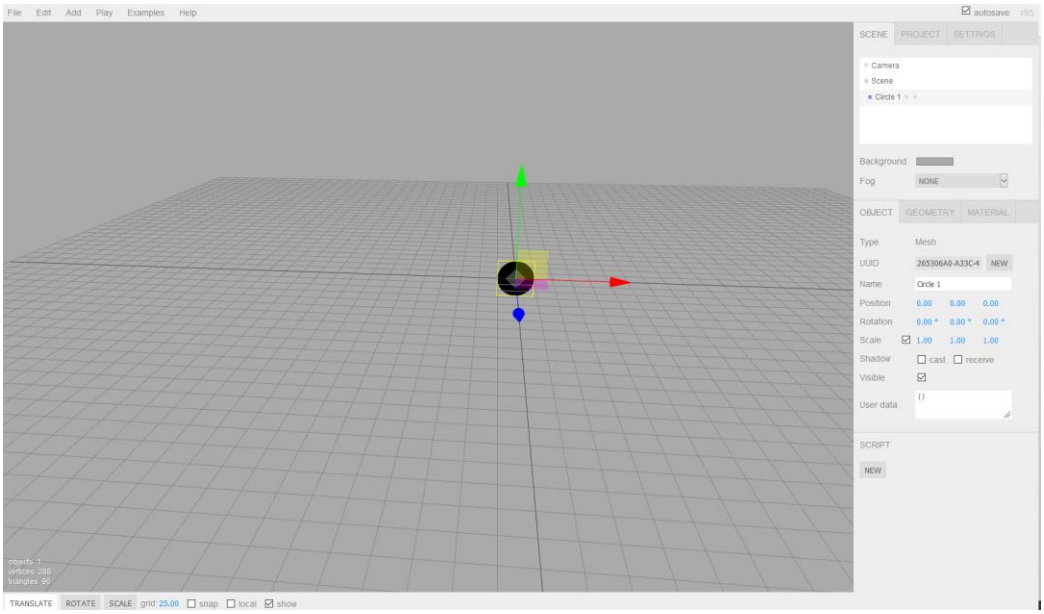


Ilustración 5 Detalle del editor Threejs Editor.

2.1.3 Conclusiones

Estos entornos, han hecho propicio que un gran número de pequeñas empresas o particulares traten de hacerse un hueco en el mercado, creando videojuegos de manera sencilla, sin necesidad de un alto presupuesto, y sin necesidad de un amplio conocimiento informático para llevarlo a cabo. Una mención especial para Gamemaker que a pesar de no ser altamente conocido posee un gran abanico de herramientas capaces de proporcionar un apoyo casi completo al desarrollo, y que tanto se acerca a los entornos profesionales.

2.2 Entornos profesionales

A partir de este punto, nos introducimos en los entornos de desarrollo utilizados generalmente en la industria del videojuego y que por tanto dan un mayor soporte, y funcionalidades como el soporte al juego Online directo, optimizaciones y por lo general una librería más extensa de contenido desarrollado por los propios usuarios.

2.2.1 Unity 5

Unity es uno de los entornos favoritos por la industria del videojuego. Este motor proporciona un alto nivel de compatibilidad con ficheros de todo tipo, admite la inserción de plugins permitiendo un nivel de escalabilidad tan alto como el usuario desee, tanto por su propia implementación como con las 1700 extensiones de su tienda. Permite la optimización de contenido, y una rápida iteración debido a sus múltiples herramientas. Como gestores de físicas dispone de BOX2D con sus propias extensiones, y las físicas de NVIDIA® PhysX® 3.3.

La GUI como tal, se nos presenta dividida de nuevo en secciones, donde destacamos el viewport como medio visual de comunicación con el entorno de juego, y los módulos de control de propiedades y elementos alrededor de este. Podemos observar como esta GUI nos ofrece una mayor cantidad de información y control acerca de nuestro juego, como por ejemplo la creación y control de menús, animaciones y posiciones en tiempo real, detalles de cómputo y coste… Sin olvidarnos de las opciones básicas, que nos aportan un mayor detalle.

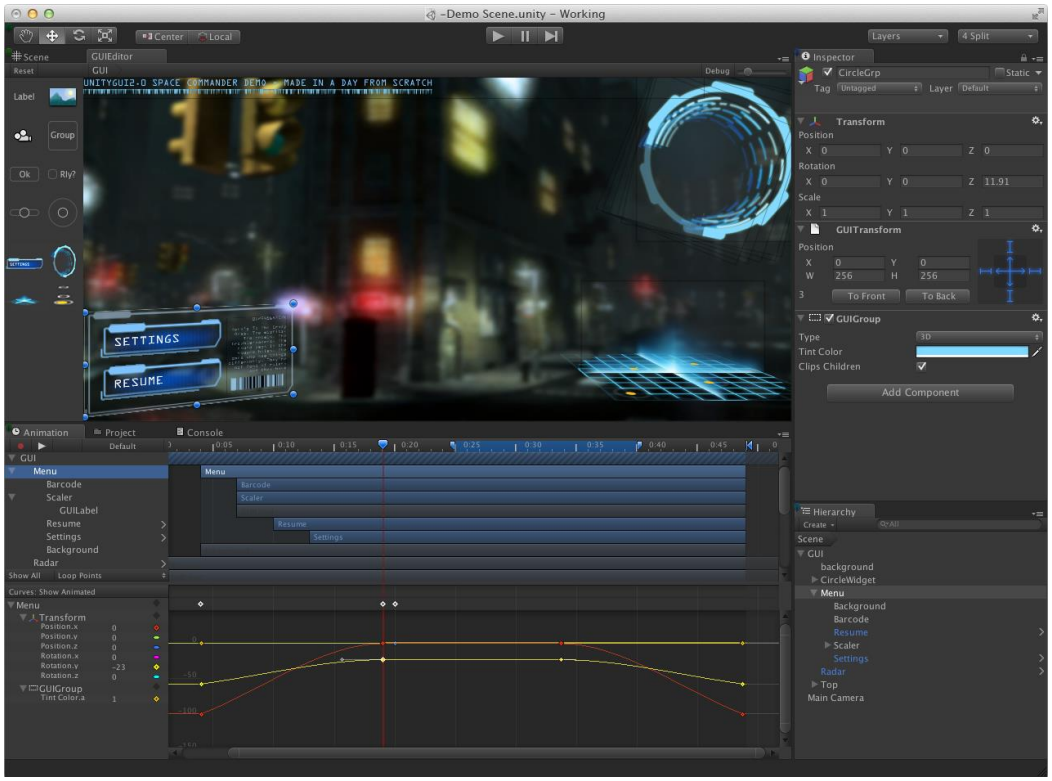


Ilustración 6 Detalle GUI Unity 5.

Además, Unity 5 proporciona por primera vez una herramienta de dibujo que nos permite modelar, texturizar y renderizar nuestros assets en el propio entorno, con un nivel de detalle bastante aceptable para no tratarse una herramienta de modelado. También nos permite ajustar los paquetes de texturas así como su edición.

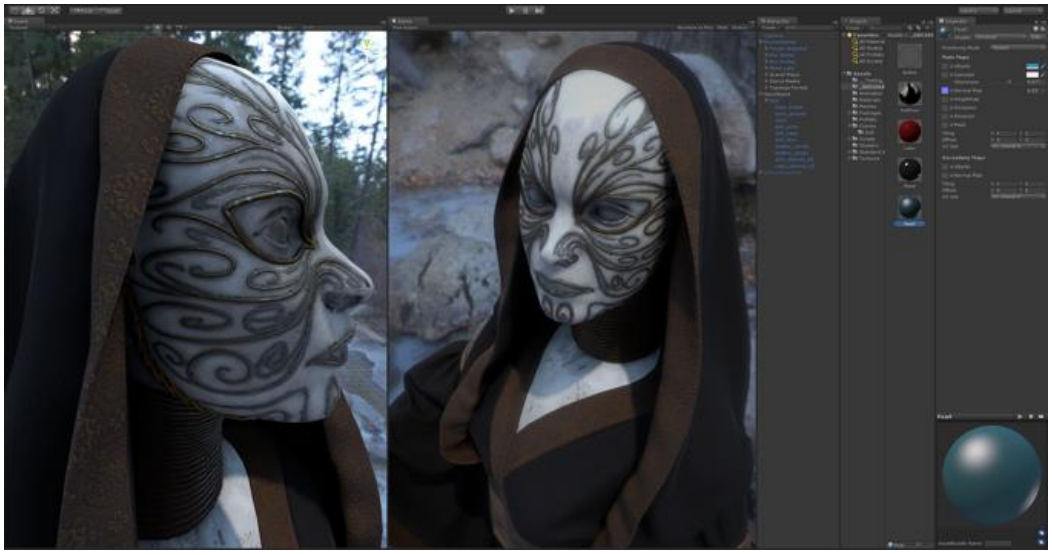


Ilustración 7 Detalle Modelador Unity 5.

Así como un mezclador para realizar las ediciones de sonido en el propio entorno.

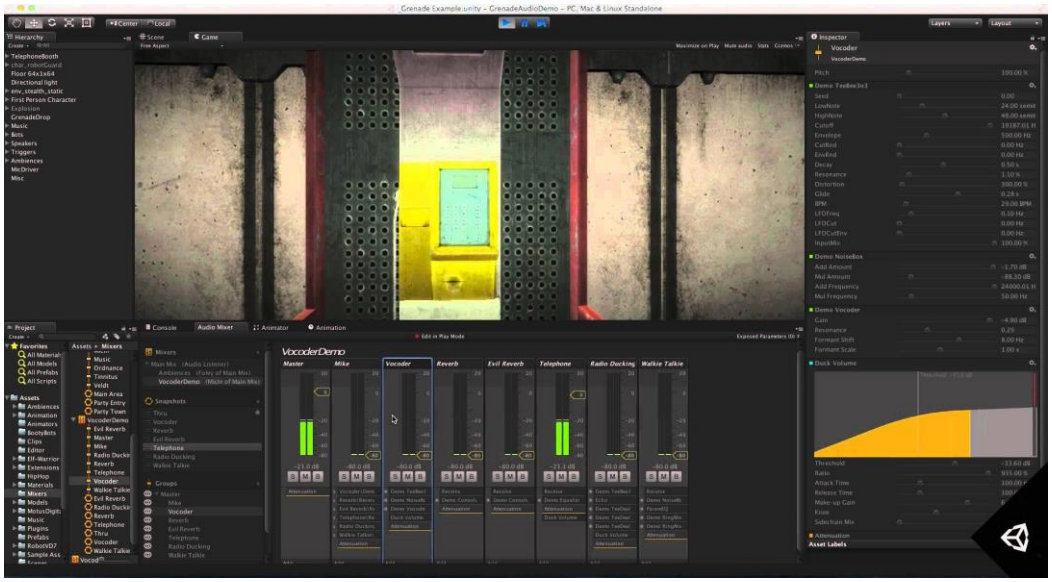


Ilustración 8 Detalle Mezclador Unity 5.

2.2.2 Unreal 4

Otro de los titanes de la industria del videojuego, es al igual que Unity un motor en continuo crecimiento por la comunidad de desarrolladores que aportan nuevos plugins a su tienda, y que consta de un alto grado de compatibilidad tanto con dispositivos, sistema de cámara y ficheros, pero que se encuentra limitado a la programación en C++ al ir junto a Visual Studio como su soporte para la inserción de código.

Lo primero que observamos al tener la GUI delante, es de nuevo un sistema de módulos o ventanas donde mostrar la información y contenido. Esta vez, disponemos de diferentes vistas en el viewport que nos permiten tener una vista real, una cenital, una frontal y una lateral; estas últimas por defecto como mallas, que nos permiten un control más amplio de nuestro entorno. Tras esto, nos fijamos en las secciones donde encontramos el contenido, y las propiedades básicas del elemento seleccionado. En la parte superior por otro lado tenemos todos los ajustes del mundo y accesos rápidos a las herramientas más frecuentes, véase, la gestión de Blueprints y los Matinee, o animaciones.

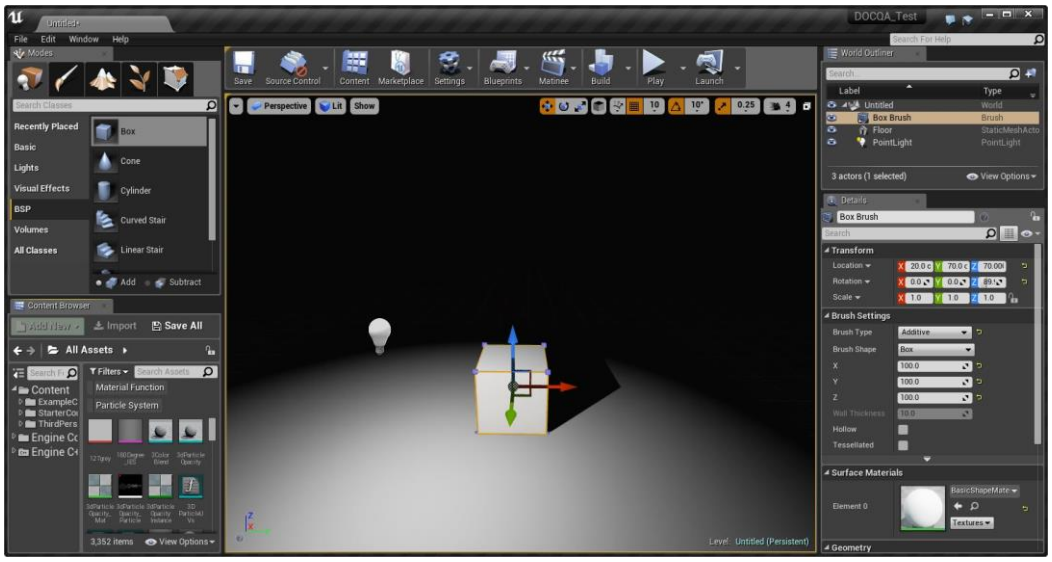


Ilustración 9 Detalle GUI Unreal 4.

En Unreal sin lugar a dudas debemos destacar los Blueprints, una forma que al igual que mencionamos en Gamemaker, facilita el acercamiento de gente sin conocimientos a la informática o simplemente al C++, la posibilidad de trabajar y crear videojuegos. Esta vez, la ventana que se nos presenta será más extensa y detallada que en Gamemaker, y nos permitirá mediante Drag&Drop, la construcción de secuencias de código, e incluso funciones y módulos enteros, donde podremos, aunque no de una forma tan eficiente, realizar casi cualquier cosa necesaria sin necesidad de una sola línea de código C++.

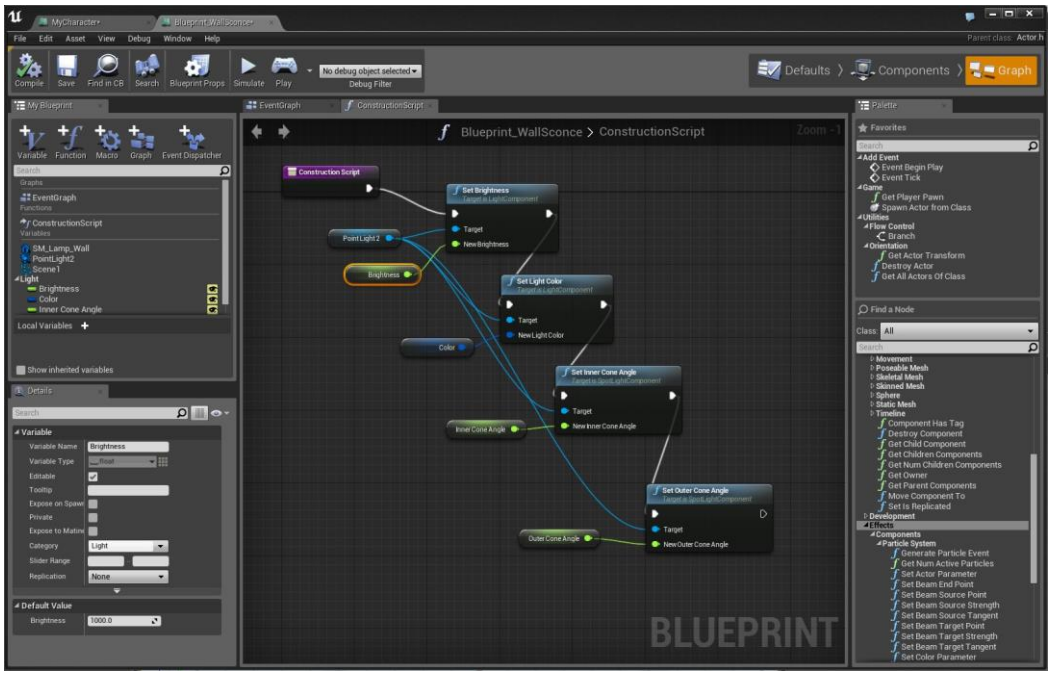


Ilustración 10 Detalle editor Blueprints Unreal 4.

Por otro lado, como se dijo un poco más arriba, el lenguaje propicio para este motor es C++, y viene de la mando de Visual Studio, entorno de sobra conocido por muchos.

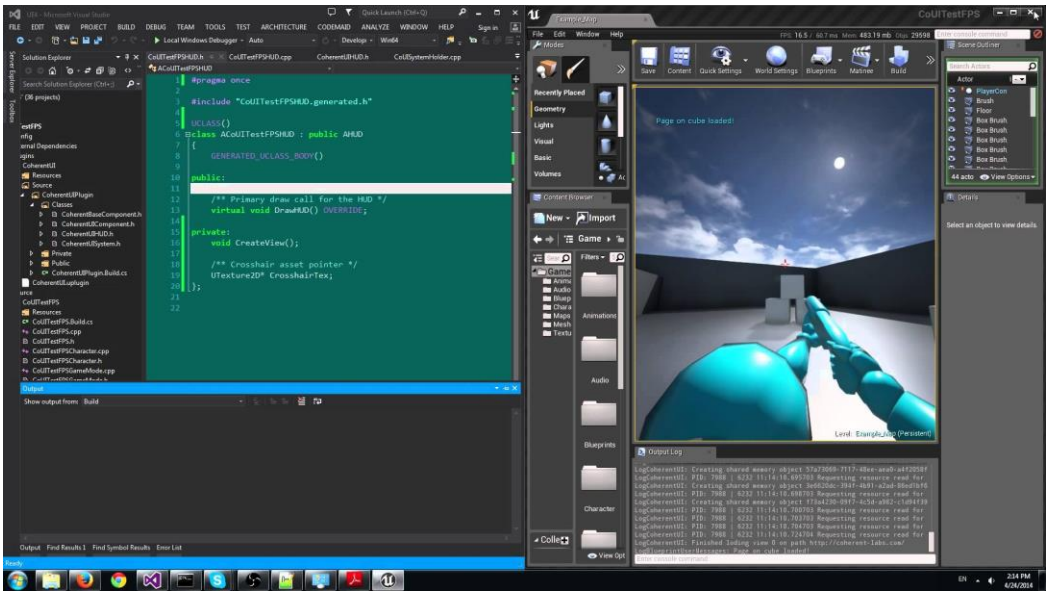


Ilustración 11 Detalle editor código Unreal 4.

Además de esto, Unreal no sería uno de los motores y entornos favoritos si no fuera porque dispone de su propio entorno de modelado, y renderizado donde podemos montar nuestros personajes y elementos, e incluso animarlos mediante CGI mediante el uso de esqueletos.

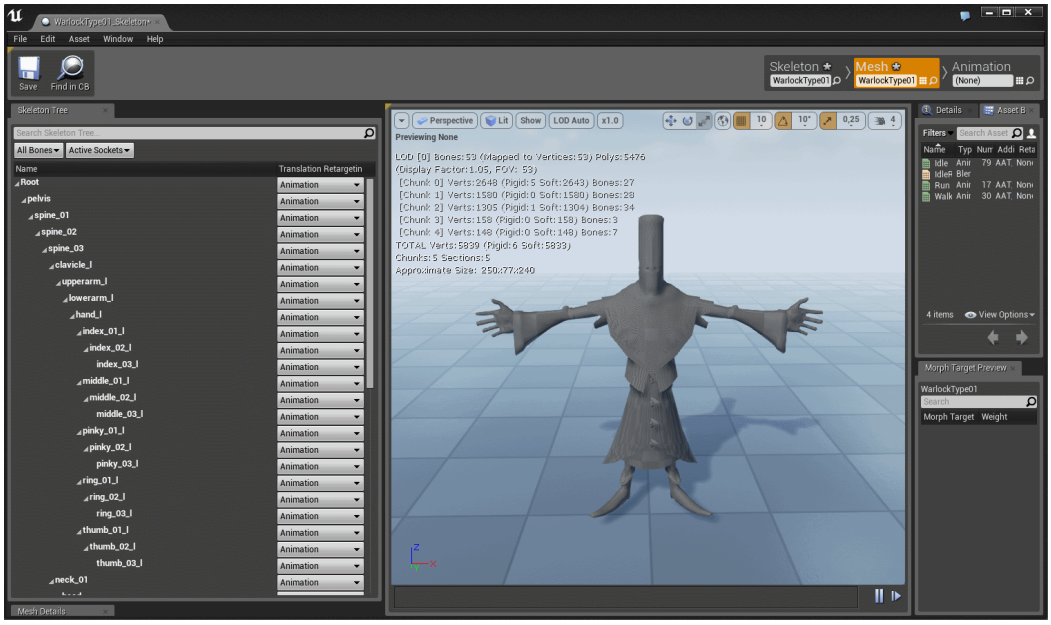


Ilustración 12 Detalle modelador Unreal 4.

Y su herramienta para la creación de animaciones o cinemáticas, Matinee, donde de una forma similar a los Blueprints podremos definir su comportamiento. Y monitorizarlo y editarlo como en una herramienta de edición de vídeo.

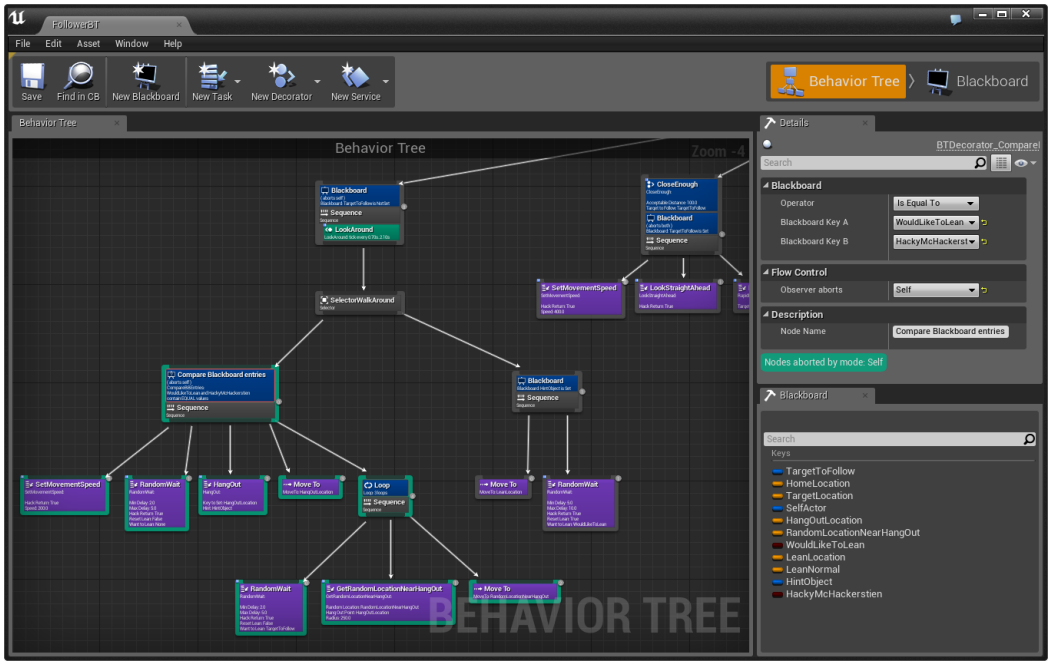


Ilustración 13 Detalle Matinee Unreal 4.

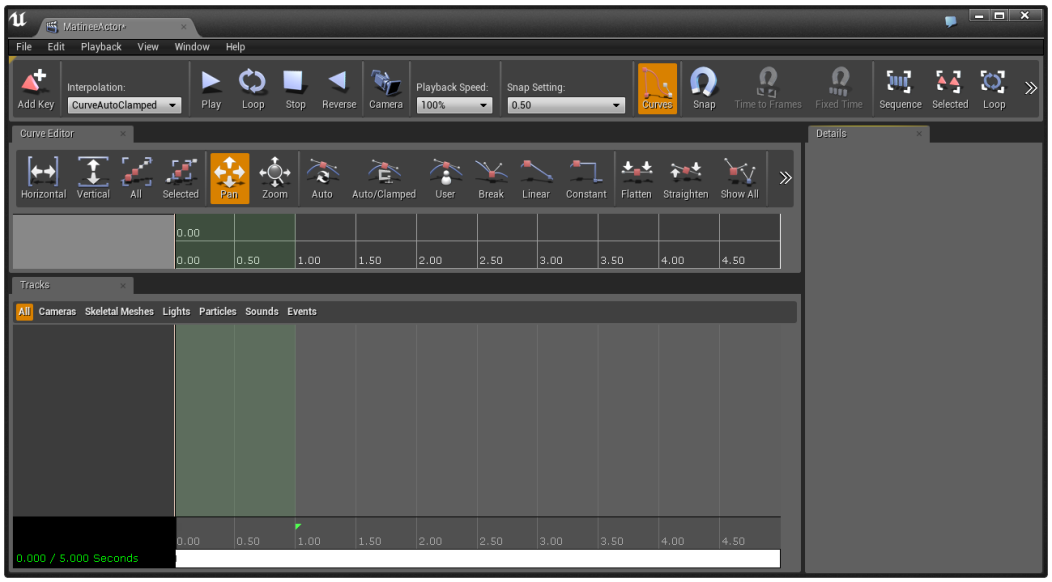


Ilustración 14 Detalle Matinee Unreal 4.

2.2.3 CryEngine 3

Uno de los motores más potentes gráficamente del mercado, y que es utilizado en juegos como Crysis, nos presenta nuevamente una GUI enfocado en un viewport rodeado de diversos módulos de control, así como un sistema de programación similar a los Blueprints de Unreal o Gamemaker.

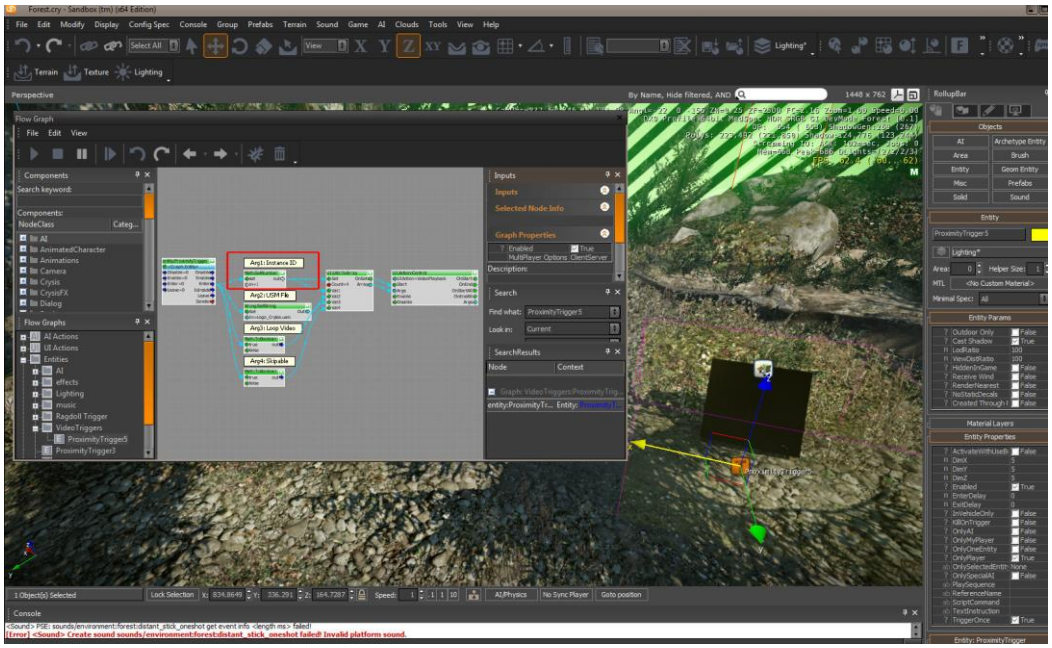


Ilustración 15 Detalle GUI CryEngine3.

2.2.4 Lumber Yard

Por último pero no menos importante se encontraría el Lumber Yard, el motor gráfico que presentó Amazon en el 2016. Podemos observar una vez más que el tipo de GUI vuelve a ser el predominio de un Viewport junto con diversos controles para los elementos, pero esta vez optando por un editor de código independiente del mismo motor.

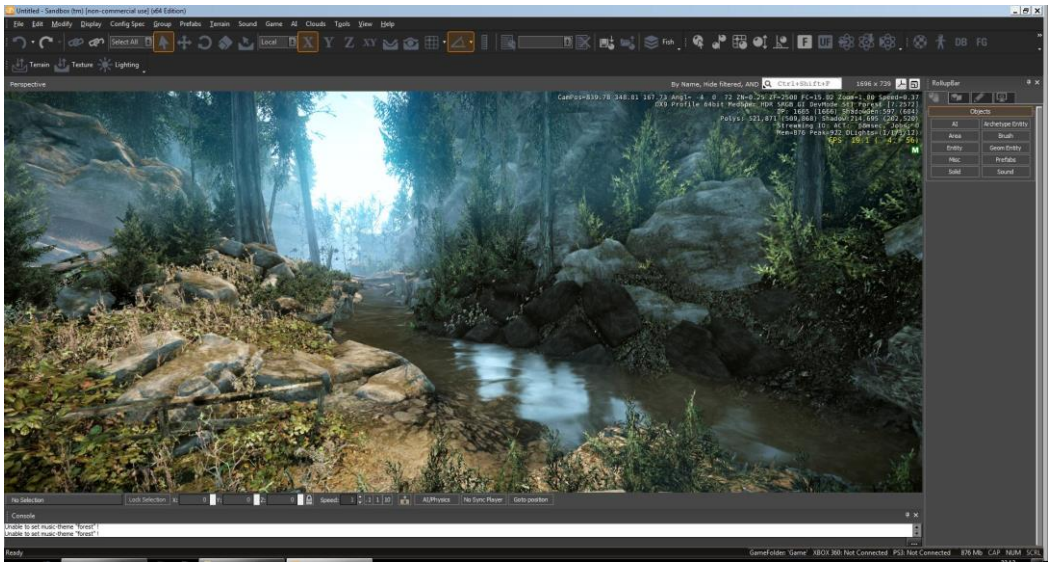
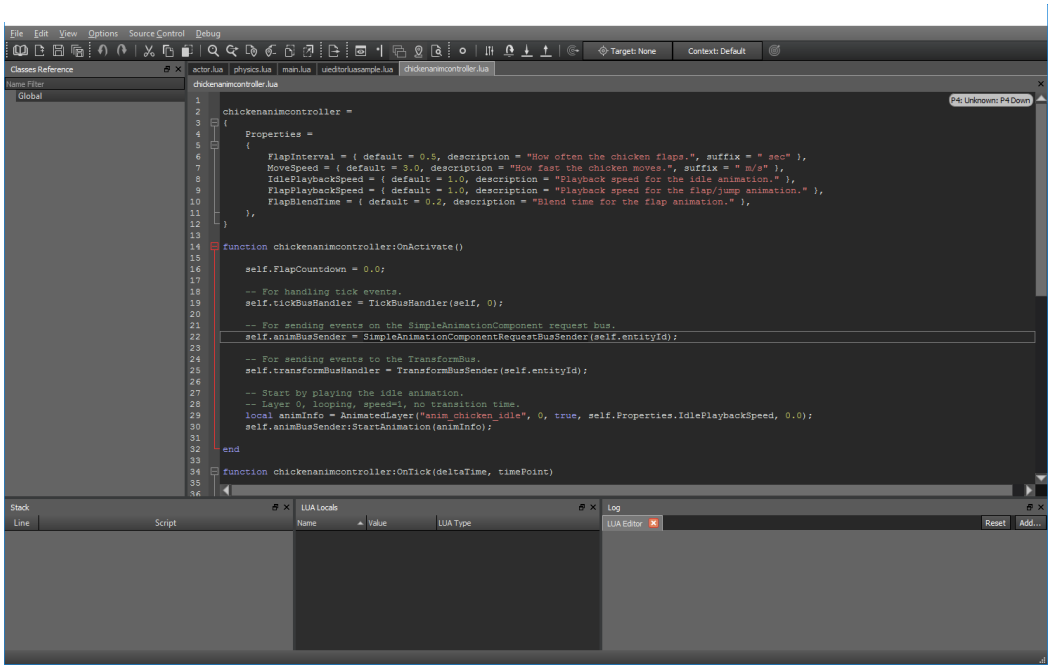


Ilustración 17 Detalle GUI Lumber Yard.



```
chickenanimcontroller =
{
    Properties =
    {
        FlapInterval = { default = 0.5, description = "How often the chicken flaps.", suffix = " sec" },
        MoveSpeed = { default = 3.0, description = "How fast the chicken moves.", suffix = " m/s" },
        IdlePlaybackSpeed = { default = 1.0, description = "Playback speed for the idle animation." },
        FlapPlaybackSpeed = { default = 1.0, description = "Playback speed for the flap/jump animation." },
        FlapBlendTime = { default = 0.2, description = "Blend time for the flap animation." },
    },
}

function chickenanimcontroller:OnActivate()

    self.FlapCountdown = 0.0;

    -- For handling tick events.
    self.tickBusHandler = TickBusHandler(self, 0);

    -- For sending events on the SimpleAnimationComponent request bus.
    self.animBusSender = SimpleAnimationComponentRequestBusSender(self.entityId);

    -- For sending events to the TransformBus.
    self.transformBusHandler = TransformBusSender(self.entityId);

    -- Start by playing the idle animation.
    -- Layer 0, looping, speed=1, no transition time.
    local animInfo = AnimatedLayer("anim_chicken_idle", 0, true, self.Properties.IdlePlaybackSpeed, 0.0);
    self.animBusSender:StartAnimation(animInfo);

end

function chickenanimcontroller:OnTick(deltaTime, timePoint)
```

Ilustración 16 Detalle editor código Lumber Yard.

2.2.5 Conclusiones

Como conclusión podemos sacar que todos los entornos profesionales optan por un modelo similar. Diferenciando cada punto del desarrollo pero integrándolo todo en un mismo entorno de ventanas.

2.3 Nuestro entorno de desarrollo

Como empezamos hablando, la tendencia general a día de hoy es el desarrollo para todo tipo de dispositivo, aquí es donde entra el Framework sobre el cual crearemos nuestro entorno, Phaser, que al basarse en JavaScript, permite al igual que Threejs o Gamemaker, la reproducción del juego en cualquier dispositivo con un navegador y capacidad para conectarse a internet. Desarrollando un entorno para otorgar a este Framework la capacidad para ser utilizado por cualquier usuario tenga o no conocimientos de informática. Partiendo de intentos como Phaser Editor, programa de escritorio de código abierto, basado en el IDE de Eclipse. Pero al ser una aplicación de escritorio queda limitado al Sistema Operativo que utilice el usuario, por lo que optaremos por la realización de una GUI con carácter Online.

Durante este pequeño estudio, hemos podido observar cual es la tendencia en el mercado para el desarrollo de GUIs, optando por la separación de los diferentes sistemas que componen el entorno de desarrollo. Por ello, en este trabajo se presentará una GUI enfocada en el sistema de construcción y diseño del mundo donde se desarrollará el juego con la utilización de técnicas de Drag&Drop para la colocación de los elemtnos y una pequeña aproximación a las propiedades y físicas de estos.

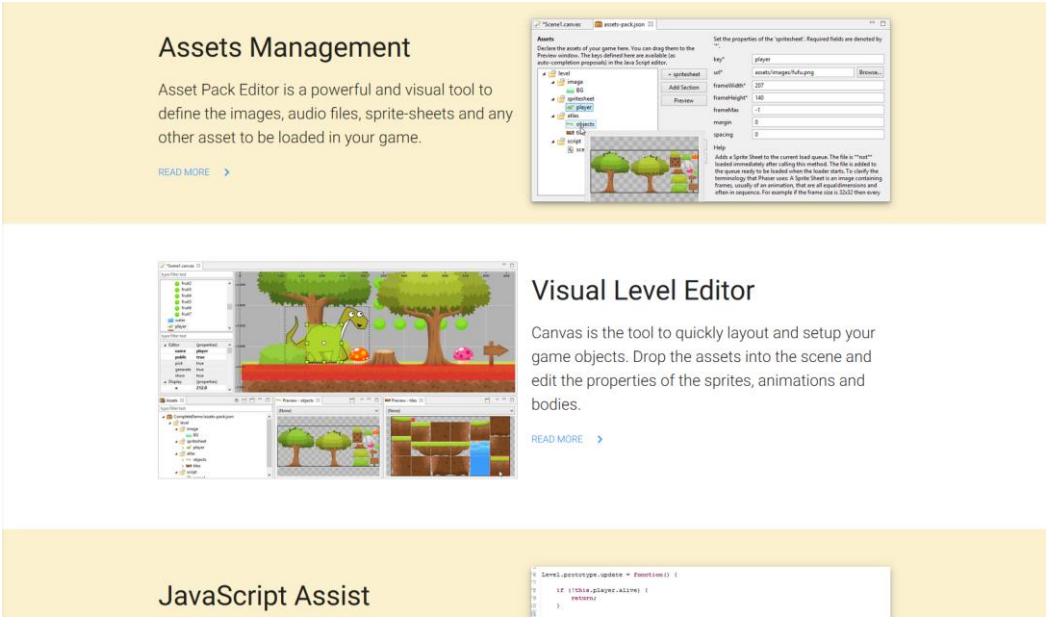


Ilustración 18 Phaser Editor

3 Phaser

3.1 Phaser

Como comentamos al principio de esta memoria, la dirección general en la informática enfocada a videojuegos está siendo la de alcanzar un mayor número de dispositivos y público. Esto primero se está logrando entre otras cosas debido a la presencia de los Smartphone, sin embargo, a día de hoy no es lógico programar para un solo tipo de dispositivo, quitando diferencias de Hardware que limiten esta portabilidad, por lo que una de las soluciones para alcanzar esta multiplataformidad ha sido el uso de navegadores, ya que todo dispositivo moderno cuenta con uno, y no precisa un gasto de recursos elevados. La aparición de HTML5 ha sido la revolución que ha podido llevar a cabo esta unificación de plataformas y explotación de los navegadores debido a las nuevas propiedades que presenta tales como la gestión de vídeos, animaciones, Canvas… Que evitan el uso de medios como Flash, tecnología no soportada por todos los navegadores y dispositivos.

Phaser como tal es un Framework open source de desarrollo para videojuegos en navegadores, basado en JavaScript. Para su funcionamiento utiliza Pixi,js, que al igual que otras herramientas como Bootstrap, confieren a nuestra página una mayor posibilidad de edición y visualización. Pixi.js además es el encargado del renderizado del proyecto utilizando Canvas en 2D y WebGL.

3.1.1 WebGl

WebGL se define a sí mismo como una API de bajo nivel para el renderizado de gráficos 3D en navegadores, basado en OpenGL ES 2.0, y que se apoya en el canvas de HTML5.

Khronos Group, un conscorcio tecnológico altruista son los encargados de su desarrollo y mantenimiento. Cabe destacar que son los responsable de la librería Vulkan, que pretende ser el estándar de librería gráfica de código abierto.

Como muestra de lo que se puede llegar a realizar con WebGL cabe destacar los shaders que destacan por su gran calidad gráfica como se puede apreciar en la Figura 4.

3.2 Funcionamiento Phaser

Para desarrollar en Phaser son necesarios ciertos elementos básicos: El mundo, el viewport, y el motor de renderizado.

- El mundo es un contenedor de información que conformará el nivel, en él se añadirán los assets y elementos que lo "pueblen". Es necesario cargar los elementos en cache previamente en una función de precarga para poder añadir elementos. Además la inserción de estos en el mundo debe realizarse de forma explícita al mundo, o bien a un grupo previamente añadido, para que este se visualice.

- El viewport que definido por el tamaño del canvas del HTML5 determinará qué elementos se mostrarán o no del mundo según la posición de la cámara.

- El motor de renderizado que como su nombre indica será el responsable de pintar los gráficos en el navegador.

Además, hay que destacar que un juego en Phaser está formado de estados, mínimo uno de ellos, que contendrá el mundo. Estos estados contienen a su vez una función indispensable llamada **create**, donde se instanciará el estado propiamente dicho y donde se inicializarán las variables de este.

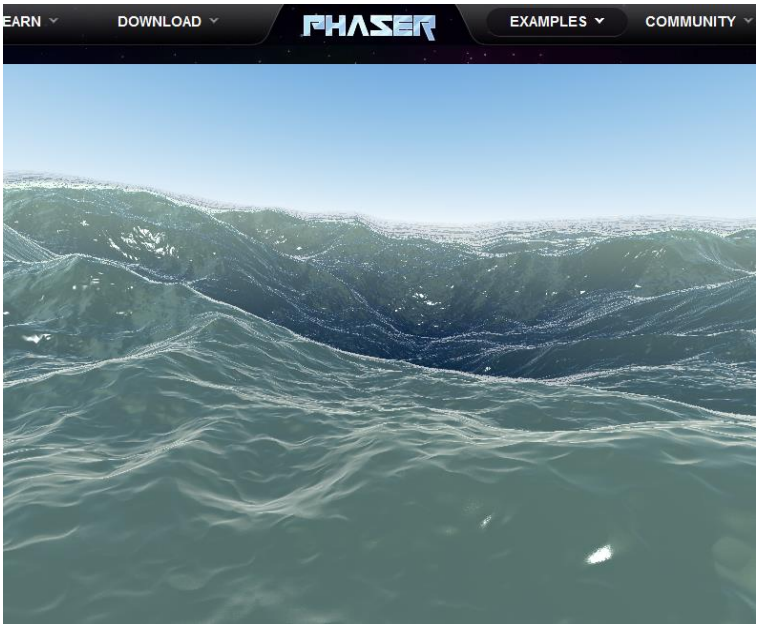


Ilustración 19 Ejemplo Shader en Phaser con WebGl.

4 Análisis y diseño

A lo largo de este apartado se procederá a describir el análisis de los requisitos de la GUI, y el diseño de la implementación de la misma.

4.1 Análisis

Se especifica los requisitos o comportamiento que debe presentar la GUI, concretando las funcionalidades de la forma más concisa y clara posible.
Los tipos de funcionalidad son: Funcional y no funcional.

4.1.1 Requisitos funcionales.

Conjunto de comportamientos a emplear:

Inicialización:

Requisitos funcionales en la creación del entorno del editor.

Tabla 1: Requisitos funcionales de la inicialización de la GUI.

Código	Requisito
INI-01	Creación de la variable Game de Phaser. Este se acoplará a los valores de ancho y alto del div central.
INI-02	Inicialización de la variable World de Game, tomará los valores ancho y alto, y background, por defecto de la GUI. Por defecto en (2000, 2000, #4d4d4d)
INI-03	Creación del Grid, tomará los valores de visibilidad, tamaño y color por defecto de la GUI. Por defecto en (true, 32, #4d4d4d)
INI-04	Colocación de la Camera en el centro del mundo, la posición se cargará por defecto de la GUI. Por defecto en (550, 200,)
INI-05	Creación de la Deadzone que seguirá a la cámara.
INI-06	Carga de los assets predefinidos

Interacción:

Requisitos funcionales referentes a la interacción del usuario.

Tabla 2: Requisitos funcionales de la interacción de la GUI con el usuario.

Código	Requisito
INT-01	El usuario puede subir imágenes para utilizarlas durante el diseño de nivel.
INT-02	El usuario puede editar las capas del nivel, es decir, la posición de los assets en el plano Z.
INT-03	El usuario puede mover la cámara mediante el uso del teclado o ratón para desplazarse por el mundo.
INT-04	El usuario puede seleccionar un asset sobre el que trabajar.
INT-05	El usuario puede desplazar un asset seleccionado mediante Drag&Drop para moverlo por el mundo.
INT-06	El usuario puede ver las propiedades de un asset seleccionado.
INT-07	El usuario puede modificar las propiedades de un asset seleccionado.
INT-08	El usuario puede eliminar un asset del mundo.
INT-09	El usuario puede ver las propiedades del mundo.
INT-10	El usuario puede modificar las propiedades del mundo.
INT-11	El usuario puede ver las propiedades del Grid.
INT-12	El usuario puede modificar las propiedades del Grid.

4.1.2 Requisitos no funcionales.

GUI

Tabla 3: Requisitos no funcionales de la GUI.

Código	Requisito
GUI-01	Usabilidad: Interfaz sencilla, con funciones simples y concisas que faciliten al usuario su aprendizaje y manejo.
GUI-02	Portabilidad: Debido a los lenguajes utilizados en el desarrollo de esta GUI, es portable a cualquier plataforma con conexión a internet.
GUI-03	Escalabilidad: Se pueden añadir nuevas funcionalidad con añadir un nuevo JS en los estados del Juego.

4.2 Diseño

Se procede a realizar una descripción de cómo está diseñada la GUI, y como se divide.

4.2.1 Patrones de diseño.

Los patrones de diseño utilizados en este proyecto han sido: MVC (Modelo Vista Controlador), Composite, y Prototype.

4.2.1.1 Modelo Vista Controlador

Este patrón de arquitectura de Software, diferencia 3 bloques en una aplicación:

- Modelo: Contiene los datos y la lógica
- Vista: La interfaz de usuario
- Controlador: Comunica los dos bloques anteriores y permite el intercambio de información.

Se ha escogido debido a que Además, en este proyecto esta diferencia se encuentra muy marcada por los lenguajes utilizados en ambas partes, véase HTML y CSS para la Vista, y JavaScript para el Modelo de datos, junto a una tercera parte encargada del intercambio de información entre estas.

4.2.1.2 Composite

Este patrón se puede apreciar en el empleo de una variable Group, ya que como indicamos en la sección 3. Phaser, para introducer elementos en el mundo necesitamos un elemento "contenedor" previamente inicializado donde añadir los elementos.

Con este elemento group, conseguimos introducir los assets que el usuario quiera en el mundo.

4.2.1.3 Prototype

Este patron lo emplea Phaser por defecto, ya que en la precarga que mencionamos anteriormente ya creamos un elemento X, en cache, que posteriormente se usará para generar un clon en la posición requerida.

Esta instanciación es el método para inserter elementos en el mundo, siendo todos los elementos en el mundo clones de unos elementos básicos precargados.

4.2.2 Diagrama de clases.

A continuación se presenta en la Ilustración 5 un diagrama de clases, módulos, que conformarán el Modelo de la GUI. Mostrando así una primera aproximación al código, y donde podremos observar las diferentes funcionalidades que posee y que detallaremos más adelante.

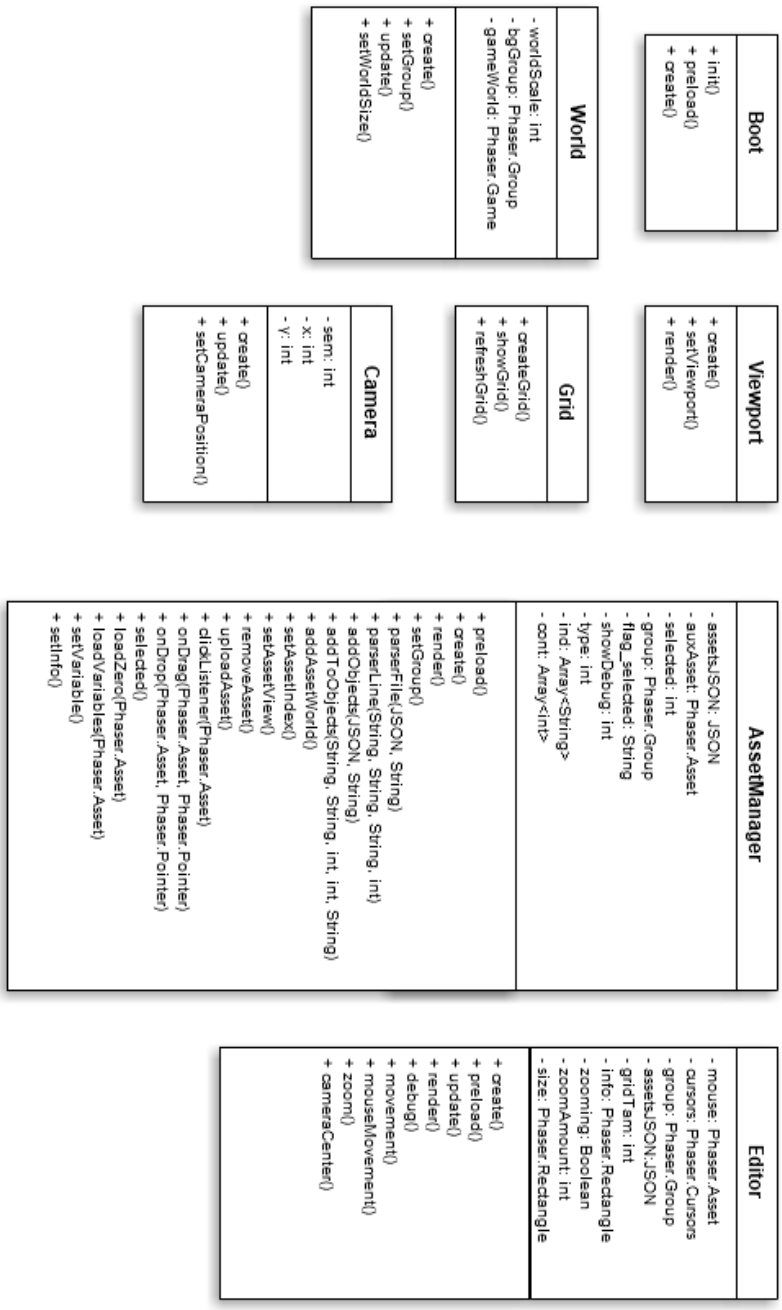


Ilustración 20 Diagrama de Clases

5 Desarrollo de la Vista

Como comentábamos en el apartado anterior, para el desarrollo de la GUI nos hemos basado en el patrón MVC, donde uno de los puntos más importantes es la Vista, ya que esta será la que se le presente al usuario, y deberá ser lo más sencilla e intuitiva posible. Para ello, se ha estudiado el diseño de programas como Photoshop, junto a los citados en el apartado 2 de esta documento, Estado del arte; llegando a la conclusión de realizar un diseño basado en widgets que mostrarán al usuario información de una forma ordenada, sobre el mundo que conformará el nivel, así como de los elementos que lo pueblan.

Finalmente se ha dividido por ello la GUI en 5 bloques: **viewport**, **assetManager**, **objectManager**, **worldSettings** y **assetSettings**.

- **Viewport:** Será el bloque principal, situado en la ubicación central y donde se cargará el nivel de Phaser sobre el que trabajaremos. En él, podremos interactuar mediante Drag&Drop con los elementos introducidos, y visualizarlos para dar forma a nuestro nivel, con la ayuda de un grid, o rejilla, de tamaño variable. Figura 5.

- **AssetManager:** Será el primer widget al uso, estará situado en la esquina superior derecha de la pantalla y en él se mostrarán todos los assets que podremos utilizar en el nivel, así como los que carguemos manualmente. Figura 6.

- **ObjectManager:** Nuestro segundo widget, situado bajo el AssetManager, nos mostrará los elementos introducidos en el nivel, así como la posición que ocupan en el eje Z, capa y funciones de edición básicas como: subir/bajar la posición de la capa, mostrar/ocultar el objeto, y destruirlo. Figura 7.

- **WorldSettings:** El último bloque de la sección derecha de la pantalla, y que mostrará las propiedades básicas del mundo, tales como el tamaño, el *background*, el tamaño de la rejilla… Figura 8.

- **AssetSettings:** Situado bajo el viewport, mostrará las propiedades básicas del elemento seleccionado actualmente, así como las físicas de dicho elemento. Figura 9.

Estos bloques se implementaron mediante HTML y CSS. Realizando una interfaz con un tema oscuro y una tipografía clara para evitar cargar la vista en un prolongado uso. Figura 10.

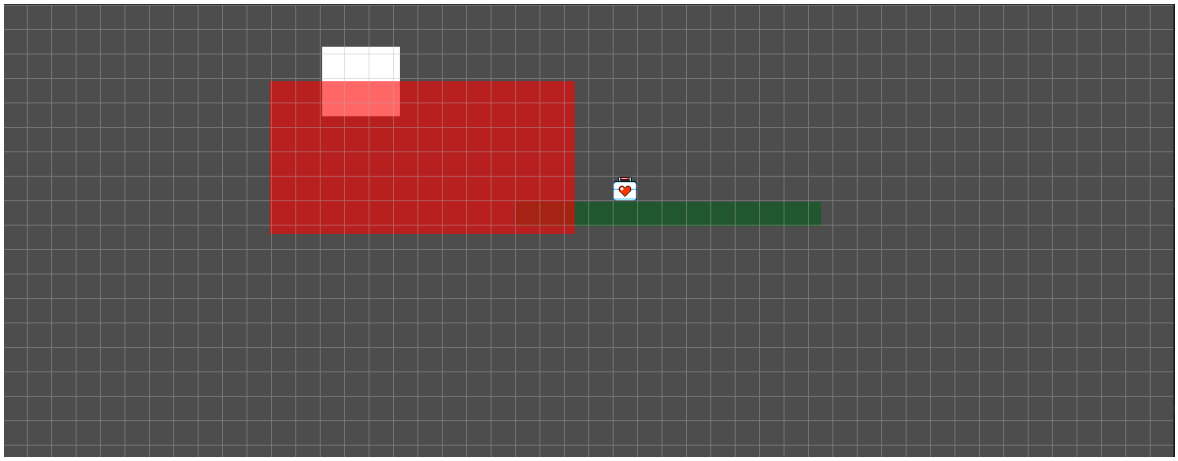


Ilustración 21 Detalle del Viewport.

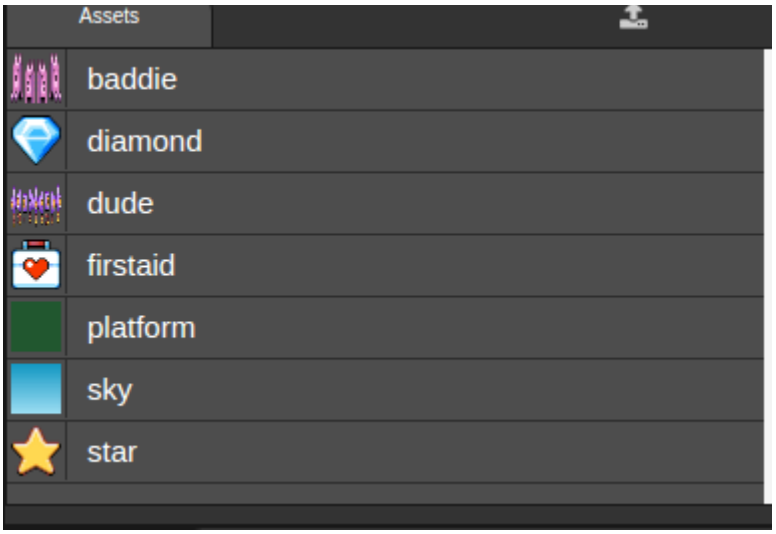


Ilustración 22 Detalle del AssetManager.

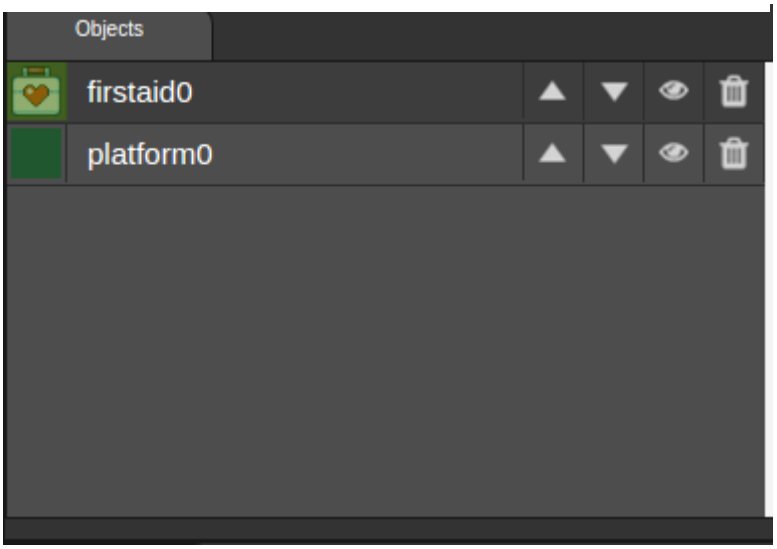


Ilustración 23 Detalle del ObjectManager.

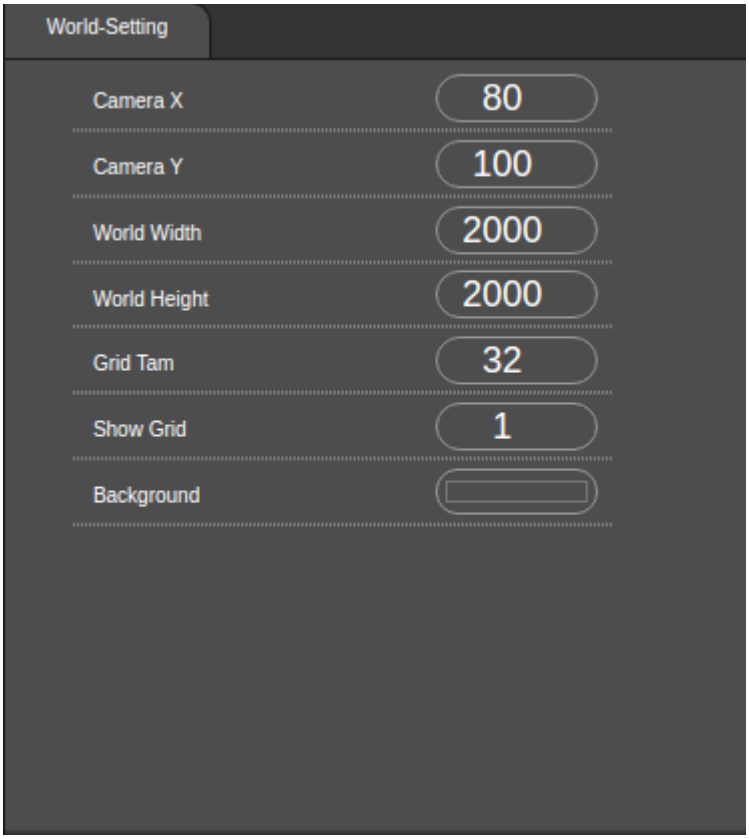


Ilustración 24 Detalle del WorldSetting.

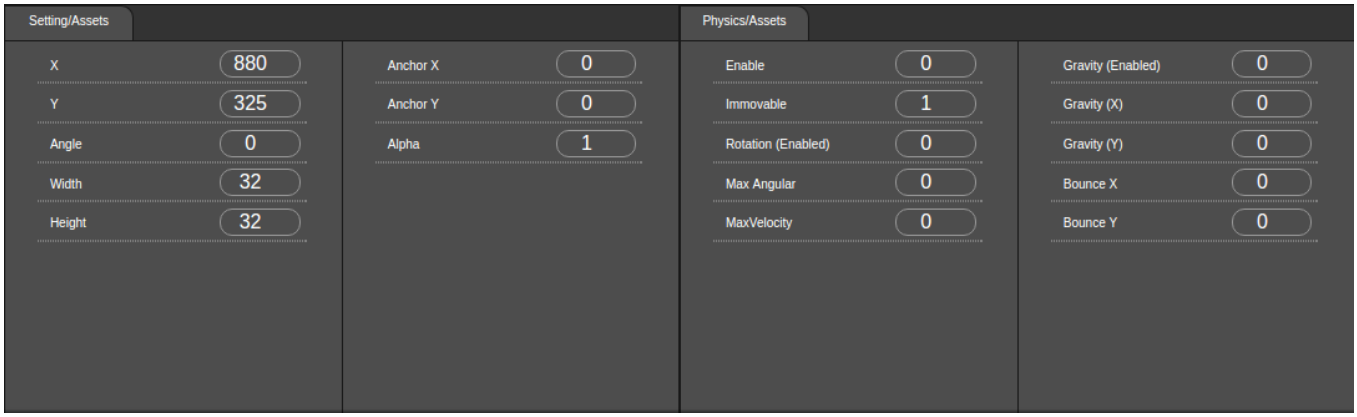


Ilustración 25 Detalle del AssetSetting.

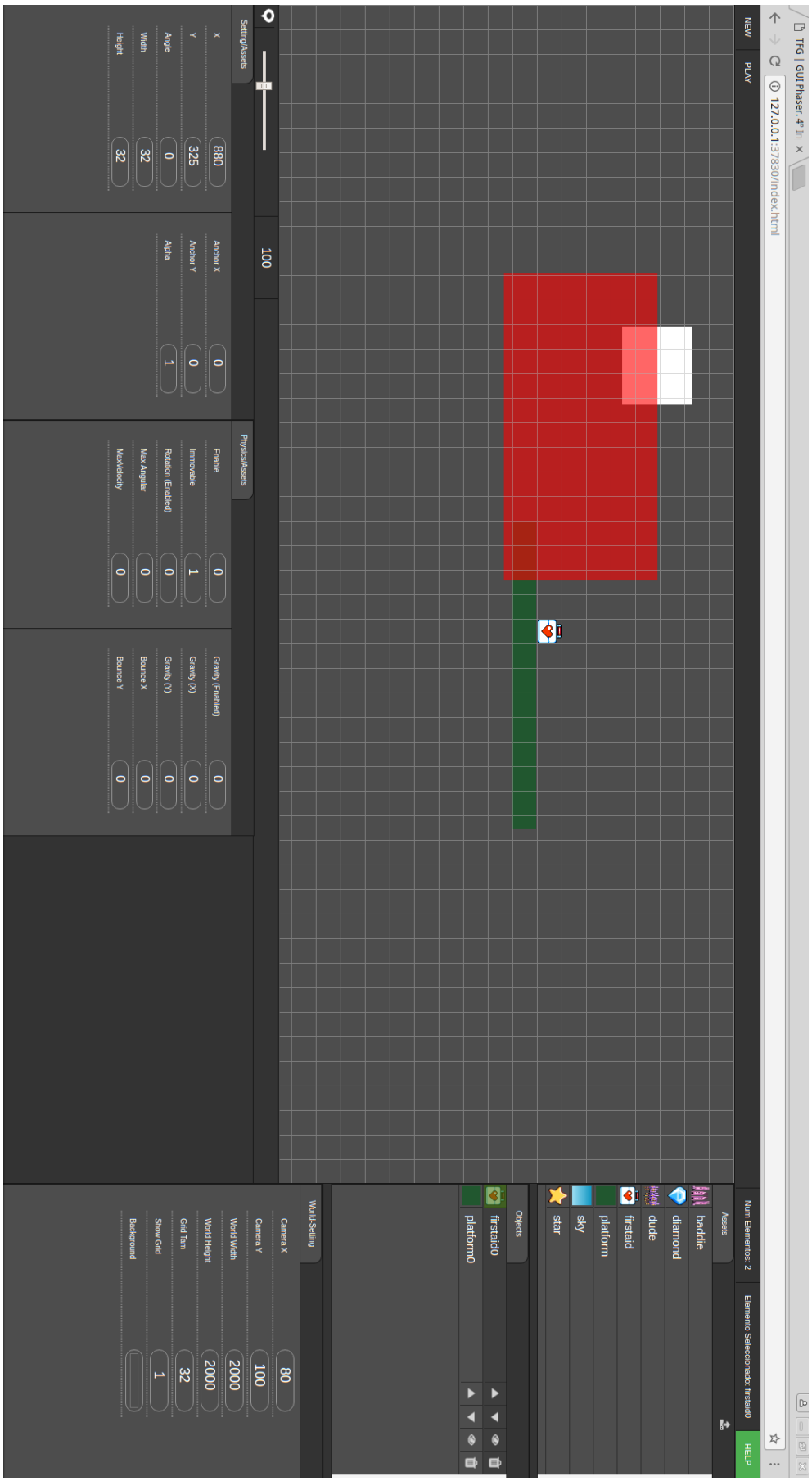


Ilustración 26 Vista de la GUI completa.

6 Desarrollo del Modelo

En esta sección abordaremos el desarrollo del Modelo de datos, así como la resolución de porqué se han tomado ciertas decisiones durante el mismo. Para ello hemos utilizado el lenguaje JavaScript, en el que está basado el Framework para el que trabajamos, y en el que construiremos el entorno de edición, realizando así un juego que genera otros juegos. Además, teniendo en cuenta el requisito de escalabilidad del proyecto así como para facilitar el desarrollo del editor, hemos optado por dividir nuestro código en varios módulos, con el fin de facilitar el desarrollo, legibilidad y la capacidad de añadir futuras mejoras a este. Esto no sería un problema con JavaScript en condiciones normales, debido a que una de las características de este lenguaje, es que siempre que se encuentren referenciados en el HTML no es necesario ningún tipo de referencia interna, sin embargo; al decidir crear un juego que genere otros, como editor, hemos necesitado recurrir a la librería JRequire que nos permitirá modular el código utilizado en Phaser. Donde se han realizado los módulos: **initial.js**, **boot.js**, **world.js**, **camera.js**, **viewport.js**, **grid.js**, **assetManager.js** y **editor.js**. Siendo los módulos principales los dos últimos, ya que estos soportan la mayor parte del proyecto.

El primer paso al utilizar JRequire, será crear un JavaScript que será el referenciado en el HTML, **Initial.js**, donde crearemos la variable del juego recogiendo el ancho y alto del canvas donde se visualizará, cargaremos el resto de módulos, y los añadiremos como estados del juego para finalmente lanzar el primer módulo propiamente dicho, **Boot.js**.

Este primer módulo o estado, contendrá las funciones de **init**, donde estableceremos propiedades básicas del juego como el número de punteros activos, las físicas básicas para el desplazamiento de la cámara, y además bloquearemos las funciones por defecto del click derecho del ratón en navegadores para evitar conflictos con futuros estados. La función **preload**, donde se cargará el asset de la cámara, y se añadirá a cache el JSON donde contenemos los assets que se mostrarán por defecto. Finalmente, la función **create**, llamará al siguiente módulo en la ejecución.

El segundo módulo es **World.js**, en él, tendremos 3 funciones distintas que se encargarán de: modificar el tamaño del mundo del juego cuando se cambien los valores mostrados en la Vista, establecer el color del *background* según los datos introducidos, y la de llamar finalmente al próximo módulo **Camera.js**, que a su vez llamará al viewport, colocará la cámara en la posición marcada por los valores de la Vista, y actualizará los mismos cuando la cámara se mueva durante el uso del editor. El cuarto y último de los módulos básicos se trata del **Viewport.js**, que colocará la deadzone, es decir, el espacio donde la cámara no se moverá aunque se desplace el ratón y que facilita el desarrollo al aumentar la comodidad del desplazamiento ligero de la cámara, y que por último inicia el **AssetManager.js**, que a su vez iniciará el **Editor.js**.

6.1 AssetManager

Este módulo será el primero con una funcionalidad extensa, y de la que hablaremos en detalle:

- **Preload:** Aquí se cargarán los assets del JSON introducidos en caché durante el módulo **Boot.js**, parseandolos y cargándolos en el catálogo de assets que instancia el patrón *Prototype* comentado anteriormente.

- **Create:** Llamará al estado de **Editor.js**.

- **SetGroup**: En esta función se inicializará una variable Group que será un clon de la que crearemos en **Editor.js**, y que instanciará el patrón *Composyte*.

- **ParserFile** y **ParserLine**: Estas dos funciones serán las encargadas de recorrer el catálogo de assets, inicializar un array de índices para cada uno, e inicializar otro para llevar la cuenta del número de instancias de cada uno en el mundo. Tras esto, añadiremos por cada asset una nueva fila con su imagen y nombre; en el widget con el mismo nombre que este módulo, es decir, AssetManager, consiguiendo así mostrar al usuario todo el catálogo de assets y que este pueda seleccionar cual introducir al mundo.

- **AddObject** y **AddToObject**: Estas funciones son análogas a las mencionadas anteriormente, con la diferencia de que esta vez en lugar de leer el catálogo de assets, leeremos el grupo donde se encuentran todos los elementos añadidos al mundo, y los mostraremos de una forma parecida al widget ObjectManager. Sin embargo, esta vez añadiremos los controles de capa que se encargarán de modificar el eje Z de los elementos, un control para la visibilidad del elemento, y un último botón para poder eliminar el elemento del mundo.

- **AddAssetWorld**: Esta función comprobará el elemento seleccionado del widget AssetManager, y lo añadirá al mundo insertándolo en el grupo. Junto a esto, se modificarán los valores de los arrays de índice y de contador, que serán utilizados para nombrar al elemento dentro del grupo, y así poder posteriormente seleccionarlo y reconocerlo.

  La inserción se realizará en la esquina superior izquierda del viewport, es decir, la posición X e Y de la cámara en ese momento, ajustándose al tamaño de la rejilla, así como habilitando las físicas que permiten interactuar con el objeto mediante inputs, otorgando con esto la solución al requisito **INT-05** de la tabla 2 del documento, permitir al usuario desplazar los elementos mediante Drag&Drop.

  Una vez insertado, se llamará a las funciones de AddObject que serán las encargadas de cargarlo en el widget.

- **SetAssetIndex**: Esta función es la que se encargará de establecer la posición Z de la capa. Para ello se cargará el índice del elemento dentro del grupo, y se le aumentará o disminuirá el valor del eje consiguiendo así desplazar los elementos y ubicarlos en la posición correspondiente para su visualización en dicho plano.

- **SetAssetView**: Tratará el estado de visibilidad del elemento seleccionado, ocultándolo o haciéndolo visible, con el fin de facilitar el desarrollo ocultando elementos no útiles, o bien, rebajar el coste de procesamiento al no tener que renderizar dicho elemento.

- **RemoveAsset**: Eliminará del grupo, el elemento seleccionado, y modificando el valor de los arrays de dicho tipo de elemento.

- **UploadAsset**: Permitirá al usuario subir imágenes para utilizarlas durante el desarrollo del nivel. Para ello, se añadirán los valores del nombre y *source* del elemento al catálogo de assets, y posteriormente se cargará en caché para poder ser utilizado. Como en las funciones de parseo que añaden elementos al AssetManager, inicializaremos el contador e índice del elemento.

- **OnDrag** y **OnDrop**: Serán los clicklistener de los eventos de Drag&Drop. Para no haber conflictos se ha creado una variable de semáforo que utilizaremos como medida preventiva para evitar el solapamiento de elementos, y carga de datos.

- **Selected**: Modificará el valor de la variable de semáforo para cargar los datos del elemento seleccionado, permitir el Drag&Drop, y además marcar en una variable el elemento seleccionado para poder utilizarse posteriormente.

- **LoadZero**: Cargará en el AssetSetting los valores por defecto de las propiedades, es decir, cargará un valor nulo de 0 en todos los campos.

- **LoadVariables**: Tras verificar el semáforo, se cargará en todos los campos del AssetSetting, los valores del asset seleccionado y marcado según la función Selected.

- **SetVariable**: Se encargará de sincronizar los valores del modelo según marque el Controlador.

- **SetInfo**: Mostrará cual es el elemento seleccionado y el número de elementos introducidos en el mundo para así tener un mayor control sobre el mundo.

6.2 Editor

Este último módulo será el que se encargue de controlar al resto de módulos mencionados, así como contener todos los elementos y lógica del juego editor. Por ello, como en el apartado anterior, hablaremos entrando en detalle acerca de sus funciones:

- **Create**: Como ya hemos mencionado anteriormente, esta es la función primordial para cualquier estado del juego. En este caso, será donde inicialicemos el asset del ratón, y estableceremos que sea el objetivo de la cámara, crearemos la variable de input para tener constancia de las entradas que sufra por teclado y ratón el entorno, e inicializaremos la variable Group que vimos en el **AssetManager**, donde la copiaremos, y por último setearemos el viewport para cargar el deadzone.

- **Update**: Esta función se ejecutará continuamente durante la ejecución del juego, y será la encargada de sincronizar todos los estados, así como permitir el movimiento de la cámara y el zoom de la misma.

- **Render**: Será la que llame a los renders correspondientes de los demás estados ya que solo puede haber uno activo en primer plano en cada momento, es decir, es necesario unificarlos en un único render.

- **Debug**: Esta función ha sido útil a la hora de debuggear el proyecto ya que mostraba en pantalla el estado del mundo, su tamaño y otros datos de relevancia en el desarrollo.

- **Movement**: Es una de las funciones encargadas del manejo de la posición de la cámara. En esta, comprobaremos si ha sido pulsada alguna de las teclas de dirección, y ajustándose al tamaño del Grid actual, se desplazará el asset del ratón, y con este la cámara.

- **MouseMovement**: La segunda función de movimiento de la cámara. Esta función será la encargada de que, al igual que en otros editores gráficos, si se realiza un drag con el ratón manteniendo pulsado el botón derecho se desplace la cámara. Este efecto se consigue sumando a la posición actual de la cámara, la posición de origen del drag con la posición actual del puntero, cambiando la posición de origen con un clon de la posición actual del ratón.

- **Zoom**: Esta función como su nombre indica, es la que se encargará de realizar un zoom sobre el entorno de desarrollo, pudiendo así comprobar el estado del mundo sin necesidad de mover la cámara. Para llevarse a cabo se precisan de variables de control, distancia y límites, ya que decidimos establecerlos en la mitad de la distancia o bien, el doble de la misma, es decir, un 50% hasta un 200%. Con esto, se comprueba si una de las teclas de control ha sido pulsada, en este caso Q y A para zoomin y zoomout respectivamente, y posteriormente se calcula el incremento del worldscaling, involucrado en el zoom y que por defecto estará a 1, manteniéndolo dentro de los límites marcados. Una vez calculado este incremento se setean las propiedades del mundo, escala y límites para ajustarlo a la nueva escala, consiguiendo así un efecto de zoom.

- **CameraCenter**: La última función del módulo y que se encarga de colocar la cámara en la posición original.

6.3 Otros elementos del editor

Otro de los módulos que conforman el editor, será el **Grid.js**¸ que deja de ser un estado del juego, razón por la que no hemos querido no introducirlo en el apartado anterior, pero sí un módulo importante ya que es el encargado de componer el gridLayaout donde trabajaremos en el Viewport, y que hemos nombrado con anterioridad. Este módulo es llamado por el **Editor.js**, y sus dos funciones principales serán:

- ShowGrid: Función encargada de modificar la visibilidad del Grid, según el parámetro que se introduzca en la Vista.

- RefreshGrid: Esta será la función principal de este módulo, y será la encargada de modificar las características del Grid según los valores que tome. Para ello, modificará las propiedades del CSS propio del Grid, para formar una repetición de imágenes con las que formaremos las casillas, invirtiendo el color del background para que siempre sea visible, y que destaque el Grid, y por último ajustado el tamaño de las casillas.

7 Integración, pruebas y resultados

En esta sección trataremos lo referente a las pruebas realizadas para corroborar el correcto funcionamiento de nuestra GUI. Realizando para ello, diversas acciones en el Editor tanto a nivel de código como de la GUI propiamente dicha, usando tanto la consola del navegador, cuyo *log* ha resultado sumamente útil durante el desarrollo y las pruebas del mismo, como las propias funciones de Debug de Phaser que nos *printean* datos del juego en la interfaz.

Para poder afirmar que nuestra GUI es correcta y operativa se han probado que se cumplan las condiciones de los requisitos mencionados en apartados anteriores, realizando una batería de pruebas donde se modificaban mediante código los valores de los elementos en el mundo, observando sus consecuencias en el editor visual; es decir, desplazamiento, rotaciones, niveles de profundidad… Y viceversa, realizando modificaciones a nivel de usuario con los diferentes medios que nos permite la GUI, y observando mediante el *log*, los cambios que aparecen en nuestras variables según nuestras acciones.

A la hora de probar el apartado de físicas teníamos que dirigirnos a un nivel en acción, es decir, ejecutándose, ya que las físicas no podían aplicarse en caliente en el mundo que conforma el Editor ya que las colisiones complican el tratamiento y posicionamiento de los elementos. Sin embargo, el tipo de prueba nuevamente ha sido recogida de datos mediante el *log* y el Debug de Phaser por pantalla, comprobando que el movimiento y colisiones de los assets según las físicas establecidas para él sean concorde a los resultados esperados.

8 Conclusiones y trabajo futuro

8.1 Conclusiones

Para analizar y estudiar correctamente las conclusiones a las que hemos llegado tras el desarrollo de este trabajo, hemos decidido dividir este apartado en las ventajas y limitaciones del proyecto, y el trabajo futuro que se podría llevar acabo para convertir este proyecto en un editor completamente funcional.

8.1.1 Ventajas.

La principal ventaja que aporta este proyecto es que permite un desarrollo de juegos sencillo, rápido y para el que no es necesario un conocimiento elevado de informática para usarse. Consiguiendo con esto que haya más gente que pueda comenzar a desarrollar videojuegos, e incluso, atraer a dicha gente a buscar un conocimiento mayor de la informática para poder avanzar más allá tras haber podido sumergirse una manera superficial en lo que podríamos considerar el desarrollo de una proyecto informático.

Además, al tratarse de un editor web, y no un editor de escritorio como el Phaser Editor que ya mencionamos con anterioridad, nos permite trabajar desde cualquier lugar con una conexión a internet, y sin la necesidad de un hardware potente como en otros entornos.

8.1.2 Limitaciones.

Por un lado podemos observar que hay limitaciones en el sentido de las físicas del editor, ya que a diferencia de las propiedades como la posición, tamaño, o rotación de un elemento, las físicas son interacciones con el entorno y/u otros elementos. Por ello, aunque en el momento de la edición podemos administrar ciertas propiedades de las físicas de un elemento, estas sin embargo, no pueden verse hasta el momento en que se ejecutase el juego editado.

Por otro parte, y en relación a las interacciones, también se encuentra una enorme limitación a la hora de no poder estimar a priori que tipo de interacciones puede necesitar el usuario entre varios elementos, como por ejemplo, la recogida de un elemento para guardarse en el inventario o simplemente que una puerta te lleve a otra pantalla. Este tipo de interacciones que dan vida al juego quedan a manos del propio desarrollador decidirlas y por ende añadirlas.

8.1.3 Trabajo futuro.

Como comenzábamos diciendo en las conclusiones, para ser un editor completamente funcional serían necesario poder resolver estas limitaciones, si no totalmente, al menos en la mayor medida posible, tomando como ejemplos editores de juegos como Unreal o Unity.

8.1.3.1 Físicas

Con el fin de tratar las físicas correctamente en el momento de la edición podríamos realizar otro widget, o ventana, donde pudiéramos comprobar sin necesidad de ejecutar el juego, las propiedades el asset seleccionado. Así por ejemplo podríamos comprobar velocidades de movimiento, rebote y gravedad respecto al mundo sin tener que ejecutar todo el juego, facilitando así su prueba, ya que quizá el elemento que queramos comprobar no se encuentre al principio del mismo o se necesite realizar alguna acción para activarlo.

8.1.3.2 Interacción

Para el tema de interacciones podrían realizarse dos implementaciones distintas, una de ellas sería proporcionar interacciones básicas entre objetos como destrucción de los mismos, siguiendo el ejemplo de editores como Unreal, o Blender, que permiten para un objeto añadirle otro con el que va a interactuar para conseguir así una primera aproximación. Por otro lado, otra solución sería de nuevo otro widget, pero que permitiera la edición de código en tiempo real, consiguiendo así que el programador pueda definir el comportamiento de las mismas de una manera más directa y eficiente.

8.1.3.3 Cinemáticas y animaciones

Otro punto que considero que podría llevarse a cabo como mejora, sería la implementación de un sistema de cinemáticas y animaciones que permitan al usuario no solo comprobar que una animación se va a comportar de la forma esperada, sino además poder crear vídeos para entre escenas o tránsitos.

8.1.3.4 Menús

Por último, otro de los puntos que propondría como mejora del editor, sería permitir modular el juego en niveles, y entre estos niveles poder crear niveles de menús. Para poder así desarrollar completamente un juego funcional en lugar de niveles sueltos que posteriormente deberán juntarse.

Referencias

Recursos web

[1] – Phaser, Visitado 22 de Abril de 2017
http://phaser.io/

[2] – W3Schools, Visitado 13 de Marzo de 2017
https://www.w3schools.com/

[3] – StackOverflow, Visitado 18 de Marzo de 2017
https://es.stackoverflow.com/

[4] – RequireJs, Visitado 22 de Abril de 2017
http://requirejs.org/

[5] – Three.js Editor, Visitado 22 de Junio de 2017
https://threejs.org/editor/

[6] – Phaser Editor, Visitado 22 de Junio de 2017
http://phasereditor.boniatillo.com/

[7] – JQuery, Visitado 22 de Abril de 2017
https://jquery.com/

[8] – Shovel Knight, Visitado 22 de Abril de 2017
http://yachtclubgames.com/shovel-knight/

[9] – Minecraft, Visitado 22 de Abril de 2017
https://minecraft.net/es-es/

[10] – Mario Bros, Visitado 22 de Abril de 2017
https://www.nintendo.es/Juegos/NES/Super-Mario-Bros--803853.html

[11] – Mighty No.9, Visitado 22 de Abril de 2017
http://www.mightyno9.com/

[12] – Gamemaker, Visitado 22 de Junio de 2017
https://www.yoyogames.com/gamemaker

[13] – Unity 5, Visitado 22 de Junio de 2017
https://unity3d.com/es/unity

[14] – Unreal 4, Visitado 22 de Junio de 2017
https://www.unrealengine.com/en-US/blog

[15] – CryEngine 3, Visitado 22 de Junio de 2017
https://www.cryengine.com/

[16] – Lumber Yard, Visitado 22 de Junio de 2017
https://aws.amazon.com/es/lumberyard/

[17] – Pixi.js, Visitado 22 de Junio de 2017
http://www.pixijs.com/

[18] – Canvas, Visitado 09 de Mayo de 2017
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D

[19] – WebGl, Visitado 09 de Mayo de 2017
https://www.khronos.org/webgl/

[20] – OpenGl Es 2.0, Visitado 09 de Mayo de 2017
https://www.khronos.org/opengl/

[21] – Khronos Group, Visitado 09 de Mayo de 2017
https://www.khronos.org/

[22] – Vulkan, Visitado 09 de Mayo de 2017
https://www.khronos.org/vulkan/

[23] – Shaders, Visitado 09 de Mayo de 2017
https://phaser.io/examples/v2/filters/seascape

Glosario

GUI	Graphic User Interface. Es un programa informático que actue como interfaz con el usuario mediante imágenes u otros elementos gráficos.
IDE	Integrated Development Environment. Aplicación informática encargada de proporcionar servicios con el fin de facilitar al desarrollador o programador el desarrollo de software.
Framework	Entorno de trabajo con asistencia definida mediante el uso de módulos que se usan para el desarrollo u organización del software.
Canvas	Elemento HTML usado para la renderización. La traducción literal sería lienzo.
Grid	Modelo de organización en rejilla que permite el posicionamiento de elementos de forma ordenada.
Asset	Denominación a los elementos gráficos partícipes en la creación de un juego.

Anexos

A Código destacable

```
/**
*       ParserFile
*       Esta función es la encargada de leer el catálogo de assets del JSON suministrado
*       catalog: Fichero JSON con los assets del editor
*       HTML: Identificador del div HTML
**/

parserFile: function(catalog, HTML) {
      document.getElementById(HTML).innerHTML = "";
      for (var i = 0; i < catalog.length; i++) {
            this.load.image(catalog[i].name, catalog[i].path);
            this.parserLine(catalog[i].name, catalog[i].path, HTML, i);
            ind[assetsJSON.catalog[i].name] = i;
            cont[catalog[i].name] = 0;
      }
},


/**
*       ParserLine
*       Esta función es la encargada de interpretar cada asset y cargarlo en la vista
*       name: Nombre del asset
*       path: Ruta de la ubicación del asset
*       HTML: Identificador del div HTML
*       i: contador
**/


parserLine: function(name, path, HTML, i) {
      document.getElementById(HTML).innerHTML =
      document.getElementById(HTML).innerHTML
      // Cargamos los elementos del arbol
      + '<div class="assetWidget" id="' + i +
      '"onclick="updateAssetInput(' + i + ')"> <div class="assetWidget-
      img"><img src="' + path + '"><p>' + name + '</p></div></div>';
},
```

```
/**
*       AddObjects
*       Esta función es la encargada de gestionar la inserción de elementos en el mundo
*       catalog: Fichero JSON con los assets del editor
*       HTML: Identificador del div HTML
**/

 addObjects: function(catalog, HTML) {
        document.getElementById(HTML).innerHTML = "";
        for (var i = 0; i < group.length; i++) {
              cont[catalog[i].name] = 0;
              var j = ind[group.children[i].key];
              var k = group.children[i].name;
              this.addToObjects(catalog, HTML, j, i, k);
        }
 },

/**
*       AddToObjects
*       Esta función es la encargada de gestionar la inserción de cada elemento al mundo
        cuando es
*       seleccionado
*       catalog: Fichero JSON con los assets del editor
*       HTML: Identificador del div HTML
*       id: Identificador del elemento según su posición en los assets cargados
*       id_group: Identificador del elemento según el grupo
*       name: Nombre del elemento
**/


addToObjects: function(catalog, HTML, id, id_group, name) {
      var path = 'assets/images/icons/timeline/';
      document.getElementById(HTML).innerHTML =
      document.getElementById(HTML).innerHTML
      // Cargamos los elementos del arbol
      + '<div class="assetWidget" id="' + id + '"> <div
      class="assetWidget-img" onclick="updateAssetGroup(' + id_group +
      ')"><img src="' + catalog[id].path + '"><p>' + name + '</p><div
      class="optionButtons"><img src="' + path + 'more.png
      "onclick="updateAssetIndex(' + 1 + ',' + id_group + ')"><img src="'
      + path + 'less.png "onclick="updateAssetIndex(' + -1 + ',' +
      id_group + ')"><img src="' + path + 'eye.png"
      onclick="updateAssetView(' + id_group + ')"><img src="' + path +
      'delete.png" onclick="updateAssetRemove(' + id_group +
      ')"></div></div></div>';
},
```

```
/**
*       AddAssetWorld
*       Esta función es la encargada de añadir el elemento al mundo
**/

addAssetWorld: function() {
      // Leemos valor de assetInput
      var index = document.getElementById('variableAssetInput').value;
      if (index != -1) {
            var j = group.length;
            var k = cont[assetsJSON.catalog[index].name]++;
            if (assetsJSON.catalog[index].format == "sprite") {
                  var asset = group.create(this.game.camera.x,
                  this.game.camera.y, assetsJSON.catalog[index].name);
            }
            else if (assetsJSON.catalog[index].format == "image") {
                  var asset = group.create(this.game.camera.x,
                  this.game.camera.y, assetsJSON.catalog[index].name);
            }
            group.children[j].name = assetsJSON.catalog[index].name + k;
            k = group.children[j].name;
            var gridTam = document.getElementById('gridTam').value;
            //  Enable input and allow for dragging
            asset.inputEnabled = true;
            asset.input.enableDrag();
            // Añadimos controladores
            asset.useHandCursor = true;
            asset.events.onInputDown.add(this.clickListener, asset);
            asset.input.enableSnap(gridTam, gridTam, false, true);
            asset.events.onDragStart.add(this.onDrag, asset);
            asset.events.onDragStop.add(this.onDrop, asset);
            this.addToObjects(assetsJSON.catalog, 'assetsInWorld', index,
            j, k);
            document.getElementById('variableAssetInput').value = -1;
      }
},
```

```
/**
*       UploadAsset
*       Esta función es la encargada de “cargar” un nuevo asset en el catálogo
**/

uploadAsset: function() {
      if (document.getElementById('newAssetVariable').value == 1) {
            var name = document.getElementById('newAsset').name;
            var path = document.getElementById('newAsset').src;
            assetsJSON.catalog.push({
                  name: name,
                  path: path,
                  format: "image"
            });
            document.getElementById('newAssetVariable').value = 0;
            var index = assetsJSON.catalog.length - 1;
            var data = new Image();
            data.src = path;
            this.game.cache.addImage(name, path, data);
            cont[name] = 0;
            // Cargamos el nuevo Asset
            this.parserLine(assetsJSON.catalog[index].name,
            assetsJSON.catalog[index].path, 'assetsLoad', index);
      }
},
```