UNIVERSIDAD POLITÉCNICA SALESIANA

SEDE QUITO

CARRERA DE COMPUTACIÓN

DESARROLLO DE UN JUEGO PROTOTIPO UTILIZANDO EL MOTOR UNREAL ENGINE PARA VIDEOJUEGOS EN 3D

Trabajo de titulación previo a la obtención del

Título de Ingenieros en Ciencias de la Computación

AUTORES: KEVIN STALIN RAMÍREZ CAIZA

SEBASTIÁN GUSTAVO VELASCO GALARRAGA

TUTOR: JULIO RICARDO PROAÑO ORELLANA

Quito-Ecuador

2022

CERTIFICADO DE RESPONSABILIDAD Y AUTORÍA DEL TRABAJO DE TITULACIÓN

Nosotros, Kevin Stalin Ramírez Caiza con documento de identificación N° 1750031021 y Sebastián Gustavo Velasco Galarraga con documento de identificación N° 1752956175; manifestamos que:

Somos los autores y responsables del presente trabajo; y, autorizamos a que sin fines de lucro la Universidad Politécnica Salesiana pueda usar, difundir, reproducir o publicar de manera total o parcial el presente trabajo de titulación.

Quito, 17 de marzo del año 2022

Atentamente,

Kevin Stalin Ramírez Caiza
1750031021

Sebastián Gustavo Velasco Galarraga
1752956175

CERTIFICADO DE CESIÓN DE DERECHOS DE AUTOR DEL TRABAJO DE TITULACIÓN A LA UNIVERSIDAD POLITÉCNICA SALESIANA

Nosotros, Kevin Stalin Ramírez Caiza con documento de identificación No. 1750031021 y Sebastián Gustavo Velasco Galarraga con documento de identificación No. 1752956175, expresamos nuestra voluntad y por medio del presente documento cedemos a la Universidad Politécnica Salesiana la titularidad sobre los derechos patrimoniales en virtud de que somos autores del Proyecto Técnico: "Desarrollo de un juego prototipo utilizando el motor Unreal Engine para videojuegos en 3D", el cual ha sido desarrollado para optar por el título de: Ingeniero en Ciencias de la Computación, en la Universidad Politécnica Salesiana, quedando la Universidad facultada para ejercer plenamente los derechos cedidos anteriormente.

En concordancia con lo manifestado, suscribimos este documento en el momento que hacemos la entrega del trabajo final en formato digital a la Biblioteca de la Universidad Politécnica Salesiana.

Quito, 17 de marzo del año 2022

Atentamente,

Kevin Stalin Ramírez Caiza

1750031021

Sebastián Gustavo Velasco Galarraga

1752956175

CERTIFICADO DE DIRECCIÓN DEL TRABAJO DE TITULACIÓN.

Yo, Julio Ricardo Proaño Orellana con documento de identificación N° 0103909412, docente de la Universidad Politécnica Salesiana, declaro que bajo mi tutoría fue desarrollado el trabajo de titulación: DESARROLLO DE UN JUEGO PROTOTIPO UTILIZANDO EL MOTOR UNREAL ENGINE PARA VIDEOJUEGOS EN 3D, realizado por Kevin Stalin Ramírez Caiza con documento de identificación N° 1750031021 y por Sebastián Gustavo Velasco Galarraga con documento de identificación N° 1752956175, obteniendo como resultado final el trabajo de titulación bajo la opción Proyecto Técnico que cumple con todos los requisitos determinados por la Universidad Politécnica Salesiana.

Quito, 17 de marzo del año 2022

Atentamente,

Ing. Julio Ricardo Proaño Orellana. PhD

0103909412

RESUMEN

Al ser los videojuegos una de las grandes industrias del mundo el desarrollo de estos está regido en su mayoría por grandes compañías las cuales cuentan con los recursos y personal especializado para crearlos. En la evolución los video juegos fueron haciéndose más complejos y caros, por lo cual se comenzaron a desarrollar herramientas que sean baratas, rápidas y faciliten el desarrollo. Por esta razón se desarrolló este proyecto el cual busca mostrar cómo se puede crear un videojuego 3D con la ayuda de las herramientas que proporciona Unreal Engine 4. Para lograrlo se utilizaron dos etapas de la metodología ágil scrum postproducción y producción con ello se consiguió el diseño y su posterior implementación de un juego de genero plataformas. En colusión, se creó un videojuego 3D que tiene una alta calidad con base en los resultados obtenidos con la experiencia de los usuarios finales, esto en un corto periodo de tiempo y sin recursos monetarios.

Palabras claves

Unreal Engine 4, Videojuegos 3D, Beta, Scrum, jugabilidad, arquitectura de un motor gráfico, diseño de videojuegos, motores gráficos, desarrollo de videojuegos. Plataformas de distribución, mercado mundial de videojuegos.

ABSTRACT

As video games are one of the largest industries in the world, their development is mostly governed by large companies which have the resources and specialized personnel to create them. In the evolution of video games were becoming more complex and expensive, so they began to develop tools that are cheap, fast and facilitate development. For this reason this project was developed which seeks to show how you can create a 3D video game with the help of the tools provided by Unreal Engine 4. To achieve this we used two stages of the agile scrum methodology postproduction and production with it was achieved the design and subsequent implementation of a platform game genre. In collusion, a 3D video game was created that has a high quality based on the results obtained with the experience of end users, this in a short period of time and without monetary resources.

Keys words

Unreal Engine 4, Videogames 3D, Beta, Scrum, gameplay, architecture of graphics engine, video game design, motores gráficos, Video game development, Distribution platforms, Object orientated programming, widget.

ÍNDICE DE CONTENIDO

INTRODUCCIÓN **1**

1. CAPITULO 1: REVISIÓN DE LA LITERATURA O FUNDAMENTOS TEÓRICOS. **5**

1.1 VIDEOJUEGOS EN 3D 5

1.1.1 Calidad de un juego en base a su jugabilidad *6*

1.1.2 Metodología de producción y desarrollo. *10*

Preproducción. 11

Producción 13

Postproducción. 14

1.2 MOTORES GRÁFICOS 14

1.3 ARQUITECTURA GENERAL DEL MOTOR GRÁFICO 15

1.3.1 Hardware, drivers y sistema operativo *16*

1.3.2 SDKs y middlewares *17*

1.3.3 Capa independiente de la plataforma *18*

1.3.4 Subsistemas principales *18*

Biblioteca matemática 18

Estructura de datos y algoritmos 18

Gestión de memoria 19

Depuración y logging 19

1.3.5 Gestor de recursos *19*

1.3.6 Motor de rendering *19*

1.3.7 Motor de físicas *21*

1.3.8 Networking *22*

1.3.9 Subsistema de juego *23*

1.3.10 Audio *25*

1.4 MOTORES GRÁFICOS COMERCIALES 26

1.5 UNREAL ENGINE 4 27

2. CAPITULO 2: MARCO METODOLÓGICO. **33**

2.1 DISEÑO DEL VIDEOJUEGO 33

2.1.1 Términos Generales del juego *33*

Concepto del juego 33

Genero 33

Apariencia del juego 33

2.1.2 Mecánicas del juego *33*

Objetivos del juego 33

Acciones del personaje 34

Controles del juego 35

Interacción con el jugador 36

2.1.3 Menús e Interfaces *42*

Menú principal 42

Menú de juego 43

Interfaz durante la partida 44

Interfaz fin del juego 44

2.1.4 Mapa de juego *44*

Primer nivel del mapa 45

Segundo nivel del mapa 46

2.2 DESARROLLO E IMPLEMENTACIÓN 47

2.2.1. Creación del juego *47*

Selección del proyecto Unreal Engine 4 47

Configuración del proyecto 48

Agregación de elementos del Marketplace al proyecto 52

Creación del nivel 53
2.2.2. Implementación de los menús e interfaces. *57*
Menú principal 57
Menú de configuraciones 61
Menú de como jugar 63
Menú de juego. 64
Interfaz durante la partida 65
2.2.3. Implementación acciones del personaje. *67*
Acción agarrar y soltar 67
Acción saltar 71
Acción trepar 72
Acción rodar 73
Implementación de las llaves 75
Acciones de las puertas 78
2.2.4. Empaquetamiento *86*
Windows 10 88
Linux 89

3. CAPITULO 3: RESULTADOS. 93

3.1. Jugabilidad *96*
3.2. Rendimiento *98*

CONCLUSIONES 100

RECOMENDACIONES. 101

GLOSARIO DE TÉRMINOS 102

LISTA DE REFERENCIAS. 103

.....

ÍNDICE DE ILUSTRACIONES

Figura 1 *Valor del mercado de videojuegos.* 1
Figura 2 *Cantidad de estudios en América latina.* 2
Figura 3 *Cantidad de empleados.* 2
Figura 4 *Experiencia de usuario.* 6
Figura 5 *Atributos de usabilidad y jugabilidad.* 8
Figura 6 *Etapas en el desarrollo de videojuegos.* 11
Figura 7 *Arquitectura motora gráfico* 16
Figura 8 *Gestor de recursos.* 19
Figura 9 *Motor de renderizado.* 20
Figura 10 *Arquitectura del subsistema de juego.* 23
Figura 11 *Blueprint visual scripting.* 30
Figura 12 *Actor.* 30
Figura 13 *Personaje.* 31
Figura 14 *Controles del juego.* 35
Figura 15 *Llaves.* 36
Figura 16 *Prensa.* 36
Figura 17 *Esqueletos.* 37
Figura 18 *Gárgola.* 37
Figura 19 *Mensajes.* 38
Figura 20 *Puerta.* 38
Figura 21 *Gárgola de ambientación.* 39
Figura 22 *Puerta de piedra.* 39
Figura 23 *Antorchas.* 40
Figura 24 *Rejas.* 40
Figura 25 *Columnas.* 41
Figura 26 *Jaulas.* 41
Figura 27 *Mapa general del juego.* 44
Figura 28 *Primer nivel.* 45
Figura 29 *Gárgola* 46
Figura 30 *Segundo nivel.* 46
Figura 31 *Pasadizo final.* 47
Figura 32 Creación de proyecto. 48
Figura 33 *Plantillas.* 48
Figura 34 *Configuraciones del proyecto.* 49
Figura 35 *Configuración Blueprint.* 49
Figura 36 *Configuración calidad máxima.* 50
Figura 37 *Configuración de escritorio o consola.* 50
Figura 38 *Configuración contenida de inicio.* 51
Figura 39 *Configuración raytracing.* 51
Figura 40 *Guardar el proyecto.* 52
Figura 41 *Selección del proyecto.* 52
Figura 42 *Assets Medieval Dungeon.* 53
Figura 43 *Carpeta de texturas.* 53
Figura 44 *Paleta de Shaders.* 54
Figura 45 *Implementación del diseño del mapa.* 54

Figura 46 *Estructura externa del mapa nivel 1.* 55
Figura 47 *Estructura de los subniveles.* 55
Figura 48 *Estructura externa del mapa nivel 2.* 56
Figura 49 *Techo del mapa.* 56
Figura 50 *Agregación de tag.* 57
F*igura 51 Widget Blueprint Menú Principal.* 57
Figura 52 *Widget Blueprint MenuPrincipal botones.* 58
Figura 53 *MenuPrincipalC++.* 58
Figura 54 *Inicialización Componentes Widget.* 59
Figura 55 *Ejemplo de Función Play.* 60
Figura 56 *Función Play.* 60
Figura 57 *Widget Blueprint Menú Opciones.* 61
Figura 58 *Menú Opciones 1.* 61
Figura 59 *Menú Opciones 2.* 62
Figura 60 *Funcionalidad Botón Opciones.* 62
Figura 61 *Funcionalidad Botón Atrás Opciones.* 63
Figura 62 *WidgetBlueprint Como Jugar.* 63
Figura 63 *Como Jugar C++.* 64
Figura 64 *Widget Blueprint MenuInGame.* 64
Figura 65 *MenuInGame C++.* 65
Figura 66 *Widget Blueprint PlayerHud.* 66
Figura 67 *PlayerHud C++.* 66
Figura 68 *Acción Agarrar 1.* 67
Figura 69 *Acción Agarrar 2.* 68
Figura 70 *Acción Agarrar C++.* 69
Figura 71 *Acción Agarrar función SetGrab.* 69
Figura 72 *Acción Agarrar Físicas 1.* 70
Figura 73 *Acción Agarrar Físicas 2 función Tick.* 70
Figura 74 *Acción Saltar.* 71
Figura 75 *Saltar clase padre.* 71
Figura 76 *Función Saltar.* 72
Figura 77 *Función Trepar.* 72
Figura 78 *Funcionalidad Trepar C++.* 73
Figura 79 *Acción Rodar.* 74
Figura 80 *Rodar C++.* 74
Figura 81 *Rodar utilizando Tick C++.* 75
Figura 82 *ARandomKeys en el editor.* 75
Figura 83 *BP_RandomKeys.* 76
Figura 84 *RandomKeys C++.* 77
Figura 85 *Acción Abrir Puerta C++.* 77
Figura 86 *ClaseHijo OpenDoor C++.* 78
Figura 87 *Clase PadreDoor C++.* 78
Figura 88 *Puerta ChildDoorLevel0.* 79
Figura 89 *Puerta ChildDoorLevel0 C++.* 79
Figura 90 *Puerta ChildDoorLevel0 Overlap C++.* 80
Figura 91 *Puerta ChildDoorLevel0 MoverPuerta C++.* 80
Figura 92 *Puerta ChildDoorLevel1.* 81
Figura 93 *Puerta ChildDoorLevel1 C++.* 81

Figura 94 *Puerta ChildDoorLevel1 Overlap C++.* 82
Figura 95 *Puerta ChildDoorLevel2.* 82
Figura 96 *Puerta ChildDoorLevel2 C++.* 83
Figura 97 *Puerta ChildDoorLevel2 Overlap C++.* 83
Figura 98 *Puerta ChildDoorWithKeys.* 84
Figura 99 *Puerta ChildDoorWithKeys C++.* 84
Figura 100 *Puerta ChildDoor.* 85
Figura 101 *Puerta ChildDoor C++.* 85
Figura 102 *Ventana ProjectSettings.* 86
Figura 103 *Cook only maps.* 86
Figura 104 *Agregar mapas.* 87
Figura 105 *Plataformas por defecto.* 87
Figura 106 *Selección drivers.* 88
Figura 107 *Imagen Ico.* 88
Figura 108 *Guía Empaquetado Windows.* 89
Figura 109 *Clang.* 89
Figura 110 *Variables del sistema Windows 10.* 90
Figura 111 *Verificación de instalación de la herramienta.* 90
Figura 112 *Configuración de visual Studio* 91
Figura 113 *Proyecto* 91
Figura 114 *Opción compilar* 91
Figura 115 *Package proyect* 92
Figura 116 *Mundo del videojuego* 93
Figura 117 *Clasificación por edad* 94
Figura 118 *Clasificación por genero* 95
Figura 119 Clasificación por experiencia previa 95
Figura 120 *Eficiencia del juego* 96
Figura 121 *Calidad de los gráficos* 97
Figura 122 *Animación del personaje* 97
Figura 123 *Satisfacción en general* 98
Figura 124 *Rango de FPS* 98

ÍNDICE DE TABLAS

TABLA 1 *MOTORES GRÁFICOS COMERCIALES* **26**

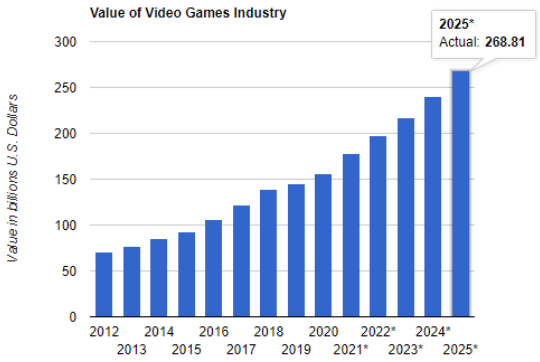
TABLA 2 *INDUSTRIAS QUE UTILIZAN UNREAL ENGINE* **32**

INTRODUCCIÓN

A nivel mundial la industria de los videojuegos represento un valor de $178,73 mil millones en 2021, que representa un aumento del 14,4% con relación al 2021. Por otro lado, se estima que para el presente año 2022 representara un valor de $197,11 mil millones. Según los últimos pronósticos publicados en enero del 2022 se estima que la industria para el año 2025 tendrá un valor en el mercado de $268 mil millones como lo muestra la figura1.(statistics, 2022).

Figura 1

Valor del mercado de videojuegos.



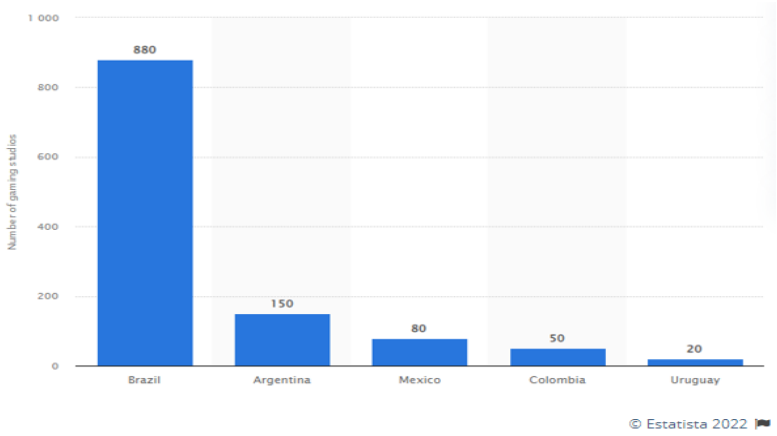
Nota: Valor que se pronostica que tendrá la industria de los videojuegos en los próximos 3 años, Fuente: (statistics, 2022).

En América latina se proyecta que los videojuegos generaran $6290 millones para el año 2023. Dentro de la región México y Brasil se ubican como los principales mercados de videojuegos, generando un estimado de $1,900 millones y $1,750 millones de dólares, respectivamente. En septiembre del 2019 Brasil contaba con 880 estudios de juegos lo que lo convierte en el líder latinoamericano en cantidad de negocios dedicados a la creación y desarrollo de videojuegos. Le siguieron Argentina con 150 y México con 80 estudios como se muestra en la figura 2 .(Department, 2021a, 2021c). Adicionalmente el número de empleados

en la industria de los videojuegos en Brasil llegó a 4.000 personas. Esto es aproximadamente 2.000 trabajadores más en el sector que en Argentina. Por su parte, México contaba con una plantilla total de 1.300 personas, seguido de Colombia con 400 y Uruguay con 150 empleados como se muestra en la figura 3.(Department, 2021b, 2021c).

Figura 2

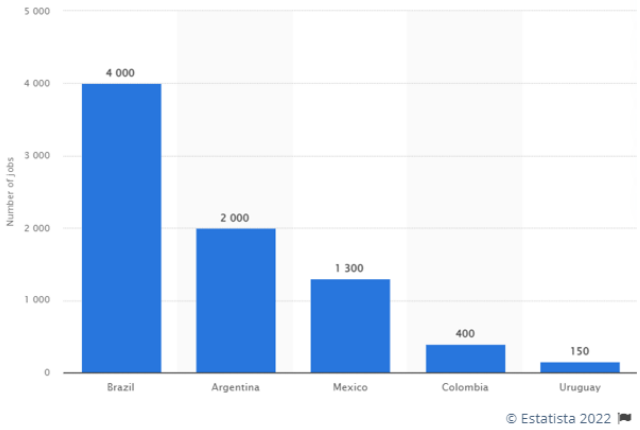
Cantidad de estudios en América latina.



Nota: Número de estudios para desarrollo de videojuegos en América Latina. Fuente: (Department, 2021c).

Figura 3

*Cantidad de empleados***.**



Nota: Número de trabajadores de la industria de videojuegos en América Latina. Fuente: (Department, 2021b).

Ecuador no forma parte del panorama económico generado por la industria de los videojuegos y ve dejos el desarrollo de la industria, el país carece de asociaciones y normativas que regulen la producción de software, sin embargo, cuenta con algunos estudios que desarrollan videojuegos como lo son Blue Lizard Game, Arrarray Estudios, Poquito Games y Freeky Creations, también se encuentran Agencias de publicidad digital como Paradais y Geeks.(Andres, 2018).

Con la iniciativa de impulsar el desarrollo de videojuegos en América Latina propuesta por Play Station, el estudio Freeky Creations, con un grupo de 10 personas comenzaron el desarrollo de "TO LEAVE", el cual es el primer juego echo totalmente en Ecuador que fue presentado en E3, la más grande feria de videojuegos del mundo donde tuvo buenas críticas. El juego está disponible en la tienda de Play Station para la consola PS4.(Primicias.ec).

El desarrollo de videojuegos 3D desde sus inicios ha estado en manos de grandes compañías, las cuales cuentan con las capacidades y recursos necesarios para para desarrollar videojuegos. En el transcurso del tiempo los videojuegos fueron haciéndose más complejos y caros en su proceso, por lo cual se comenzaron a desarrollar distintas herramientas para facilitar su desarrollo. (Gil Romero, 2014) Con ellos aparece el motor grafico o Game Engine el cual que hace referencia a una serie de rutinas de programación que ayuda con el diseño, creación y la representación del videojuego 3D. La funcionalidad del motor grafico es proveer el renderizado para gráficos, motor físico o detector de colisiones, sonido, scripting, animación, inteligencia artificial, redes y streaming. En la actualidad es impensable crear un videojuego 3D sin un motor grafico que lo soporte. (Cleto, 2013)

La relevancia de este proyecto radica en dar a conocer la tecnología de motores gráficos. Además de las herramientas que posee el motor Unreal Engine para el desarrollo de videojuegos 3D, con eso se busca una posible opción para que en un futuro exista un creciente incentivo en

el desarrollo de videojuegos en el país. Los beneficiados de este proyecto son programadores independientes que les guste el mundo de los videojuegos y quieran empezar su camino con poco presupuesto y muchas ganas de aprender. Las empresas de desarrollo también tienen cabida ya que con el uso de las nuevas tecnologías se puede crear grandes proyectos a bajo costo, formando parte de una industria que cada vez es más rentable.

Como objetivo se propone desarrollar un juego prototipo utilizando el motor Unreal Engine 4, iniciando con entender cuál es el funcionamiento del motor para poder diseñar el mapa donde se implementa toda la funcionalidad asociada al juego, además de crear interfaces de usuario con el fin de agregar interactividad, para las pruebas se orientó en la jugabilidad con base en la experiencia de usuario, obteniendo resultados que comprueban que el juego tiene gran expectativa.

En el documento se encuentran temas relacionados al desarrollo de videojuegos teniendo en cuenta el diseño, implementación, y pruebas. Todo esto bajo el control de la metodología scrum, la se orientó al software de videojuegos. Además, se contextualiza que es un motor gráfico mostrando de manera general su arquitectura e indicando cuales son los más importantes del mercado. Finalmente, en la etapa de evaluación se sometió el juego a pruebas de jugabilidad por parte de un grupo usuarios.

1. CAPITULO 1: REVISIÓN DE LA LITERATURA O FUNDAMENTOS TEÓRICOS.

En la presente sección se presentará información necesaria para describir los principales fundamentos sobre los que se basa el desarrollo de videojuegos 3D, además presentar los actuales motores gráficos y las herramientas que facilitan la creación de videojuegos. Esto con el objetivo de establecer un lenguaje común que permitirá avanzar con el resto del trabajo.

1.1 VIDEOJUEGOS EN 3D

Un videojuego 3D se concibe como un medio de entretenimiento donde interactúan uno o varios usuarios llamados Players o Jugadores, los cuales toman un papel activo dentro de un universo simulado e interactúan con el mismo a través de varias interfaces como pueden ser controles, teclado, mouse, entre otros, además de un dispositivo e video como monitor de PC, TV, Realidad Virtual, etc. (Arce, 2011) (Iglesias & Blanque, 2011).

Los videojuegos en 3D poseen el suficiente atractivo y despiertan la motivación como para que los jugadores se sientan conectados con la dinámica interna, esta sumada a un alto valor de estimulación auditiva, visual y la incorporación de niveles de dificultad progresivos lo hacen uno de los mejores medios de entretenimiento.(Arce, 2011).

A lo largo de 30 años de desarrollo, los videojuegos han incorporado nuevas características y capacidades tecnológicas como la integración de múltiples lenguajes audiovisuales en un mismo medio, interoperabilidad, capacidades de procesamiento de información y conectividad.(Iglesias & Blanque, 2011).

1.1.1 Calidad de un juego en base a su jugabilidad

Al desarrollar sistemas interactivos, es muy importante que el usuario se involucre para que tenga la seguridad de que se brindará una buena experiencia de usuario. El software de videojuegos puede considerarse un caso especial de un sistema interactivo.

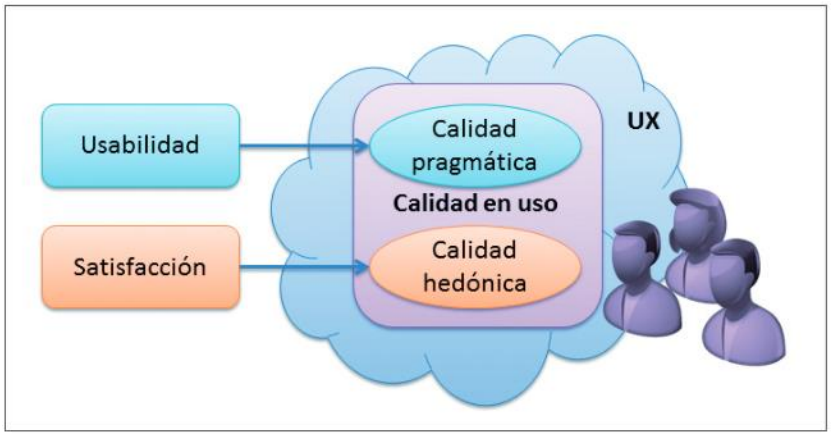
En el desarrollo de videojuegos es muy importante que el jugador sienta las mejores experiencias al momento de interactuar con el juego, experiencias como entretenimiento, diversión, etc. El incremento de estas experiencias influye directamente en el éxito del videojuego, por lo cual es importante conocer las principales características de estas experiencias para así poder medirlas durante el desarrollo y asegurar su éxito y calidad. Este concepto es conocido como experiencia del jugador (UX). (David Villa, 2015).

Una característica de un videojuego es la **jugabilidad** y esta se extiende del concepto de **usabilidad** con ello se puede lograr medir la experiencia que tiene el usuario al usar el juego. (David Villa, 2015).

Se define a la usabilidad como la capacidad que tiene un software para ser usado, aprendido, entendido y que genere satisfacción en el usuario. Para entender y medir la usabilidad se identifica una colección de propiedades como lo son efectividad, eficiencia, satisfacción, aprendizaje y seguridad.(González-Sánchez, Montero-Simarro, & Gutiérrez-Vela, 2012).

Figura 4

Experiencia de usuario.



Nota: Caracterización de la experiencia del usuario a través de la calidad en uso. Fuente: (González-Sánchez et al., 2012).

La jugabilidad se entiende como la calidad de uso de un videojuego y esta extiende el concepto de usabilidad se puede apreciar de mejor manera en la figura 4, los atributos que caracterizan a la jugabilidad son los siguientes:

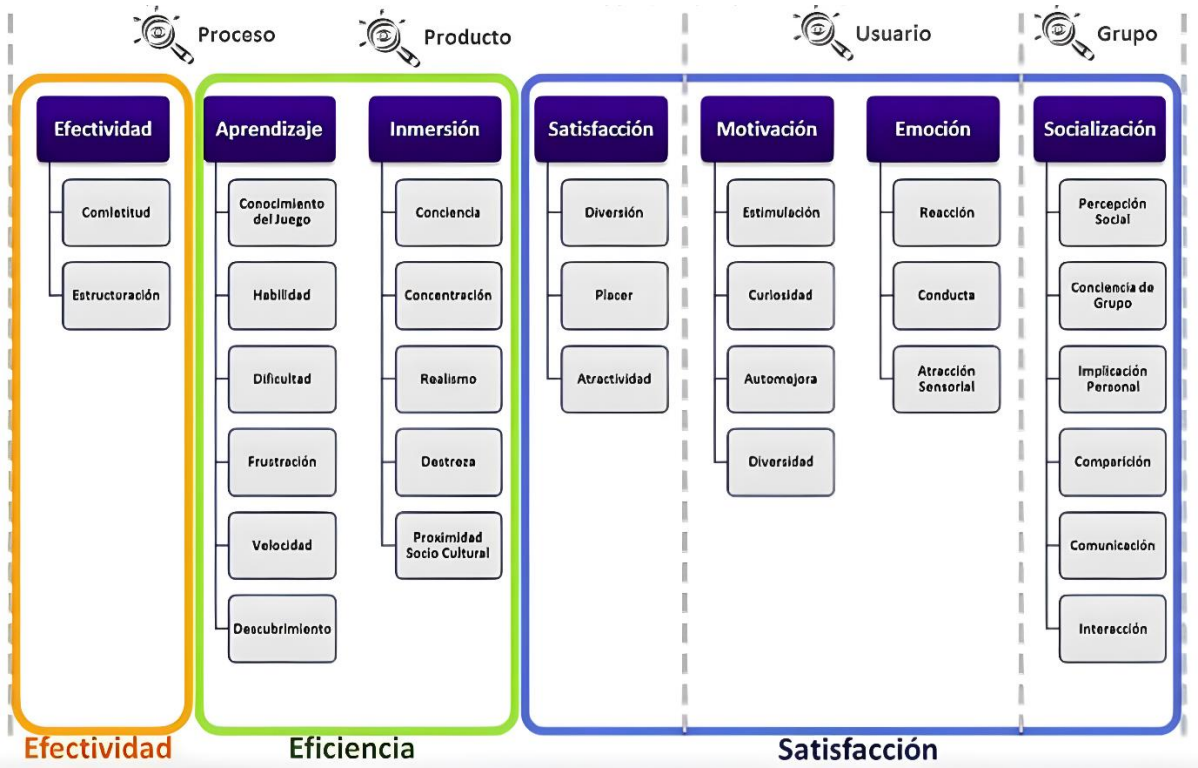
- Satisfacción. Se refiere al nivel de agrado del jugador ante el videojuego y el transcurso de jugarlo.
- Aprendizaje. Se refiere a la sencillez para entender y dominar el sistema y la mecánica del videojuego.
- Efectividad. Se refiere a los recursos y tiempo necesario para ofrecer diversión al jugador mientras éste logra los objetivos propuestos y alcanza la meta final del videojuego.
- Inmersión. Se refiere a la capacidad para integrarse en el mundo virtual del videojuego.
- Motivación. Se refiere a las características del videojuego que incentivan al jugador a realizar acciones definidas y a persistir en ellas hasta su culminación.
- Emoción. Se refiere al impulso involuntario que surgen en respuesta a los estímulos del videojuego.

- Socialización. Se refiere a los atributos que hacen valorar el videojuego de distinta manera al jugarlo en equipo, ya sea de manera competitiva, colaborativa o cooperativa.

Se puede ver en la figura 5 como estos atributos pueden relacionarse con el conceto de usabilidad. (David Villa, 2015).

Figura 5

Atributos de usabilidad y jugabilidad.



Nota: relación que existe entre los atributos de usabilidad y jugabilidad. Fuente: (David Villa, 2015)

La experiencia del jugador en base a la jugabilidad mediante las propiedades, atributos y facetas proporciona medidas de calidad de las pericias durante el juego. Con ello se puede utilizar el estándar de calidad ISO 25010:2011 en el entorno de los videojuegos. A continuación, se presentan las métricas para el caso de los videojuegos.

- Efectividad. Se define como el grado con el que los jugadores pueden lograr los objetivos propuestos con precisión y completitud en un contexto particular.
- Eficiencia. Se define como el grado con el que los jugadores pueden son capaces de alcanzar sus objetivos planteados invirtiendo una cantidad adecuada de recursos en relación con el desempeño alcanzado, que está determinado por la facilidad de aprendizaje y la inmersión.
- Flexibilidad. Se define como el grado con el que el videojuego se puede jugar en distintos marcos posibles o por distintos perfiles de jugadores y de juegos existentes.
- Seguridad/Prevención. Se define como el nivel aceptable de riesgo para la salud del jugador.
- Satisfacción. Se define como el grado con el que los jugadores están satisfechos con el juego en el contexto de un uso concreto, en este factor consideramos distintos atributos como agrado atracción placentero confortable confiable motivado emocionable y sociable.

La principal forma de medición es la observación, se pude utilizar cuestionarios o test heurísticos para preguntar o interrogar los atributos de la jugabilidad. También se toma en cuenta el rendimiento que tiene el videojuego al momento de ejecutarse, una de las herramientas para medir esto son los denominados FPS ya que estos se relacionan directamente con los recursos de hardware en donde se ejecuta el juego. (David Villa, 2015).

Se conoce como FPS al número de fotogramas renderizados por segundo, los valores altos de este número dan como resultado un movimiento fluido, claro y más detallado, mientras que los valores bajos de fotogramas causan un movimiento cortado que da una sensación como de saltos entre fotogramas. (Janzen & Teather, 2014).

Este tiempo entre fotogramas se le suele conocer con el nombre de latencia y cuando los fotogramas por segundo bajan el valor de la latencia incrementa, además que con tasas de FPS bajas es mucho más difícil entender que está pasando dentro del juego debido a que se relaciona con la percepción de movimiento de los objetos dentro del juego. (Janzen & Teather, 2014).

Se recomienda que los de videojuegos corran a 45 FPS esto demuestra que el juego presenta un buen rendimiento y una imagen nítida, los valores de FPS menores a 30 demuestran que el juego se ejecuta de manera estable a regular esto quiere decir que el jugador no obtendrá el mejor rendimiento al jugar el videojuego y puede sufrir de escenas con tirones.

1.1.2 Metodología de producción y desarrollo.

Como en el desarrollo de cualquier producto de software para la realización de videojuegos, es necesario tener en cuenta los fundamentos de la ingeniería de software y sobre todo la metodología de desarrollo adecuada al producto que se va a utilizar. sin embargo el desarrollo de un videojuego no solo se reduce al desarrollo técnico de un producto de software sino que supone una actividad multidisciplinar que comprende desde la idea y concepción inicial hasta su versión final.(David Villa, 2015).

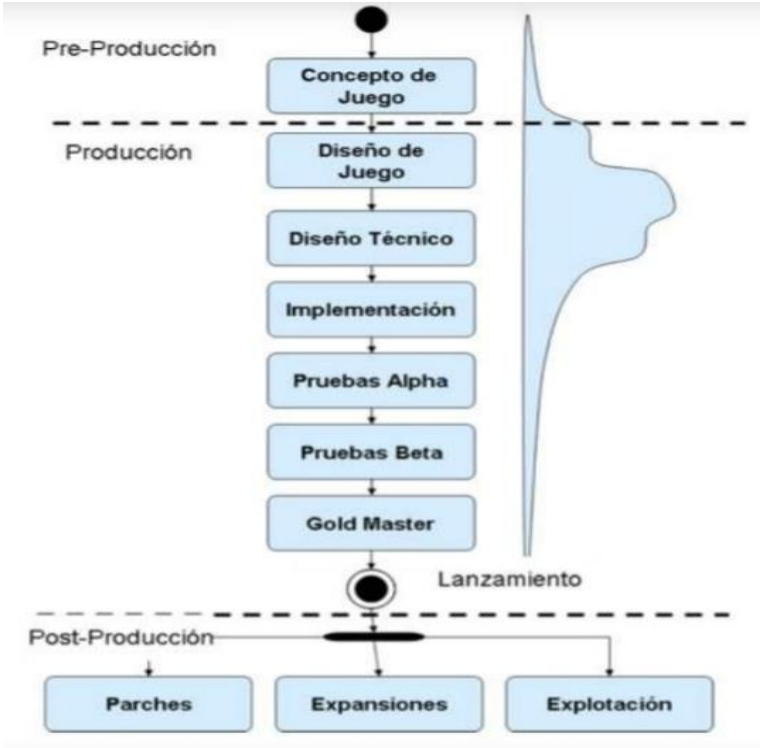
Crear un juego es una tarea delicada y requiere de una metodología específica, sin embargo, las metodologías establecidas para el desarrollo de software no son adecuadas para este proceso, ya que no cuentan con suficiente garantía de calidad y no existe un enfoque claro en esta área sobre cómo hacerle frente.

Scrum es una metodología ágil para dirigir y controlar el desarrollo de software de un producto en forma iterativa e incremental. Una de sus características es que no indica prácticas específicas a seguir durante el desarrollo (Pekka, Outi, Jussi, & Juhani, 2002), lo que brinda

flexibilidad y permite ajustar el proceso a la realidad y forma de trabajo de cada proyecto, así como a los diferentes requerimientos de los clientes. Según la descripción que realiza Ken Schwaber (Schwaber & Beedle, 2002), Scrum se estructura en tres fases denominadas pre-game, game y post-game o también conocidas como preproducción, producción y postproducción. A continuación, se muestra en la figura 6 las etapas.

Figura 6

Etapas en el desarrollo de videojuegos.



Nota: Faces que muestran cada etapa de la producción de un videojuego. Fuente: (David Villa, 2015).

Preproducción. Esto sucede en la conceptualización del juego, identificando los elementos esenciales y, si es posible, finalizando en su diseño conceptual. Esta información está organizada y forma lo que se considera una versión del documento de diseño del juego o GDD (Game Design Document). Este GDD será desarrollado por el equipo creativo de diseño de videojuegos, cualquier cosa relacionada con el diseño de videojuegos que deba abordarse

más adelante en la producción debe identificarse y corregirse. (David Villa, 2015). El documento de diseño del juego puede contener lo siguiente:

- **Género**: Clasificación del juego según su esencia, determinar a qué genero pertenece dará una pauta a una serie de características imprescindibles para su posterior diseño.
- **Jugadores**: Modalidad de juegos, individual, colectivo o multijugador si los jugadores son personas o máquinas.
- **Historia**: Resumen de la historia del juego, se realiza una pequeña trama o historia la cual se desarrolla durante el juego. Esto se denomina storyline y storytelling respectivamente.
- **Bocetos y Look and feel**: Los bocetos son diseños preliminares fundamentales de los personajes y de los escenarios. A partir del boceto se define un aspecto gráfico y artístico del juego colores, temas dominantes, musicalidad, técnicas de diseño 3D, posiciones de Cámara. Etc.
- **Interfaz de Usuario**: Es la herramienta en la que el jugador interactúa con el juego.
- **Objetivos**: en este apartado se determinan las metas del juego de acuerdo con la historia que se va a desarrollar.
- **Reglas**: Se establece qué acciones podrá desarrollar el jugador y cómo hacerlo.
- **Diseño de niveles**: Se describen los niveles de dificultad que presenta el juego indicando cuántos serán y cómo serán, así como los retos a los que el jugador se deberá enfrentar.

Como ya se indicó, esta primera versión del documento diseño del juego será el punto de partida para el inicio de la etapa de producción.

Producción. La etapa de producción es donde se concentra el trabajo principal, el tamaño y número de personas involucradas, es el proceso de desarrollo del videojuego. (David Villa, 2015). Se pueden identificar algunas de las principales etapas como:

Diseño del juego: Esta es la etapa primaria en la que se describen con un alto grado de detalle todos los elementos que formarán parte del juego, y básicamente lo que se hace es afinar lo previsto en GDD.

Diseño de la mecánica del juego: en esta etapa se trabajan aspectos como.

- Cuáles son las reglas que lo rigen y cuál es la comunicación que tendrá si se trata de un juego online.
- Se diseña el comportamiento, habilidades y otros detalles de los personajes y el mundo que los rodea
- Se empieza a trabajar en el diseño del motor de IA (Inteligencia Artificial).
- Se diseña el denominado motor físico con el propósito de crear los aspectos físicos del juego como explosiones, disparos, choques, etc.

Un motor de juego se refiere a una serie de procesos que permiten mostrar todos los elementos funcionales de un juego, es decir, cualquier cosa relacionada con motores gráficos, el motor de sonido, el motor de IA, el motor físico y todo el resto de los gestores que pueden ser necesarios para manejar el mundo virtual completo del videojuego.

Implementación: En esta etapa debe abordarse el establecimiento de los elementos programáticos del proyecto descritos en la etapa anterior utilizando métodos y herramientas de ingeniería. En muchos casos, esta etapa y la anterior se repiten o siguen en ciclos de repetición. Por lo general, en esta etapa se suelen crear y publicar pequeñas demos del juego, que ayudan

a implementar la campaña de marketing y publicidad necesaria para el éxito comercial del producto.(David Villa, 2015).

Pruebas Alpha. Estas pruebas se resuelven cuando se completa parte del producto de software, y el producto se somete a varias pruebas realizadas por un pequeño equipo que se encarga del diseño y desarrollo del juego. El objetivo de estas pruebas es encontrar pequeños errores y mejorar ciertos aspectos.(David Villa, 2015).

Pruebas Beta. En estas pruebas se completa todo lo relacionado con el encuadre de misiones, gráficos, texto en diferentes idiomas, voces en off y más. terminado. Además, estamos trabajando arduamente para garantizar que el contenido del juego cumpla con las leyes aplicables y la ética establecida en el país donde se pretende lanzar el juego.(David Villa, 2015).

Gold máster. Esta fase se refiere a la prueba final, que incluye los puntos finales que se lanzarán y producirán, y debe publicitarse tanto como sea posible en este punto, incluida la implementación. de reportajes, artículos etc.(David Villa, 2015).

Postproducción. Es la última fase en el desarrollo del videojuego, básicamente, trata la operación y el mantenimiento del juego como otros productos de software.

1.2 MOTORES GRÁFICOS

Un motor Dado que los componentes gráficos son una parte fundamental de cualquier juego y necesitan mejoras constantes, el motor de renderizado es una de las partes más complejas de cualquier motor gráfico. El enfoque más utilizado para el diseño del motor de renderizado es una arquitectura en capas figura 9, en cuanto a la arquitectura del motor, o el propio motor del juego, se trata de un conjunto de motores que realizan los distintos cálculos geométricos y físicos que se utilizan en los videojuegos. Esta colección representa un simulador

flexible en tiempo real que crea las características del mundo de fantasía en el que se desarrollan los videojuegos, basa su arquitectura a la reutilización, cuenta con componentes como el renderizado para gráficos, motor físico o detector de colisiones, sonido, scripting, animación, inteligencia artificial, redes, streaming, el sistema de audio y lo elementos más artísticos; como los escenarios virtuales o las reglas del juego.(Cleto, 2013).

Los motores gráficos datan de los años 90 con la finalidad de separar los diversos elementos del juego. Separando el núcleo de un videojuego (sistema de colisión, sonido, sistema de renderizado 2D o 3D) de otros activos (arte, físicas y reglas del juego) para que se pueda crear la estructura inicial a partir de la cual se puedan desarrollar múltiples videojuegos.

Un motor de videojuegos debe ofrecer como mínimo las siguientes utilidades:

- Motor 3D que permita la creación y visualización de un juego en un mundo 3D.
- Motor de audio que permita la incorporación de elementos sonoros y música en el juego.
- Motor físico esto hace posible gestionar comportamientos físicos en el universo 3D, como la gravedad.
- Herramientas de gestión de red para añadir por ejemplo un componente multijugador online vía internet.

1.3 ARQUITECTURA GENERAL DEL MOTOR GRÁFICO

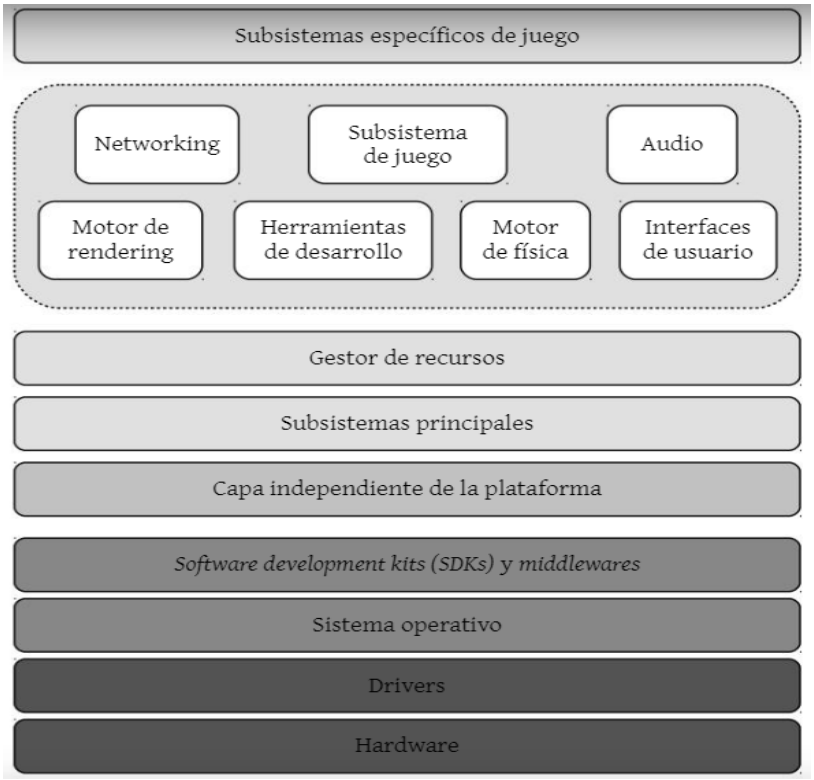
Esta sección planea dar una visión general de la arquitectura del mor gráfico haciendo énfasis a los bloques más sobresalientes desde el punto de vista del desarrollo de videojuegos.

Como la mayoría de los sistemas de software altamente complejos, los motores gráficos se basan en una arquitectura en capas. De esta forma, las clases de alto nivel dependen de las clases de bajo nivel, pero no al revés. Esto le permite agregar capas gradualmente y, lo que es

más importante, cambiar algunos aspectos de una capa en particular sin afectar a otras. En la figura 7 se describen los principales módulos que forman parte de la arquitectura.

Figura 7

*Arquitectura motora gráfico***.**



Nota: Visión general de los componentes que presenta la arquitectura del motor gráfico. Fuente: (Cleto, 2013).

1.3.1 Hardware, drivers y sistema operativo

La capa de hardware está relacionada con la plataforma en la que se ejecutará el motor, un tipo específico de plataforma podría ser una consola de videojuegos de escritorio. Muchos principios de diseño y desarrollo son comunes a todos los videojuegos, independientemente de la plataforma de implementación final.(Cleto, 2013).

La capa de drivers soporta componentes de software de bajo nivel que permiten una gestión precisa de ciertos periféricos, como tarjetas aceleradoras de gráficos o tarjetas de sonido.

La capa del sistema operativo representa la capa de comunicación entre los procesos que se ejecutan en ella y los recursos de hardware dependientes de la plataforma. En el mundo de los videojuegos, los sistemas se compilan tradicionalmente junto con el propio juego para producir archivos ejecutables. Sin embargo, la última generación de consolas, como Sony PlayStation o Microsoft Xbox 360, incluyen un sistema operativo que puede controlar ciertos recursos o incluso interrumpir un juego en curso.(Cleto, 2013).

1.3.2 SDKs y middlewares

El desarrollo de un motor de videojuegos se suele apoyar de bibliotecas existentes y SDK para proporcionar una determinada funcionalidad, al igual que en otros proyectos de software. Aunque el software está bien optimizado, algunos desarrolladores prefieren personalizarlo según sus necesidades, especialmente en consolas domésticas y portátiles.

Un ejemplo representativo de gestión de estructuras de datos es STL (Standard Template Library). STL es la biblioteca de plantillas estándar de C++ y actualmente es el lenguaje más utilizado para el desarrollo de videojuegos debido a su portabilidad y eficiencia..(Cleto, 2013).

En el campo de los gráficos 3D, hay una serie de bibliotecas que se ocupan de la representación de modelos 3D y más. Los mejores ejemplos son las API de gráficos OpenGL y Direct3D mantenidas por los equipos de Khronos y Microsoft. El objetivo principal de dichas bibliotecas es ocultar diferentes aspectos de la tarjeta gráfica, representando una interfaz común. Direct3D está totalmente ligado a los sistemas Windows, mientras que OpenGL es multiplataforma.(Cleto, 2013)

La rama de la inteligencia artificial en los videojuegos también se beneficia de herramientas que pueden interactuar directamente con bloques de bajo nivel para resolver

problemas clásicos, como la búsqueda óptima de caminos entre dos puntos o acciones y la evasión de obstáculos.

1.3.3 Capa independiente de la plataforma

Durante el desarrollo de una gran cantidad de juegos, se tiene presente la posibilidad de su lanzamiento en diferentes plataformas. Por ejemplo, se puede desarrollar un título para múltiples plataformas de escritorio y PC simultáneamente. Por lo tanto, es común encontrar una capa de software que aísle al resto de las capas anteriores de cualquier aspecto dependiente de la plataforma, y esta capa se denomina capa independiente de la plataforma.

Algunos ejemplos de módulos contenidos en esta clase son de manejo de hijos o los wrappers o contenedores sobre algunos módulos de nivel superior, como módulos de detección de colisiones o módulos portadores. Responsable de gráficos.(Cleto, 2013).

1.3.4 Subsistemas principales

Las clases principales del subsistema están relacionadas con cualquier utilidad o biblioteca que admita el motor. Algunos de ellos son específicos del campo de los videojuegos, mientras que otros son comunes a proyectos de software bastante complejos de cualquier tipo. Algunos de los subsistemas más relevantes se enumeran a continuación:

Biblioteca matemática. Responsable de proporcionar a los desarrolladores una variedad de utilidades para facilitar operaciones que involucren vectores, matrices, cuaterniones u operaciones que involucren líneas, rayos, esferas y otras geometrías. Las bibliotecas matemáticas son esenciales en el desarrollo de motores de juegos porque los juegos son de naturaleza matemática.

Estructura de datos y algoritmos. Responsable de proporcionar implementaciones más optimizadas y personalizadas de varias estructuras de datos (como listas vinculadas o árboles binarios) en bibliotecas como STL y algoritmos como búsqueda y clasificación. Este subsistema

es importante cuando la plataforma en la que se ejecuta el motor tiene memoria limitada, como suele ser el caso de las consolas domésticas.

Gestión de memoria. Responsable de garantizar que la memoria se asigne y libere de manera eficiente.

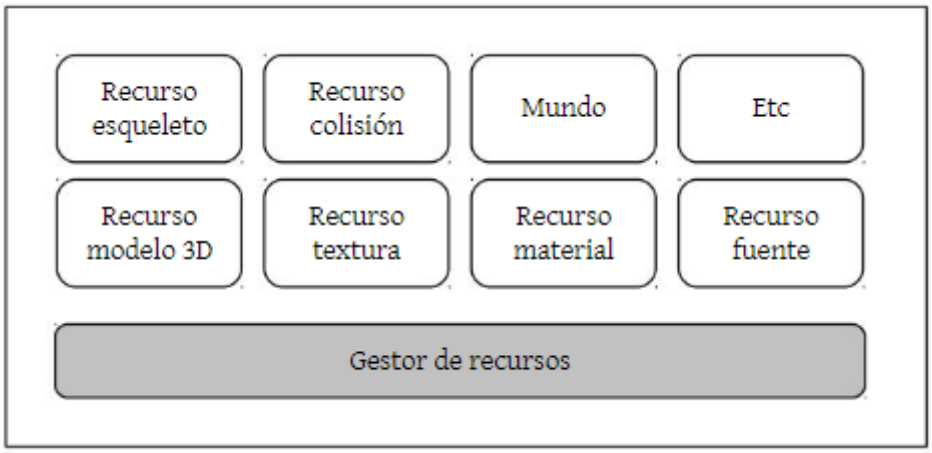
Depuración y logging. Responsable de brindar herramientas para facilitar la depuración y volcado de logs para su posterior análisis.

1.3.5 Gestor de recursos

Proporciona una interfaz que nos permite acceder a las diferentes unidades de software que componen el motor. (Cleto, 2013).

Figura 8

Gestor de recursos.



Nota: Gestor de recursos ilustrado desde una visión general. Fuente: (Cleto, 2013).

La figura 8 muestra una descripción general del gestor de recursos, que representa una interfaz común para administrar diferentes entidades, como objetos 3D, texturas o materiales.

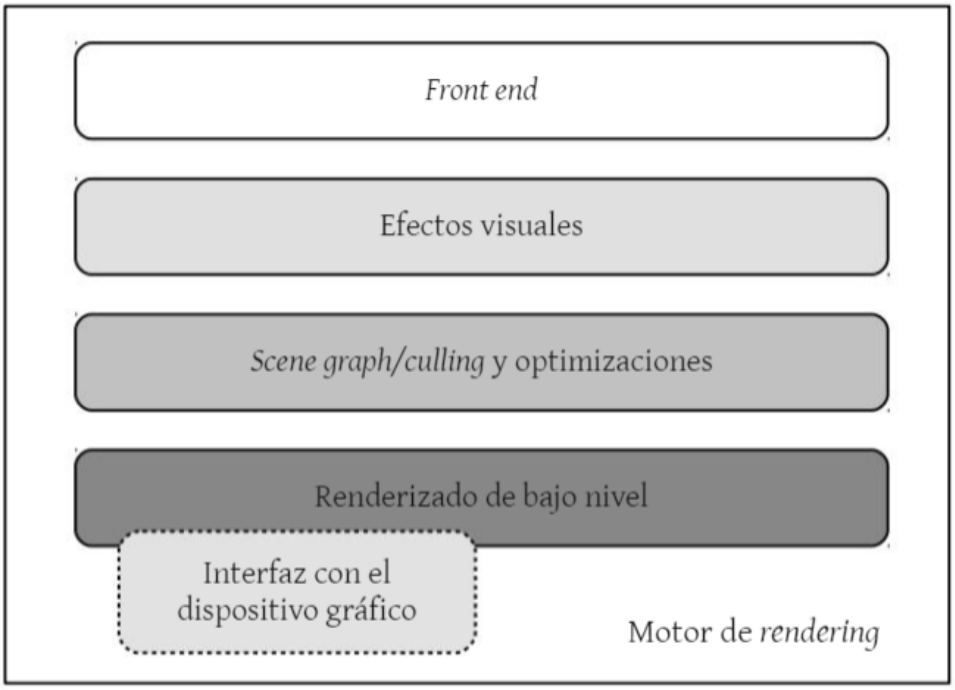
1.3.6 Motor de rendering

Dado que los componentes gráficos son una parte fundamental de cualquier juego y necesitan mejoras constantes, el motor de renderizado es una de las partes más complejas de

cualquier motor gráfico. El enfoque más utilizado para el diseño del motor de renderizado es una arquitectura en capas figura 9, al igual que sucede con la propia arquitectura del motor. (Cleto, 2013).

Figura 9

Motor de renderizado.



Nota: Arquitectura de un motor de renderizado, visión conceptual. Fuente: (Cleto, 2013)

La capa de **renderizado de bajo nivel** se encarga de la Funcionalidad relacionada con la representación gráfica de distintas entidades como cámaras, primitivas de render, materiales, texturas, etc. El objetivo principal de esta clase es renderizar varias primitivas geométricas lo más rápido posible, independientemente de las posibles optimizaciones o de qué partes de la escena son visibles desde el punto de vista de la cámara. Esta capa también es responsable de administrar las interacciones con las API de programación de gráficos como OpenGL o Direc3D para acceder a los diversos dispositivos gráficos disponibles.(Cleto, 2013)

La capa por encima de la capa de renderizado de bajo nivel se denomina gráfico de escena/rechazo y optimizador, y es responsable de seleccionar partes de la escena para enviarlas a la capa de renderizado. Esta selección u optimización puede ayudar a mejorar el rendimiento del motor de renderizado porque la cantidad de geometría enviada a las capas inferiores es limitada.

Aunque solo las primitivas dentro del campo de visión de la cámara (dentro del viewport) se dibujan en la capa de renderizado, se pueden aplicar otras optimizaciones para simplificar la complejidad de la reproducción. En juegos de considerable complejidad, este tipo de optimizaciones son importantes para lograr velocidades de cuadro aceptables.

Además de las capas que involucran la optimización, se encuentra la capa de efectos visuales, que admite varios efectos que luego se pueden integrar en los juegos desarrollados con el motor. Ejemplos representativos de módulos incluidos en esta clase son los módulos responsables de la gestión del sistema de partículas (humo, agua, etc.), el mapeo del entorno o las sombras dinámicas.

Finalmente, la capa de **Front-end** generalmente se enfoca en la funcionalidad relacionada con la superposición de contenido 2D en una escena 3D. Por ejemplo, es habitual utilizar algún tipo de mod que permita visualizar el menú del juego o una interfaz gráfica que permita conocer el estado del protagonista del videojuego (stocks, armas, herramientas, etc.). Esta clase también incluye componentes para reproducir video previamente grabado e integrar secuencias cinematográficas interactivas en el propio videojuego, el llamado IGC. (in-Game Cinematics).

1.3.7 Motor de físicas

La detección de colisiones en los videojuegos es esencial para el realismo. Sin un mecanismo de detección de colisiones, los objetos se cruzarían entre sí y no podrían interactuar

con ellos. Como punto de vista general, el sistema de detección de colisiones se encarga de realizar las siguientes tareas:

1. La **detección de colisiones**, su salida es un valor lógico que indica si hay una colisión.
2. La **determinación de la colisión**, responsable de calcular la intersección de la colisión.
3. La **respuesta a la colisión** Se utiliza para determinar la acción resultante que se genera por la colisión.

Debido a las limitaciones impuestas por la naturaleza del juego en tiempo real, los mecanismos de manejo de colisiones a menudo se aproximan para simplificar la complejidad de las colisiones sin sacrificar el rendimiento del motor. También es común usar un árbol BSP para representar el entorno y optimizar la detección de colisiones para los propios objetos.(Cleto, 2013).

Por otro lado, algunos juegos incluyen sistemas de simulación dinámica realistas o semi realistas. En la industria de los videojuegos, estos sistemas se denominan sistemas físicos y están directamente relacionados con los sistemas de gestión de colisiones. Actualmente, la mayoría de las empresas utilizan motores de colisión/física desarrollados por terceros, integrando estos kits de desarrollo en el propio motor. Los más conocidos en el mundo comercial son Havok, que representa el estándar de facto en la industria por su potencia y rendimiento, y PhysX desarrollado por NVIDIA e integrado en motores como Unreal Engine. (Cleto, 2013).

1.3.8 Networking

El módulo de networking es el encargado de informar del progreso del juego a los distintos agentes o usuarios que participan en el mismo mediante el envío de paquetes de información. Esta información se transmite mediante sockets. Para reducir el retraso en el modo

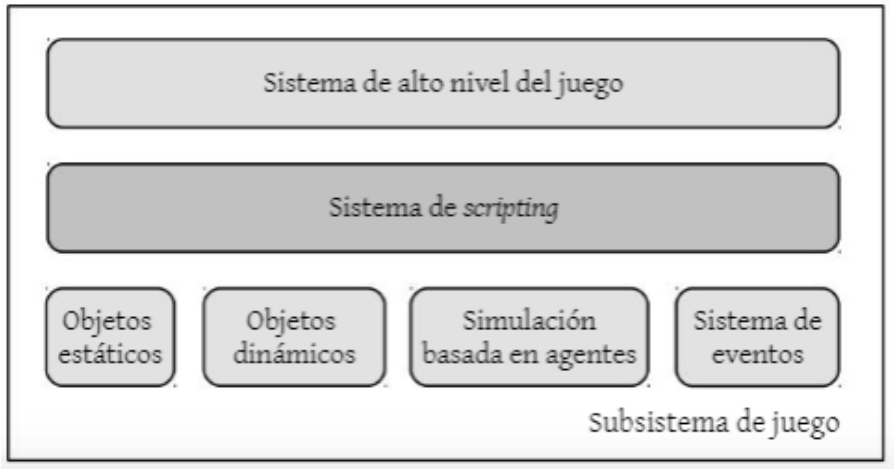
multijugador, especialmente a través de Internet, solo envía/recibe información relevante para el correcto funcionamiento del juego. (Cleto, 2013).

1.3.9 Subsistema de juego

El subsistema del juego, llamado Game Play en inglés, integra todos los módulos relacionados con el funcionamiento interno del juego, reuniendo las propiedades del mundo virtual y las propiedades de los diferentes personajes del juego. Además, permite definir las reglas que rigen el mundo virtual en el que se desarrolla el juego, como la necesidad de derrotar a los enemigos antes de enfrentarse a enemigos de mayor nivel, y este subsistema también permite definir la mecánica de los personajes. es su objetivo en el juego. En la figura 10 se ilustra la estructura. (Cleto, 2013).

Figura 10

Arquitectura del subsistema de juego.



Nota: visión conceptual de la arquitectura general del módulo subsistema de juego. Fuente: (Cleto, 2013)

Este subsistema también actúa como una capa de aislamiento entre las capas inferiores, como el renderizado y el funcionamiento del juego en sí. Uno de los principales objetivos de diseño era separar la lógica del juego de la implementación. Por esta razón, es común encontrar

algún tipo de sistema de scripting o lenguaje de alto nivel definido en esta clase. Por ejemplo, el comportamiento de los personajes que participan en el videojuego.

Conceptos como la gestión de clases del mundo del juego, relacionado con el subsistema del juego, ya sean elementos estáticos o dinámicos. Los tipos de objetos que pertenecen a este mundo a menudo se denominan modelos de objetos de juego. Este modelo proporciona una simulación en tiempo real de esta colección heterogénea, incluyendo:

- Cuerpos rígidos, como unas piedras o una silla.
- Elementos geométricos relativos a fondos estáticos, como por ejemplo edificios o carreteras.
- El personaje principal.
- Los personajes no controlados por el jugador (NPCs).
- Cámaras y luces virtuales.
- Armas, proyectiles, vehículos, etc.

El modelo de objetos de juego está estrechamente relacionado con el modelo de objetos de software y puede entenderse como una colección de propiedades, estrategias y transformaciones de lenguaje que utilizan una filosofía orientada a objetos para implementar código.

El sistema de eventos está integrado en la parte del subsistema del juego, y su tarea principal es facilitar la comunicación entre objetos, independientemente de su naturaleza y tipo. Un enfoque en el campo de los videojuegos involucra el uso de una arquitectura impulsada por eventos, donde las entidades primarias son eventos. Los eventos incluyen una estructura que contiene información relevante, por lo que la comunicación se guía precisamente por el contenido del evento y no por el remitente o el receptor del evento. Los objetos generalmente

implementan controladores de eventos para manejarlos y tomar medidas sobre ellos.(Cleto, 2013).

Por otra parte, el **sistema de scripting** permite modelar fácilmente la lógica del juego, como el comportamiento del enemigo o NPC, sin volver a compilar para verificar que dicho comportamiento sea correcto. El motor puede continuar funcionando bajo ciertas condiciones del script.(Cleto, 2013).

Finalmente, en la clase de subsistema del juego, puedes encontrar módulos que brindan funcionalidad relacionada con el procesamiento de IA, generalmente NPC, cuya funcionalidad generalmente se incluye en el juego principal. Una capa de software específica en lugar de integrarla en la herramienta nos permitiría especificar un comportamiento preestablecido sin programarlo. También puede encontrar módulos relacionados con aspectos de problemas de IA, como encontrar el camino óptimo entre dos puntos, llamado pathfinding, que implica el uso de algoritmos. Asimismo, la información privilegiada se puede utilizar para optimizar ciertas tareas, como la ubicación de entidades de interés, para acelerar cálculos como la detección de colisiones.

1.3.10 Audio

En el mundo desarrollado de los videojuegos, el componente gráfico siempre ha recibido más atención. Sin embargo, el sonido también es importante para que los usuarios se sumerjan en el juego. Es por eso por lo que las herramientas de audio son cada vez más importantes.

Con el desarrollo y la aparición de nuevos formatos de audio de alta definición y la proliferación de sistemas de cine en casa, han contribuido a este desarrollo, en el que la parte de audio ha cobrado cada vez más importancia.(Cleto, 2013).

Actualmente, como ocurre con otros componentes de la arquitectura del motor de juego, es común encontrar desarrollos que no están disponibles en un motor de juego, desarrollados por empresas ajenas a ese motor. El audio también debe cambiarse para juegos específicos para garantizar una buena experiencia desde una perspectiva auditiva.

1.4 MOTORES GRÁFICOS COMERCIALES

En la siguiente tabla 1 se demostrará especifican algunos de los motores gráficos comerciales más utilizados en el mundo, se destaca los tipos de videojuegos que pueden realizar 3D o 2D, el lenguaje con el que trabajan, el motor de físicas que utilizan, las plataformas donde se puede distribuir los juegos realizados en estos y sus tipos de licencias que son necesarias para trabajar con cada uno.

Tabla 1

Motores gráficos comerciales

Nombre	2D/3D	Lenguaje	Motor Fisico	Plataformas de Distribucion	Licencia
Unreal Engine 4	3D	C++	PhysX	Microsoft Windows, macOS, Linux, SteamOS, HTML5, iOS, Android, PlayStation 4, Nintendo Switch, Xbox One SteamVR/HTC Vive, Oculus Rift, PlayStation VR, Google Daydream, OSVR y Samsung Gear VR.	Comercial. Versión libre y pago de royalties sobre producto vendido al superar un umbral de facturación.
Unity 3D	3D	C#, Boo, JS	PhysX	Mac, Windows, Linux, iOS, Android, PS3, XB360, PSP, Wii, PSVita, Flash, Web	Comercial. Versión básica gratuita (desktop y web) y versiones PRO y plataformas móviles y consolas con un pago único sin royalties
Cry Engine	3D	Visual Script	Motor propio	Windows, XB360, PS3	Comercial. Es gratis para uso no comercial
Hero Engine	3D	Hero Script, C++ (PRO)	PhysX	Windows	Comercial. Precio por año.
Havok	3D	C++ y Havok Script	Havok Physics	Xbox 360®, PlayStation 3®, PC Games for Windows, PlayStation® Vita, Wii™, Wii U™, Windows 8, Android™, Apple Mac, and iOS	Comercial, Tienen una versión limitada gratuita para juegos que se vendan por debajo de 10$ solo para pc

Nota: cuadro comparativo motores gráficos comerciales más importantes. Fuente: (Moltó, Pellicer, & Pérez, 2012).

Para el presente proyecto se empleará el motor de Unreal Engine 4 debido a que presenta los siguientes beneficios.

- Es totalmente gratis en su versión más actualizada del mercado.
- Es actualizado por sus creadores y cuentan con los últimos avances tecnológicos.
- Nos ofrece el cien por ciento de la capacidad del motor.
- Se pueden desarrollar proyectos netamente comerciales sin la necesidad de adquirir alguna licencia especial.
- En el mercado se desarrolla más proyectos con Unreal Engine 4 que con sus competidores.
- Cuenta con un Marketplace donde se puede adquirir todo tipo de contenido para comenzar a probar en el motor.
- Cuenta con una documentación muy sólida, además de su comunidad que va en constante crecimiento.

1.5 UNREAL ENGINE 4

Es uno de los motores de videojuegos más conocidos y usados del momento, fue desarrollado en primera instancia por Tim Sweeney a los 21 años cuando diseñaba ZZT, el primer juego oficial de Epic Games. En 1991 los videojuegos eran creados por puñados de personas y cada juego nuevo se construía básicamente desde cero en un proceso que requería el trabajo preciso de ingenieros, programadores y de los diseñadores, Sweeney baso el proceso de creación del videojuego ZZT en un estilo de programación orientada a objetos y diseño ZZT-OOP para un fácil control sobre los objetos del juego. Al convertir en una parte central el código del juego, permitió modificaciones significativas al usuario más allá de los simples editores de mapas de la época. Este enfoque estableció el marco conceptual para la idea de un motor de juego.(Thomsen, 2012)

En 1998 de la mano de Epic Games se lanzó a primera generación de Unreal Engine desde entonces la compañía ha continuado desarrollando el motor y se han creado 4 versiones, con funcionalidad gradualmente expandida, brindando una mayor capacidad para procesamiento de datos, renderizado, mejoras de texturizado e integración de nuevas funciones.

Unreal Engine ahora es un entorno de desarrollo que incluye todas las herramientas que se necesita para construir un juego o una simulación, como un editor de video, un estudio de sonido, un renderizado de imágenes o un animador, y también ha encontrado nichos en diferentes campos como la arquitectura, la ingeniería, la medicina, la realidad virtual, etc.(S.L., 2020).

En 2015 esta herramienta salió de manera totalmente gratuita al público en general, con el único requisito impuesto por Epic Games estipula que debe pagarse el 5% de las ganancias brutas superiores a $ 3,000 trimestralmente. Su facilidad de uso y el hecho de que no tiene que pagar ninguna licencia al desarrollar un producto lo hacen ideal tanto para estudios independientes como para estudios más grandes y antiguos. Además, cuenta con un Marketplace, una tienda donde los desarrolladores pueden comprar contenido, lo cual es útil para la creación rápida de prototipos. (S.L., 2020).

Unreal Engine Marketplace es una plataforma de comercio electrónico que se inauguró en el año 2014 donde diseñadores y artistas gráficos pueden vender sus creaciones y recursos para que el comprador los utilice en sus proyectos con el motor gráfico. El sitio cuenta con aproximadamente 5000 productos de cerca de 1500 autores los cuales se llevan el 88% de las ganancias y Epic un 12%, entre los productos que se pueden encontrar están elementos de sonido, gráficos, blueprints, assets, etc. Se estima que el sitio tiene un aproximado de 8 millones de descargas de assets siendo solo un millón de estas gratuitas. Con la ayuda del crecimiento

del Marketplace y el éxito de Fortnite Epic ópera como un enorme comercio digital. (pinjed, 13 julio 2018).

Unreal Engine 4 cuenta con numerosas funciones que permiten desarrollar juegos muy realistas consiguiendo una inmersión en el videojuego. El motor realiza un renderizado basado en la física, creando mundos y escenarios con iluminación y sombras fotorrealistas.

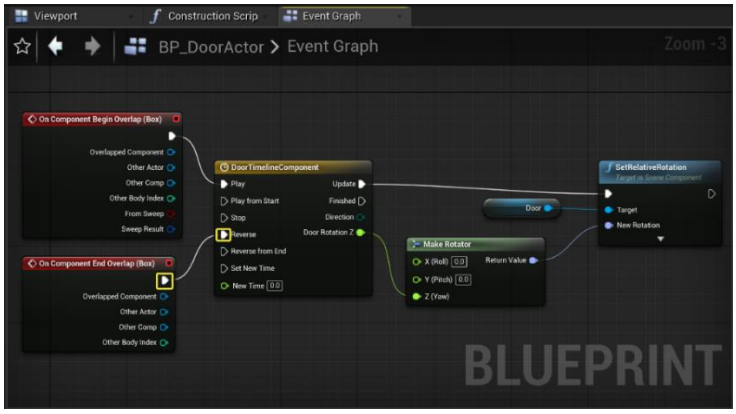
Para conseguir la creación de estos mundos y escenarios de juego, Unreal nos ofrece dos maneras de desarrollo, mediante visual scripting Blueprint, o programación con el lenguaje de alto nivel C++. Durante el desarrollo se trabaja con actores, objetos, clases, etc. Además, se utiliza el paradigma de programación orientado a objetos.

A continuación, se describe de forma breve algunos de los conceptos fundamentales con los que trabaja Unreal al momento de desarrollar videojuegos y que serán necesarios para comprender el apartado de implementación y desarrollo.

- Paradigma de programación orientado a objetos se define como un estilo de programación, se basa en los conceptos de clases y objetos. Este tipo de programación se utiliza para estructurar programas de software en piezas de código simples y reutilizables.
- **C++** es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup.
- **Blueprint Visual Scripting** es un sistema de secuencias de comandos que utiliza una interfaz visual basada en nodos para crear elementos del juego desde editor de Unreal.
- Se denomina coloquial mente solo como Blueprint, se lo pude observar en la figura 11.

Figura 11

Blueprint visual scripting.

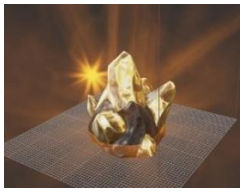


Nota: Editor de Blueprint Unreal Engine. Fuente: (Engine, 2020)

- **Clases**, en Unreal definen los comportamientos y propiedades de un Actor u Objeto en particular. Las clases se pueden crear en código C++ o en Blueprints.
- **Objetos**, son la clase más básica en Unreal Engine, en otras palabras, actúan como bloques de construcción y contienen muchas de las funciones esenciales para sus activos. Casi todo en Unreal Engine obtiene alguna funcionalidad de un objeto.
- **Actor**, es cualquier objeto que se puede colocar en mapa del juego(nivel), como una cámara, una malla estática o la posición inicial del jugador. Los actores admiten transformaciones 3D como traslación, rotación y escalado. Se pueden generar y destruir a través del código del juego (C++ o Blueprints). En la figura 12 se muestra un ejemplo de actor.

Figura 12

Actor.

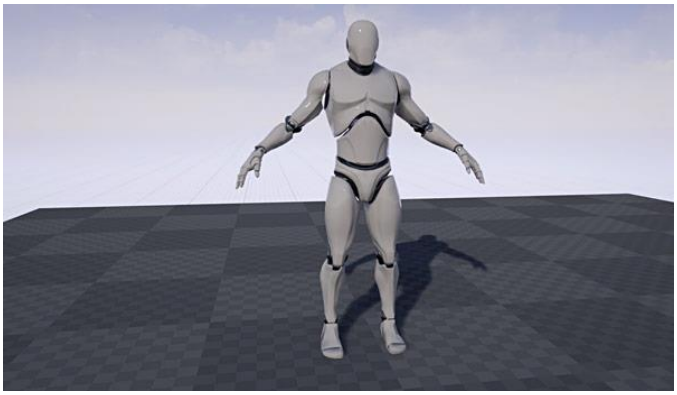


Nota: Ejemplo de actor en Unreal Engine. Elaborado por: Los autores.

- **Componente**, es una pieza de funcionalidad que se puede agregar a un Actor. Cuando agrega un Componente a un Actor, el Actor puede usar la funcionalidad que proporciona el Componente. Por ejemplo:
    - Un componente de luz hará que su actor emita luz.
    - Un componente de movimiento giratorio hará que tu actor gire.
    - Un componente de audio le dará a su actor la capacidad de reproducir sonidos.
- **Character**, es una subclase de un actor que está destinado a ser utilizado como personaje de jugador.

Figura 13

Personaje.



Nota: Personaje por defecto del motor. Elaborado por: Los autores.

- **Colisión**, es una forma programática de evitar que los objetos se superpongan durante una simulación física, dando la ilusión de solidez en un objeto.
- **Nivel**, es un área de juego que es definida por el desarrollador. Los niveles contienen todo lo que un jugador puede ver e interactuar, como geometría, peones y actores.

- **Widget**, son una serie de funciones prefabricadas que se pueden usar para construir su interfaz (cosas como botones, casillas de verificación, controles deslizantes, barras de progreso, etc.).

En la actualidad Unreal acompaña a la industria de los videojuegos facilitando el desarrollo, proporciona un gran rendimiento y potencial a los proyectos que se realizan en él. Se utiliza en grandes proyectos donde se requiera obtener gran calidad. Varias de las industrias que apestan por el de detallan en la tabla 2.

Tabla 2

Industrias que utilizan Unreal Engine

Nombre	**Uso del motor Unreal Engine 4**
NASA	La NASA usa Unreal Engine 4 para ingeniería y para ayudar a entrenar a sus futuros astronautas. De esta forma, pueden conseguir una formación más larga y económica para sus especialistas. Actualmente están siendo entrenados en ejercicios de mantenimiento de la ISS. (S.L., 2020) La NASA se asoció con Fusión Media, el Laboratorio de Sistemas Espaciales del MIT y el desarrollador Irrational Games para crear la "Experiencia Mars 2030". Cualquier persona con un sistema de realidad virtual puede visitar Marte y recorrer su paisaje rojo.(Patiño, 2016)
McLaren	Las aplicaciones de Unreal Engine 4 también se han abierto camino en el espacio automotriz. La marca británica McLaren está utilizando Unreal Engine 4 para diseñar algunos de sus autos. La alianza se anunció en la Game Developers Conference 2016. El gerente de diseño de McLaren, Mark Roberts, dijo: "McLaren ha enviado datos capturados de todos sus autos, junto con muestras de pintura reales, muestras de materiales de pintura y tela, y audio a Epic. Grabando cosas como el ruido del motor".(S.L., 2020)
BMW	BMW es otro fabricante de automóviles que utiliza Unreal Engine 4 para sumergirse por completo en la realidad virtual. La marca alemana eligió el motor gráfico de Epic Games para diseñar y modelar rápidamente el interior de sus vehículos. Con este enfoque, permiten a los clientes elegir y personalizar instantáneamente sus futuros automóviles.(S.L., 2020)

Nota: Ejemplos de industrias que apuestan mejoras para la empresa utilizando Unreal Engine. Elaborado por: los autores

2. CAPITULO 2: MARCO METODOLÓGICO.

2.1 Diseño del videojuego

2.1.1 Términos Generales del juego

Concepto del juego. El videojuego comienza con personaje principal acostado detrás de una fogata el cual se encuentra encerrado en una celda, no recuerda nada de lo que paso, por que busca salir de ese lugar, en el camino a su escape se va topando con varias habitaciones en un tipo de mazmorra, las cuales tienen instrucciones de cómo se pude abrir la celda.

La única alternativa que cuenta el personaje es ir descifrando la manera de abrir la puerta y avanzar hasta la última habitación donde para escapar deberá recolectar 3 llaves con las cuales están escondidas en las habitaciones posteriores.

Genero. Es un juego de género "Plataformas" ya que el jugador controla a un personaje el cual debe avanzar en la escena evitando los obstáculos físicos, ya sea saltando, trepando o agachado. Además de poder moverse como saltar o correr, los personajes de plataformas pueden moverse libremente por el escenario a su gusto y no en línea recta ya que se trata de un juego en 3D.(Belli & Raventós, 2008).

Apariencia del juego. El juego busca introducir a los jugadores en un mundo de temática medieval con la libertad de poder moverse en todo el mapa además de poder sujetar diversos objetos que le serán de ayuda para poder abrir las puestas.

Por otro lado, se presentan sonidos de acuerdo con los distintos objetos dentro del juego también el sonido de fondo que da una atmosfera de misterio que supone estar solo en una mazmorra desconocida.

2.1.2 Mecánicas del juego

Objetivos del juego. El juego cuenta con diferentes retos en los dos niveles. Para poder pasar de nivel hay que completar todas las misiones disponibles en cada nivel, las cuales son:

Primer nivel

- Obtener las 3 llaves que aparecerán en distintos puntos del mapa, siempre que inicie partida las llaves se moverán de manera aleatoria a los puntos seleccionados del mapa. Este objetivo se deberá llevar a cabo en todo el trascurso del primer nivel.
- Colocar un objeto cualquiera encima de una placa de presión que abrirá la segunda puerta. Este objetivo se llevará a cabo en el trascurso del segundo subnivel.
- Ir agregando objetos dentro de un féretro hasta conseguir que la tercera puerta se abra. La puerta indicara el número de objetos que se deben colocar dentro del féretro, este número varia en cada intento de partida siempre será de cinco a doce objetos. Este objetivo se llevará a cabo en el trascurso del segundo y tercer subnivel.

Segundo nivel

- Buscar el objeto final dentro del primer y segundo nivel el cual será necesario para abrir la última puerta.

Acciones del personaje. El personaje en la fase de primera persona realizara las siguientes acciones

- Moverse
  - Hacia adelante
  - Hacia atrás
  - Hacia la izquierda
  - Hacia la derecha
- Girarse
  - Girarse 360 en el eje Z
  - Girarse 90 en el eje X
- Saltar

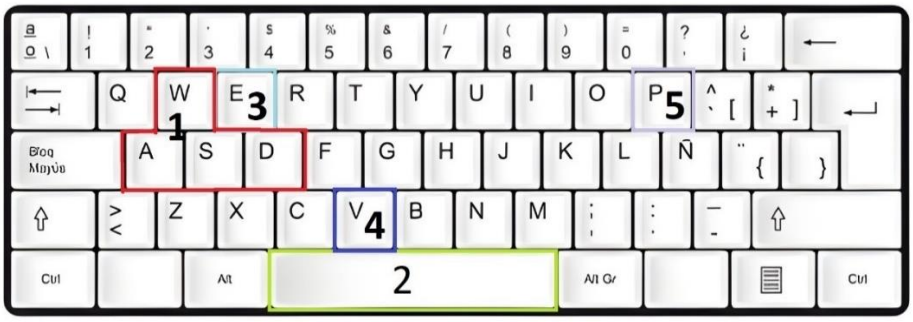
- Hacia adelante
- Hacia atrás
- Hacia la izquierda
- Hacia la derecha

- Rodar
  - Hacia adelante
- Tomar objetos

Controles del juego. Se podrá controlar las acciones del personaje dentro del videojuego usando el teclado como lo muestra la figura 14 y ratón.

Figura 14

Controles del juego.



Nota: Teclas asignadas para controlar el personaje. Elaborado por: los autores.

1. Movimiento del jugador
   - Tecla W movimiento hacia adelante.
   - Tecla A movimiento hacia la izquierda.
   - Tecla S movimiento hacia atrás.
   - Tecla D movimiento hacia la derecha
2. Saltar en la dirección que se mueva el jugador
3. Agarrar y soltar objetos

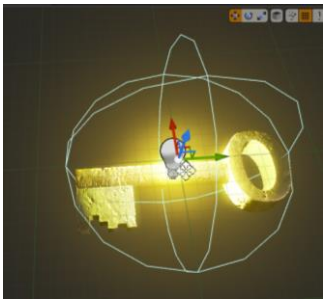
4. Rodar
5. Abrir y cerrar el panel de juego

Interacción con el jugador

- **Llaves** (primer nivel): Hay un total de 3 llaves las cuales son necesarias para abrir la puerta hacia el segundo nivel. Figura 15.

Figura 15

Llaves.



Nota: Objeto que interactuara con el jugador durante la partida. Elaborado por: Los autores.

- **Prensa** (primer nivel): Se necesita colocar un objeto para conseguir aplastarla. Figura 16.

Figura 16

Prensa.

Nota: Objeto que interactuara con el jugador durante la partida. Elaborado por: Los autores.

- **Esqueletos** (primer nivel): Objetos que se encuentran en el mapa los cuales pueden ser movidos de un lado a otro. Figura 17.

Figura 17

Esqueletos.

Nota: Objeto que interactuara con el jugador durante la partida. Elaborado por: Los autores.

- **Gárgola** (primer nivel): Objeto con el cual se abrirá la puerta final. Figura 18.

Figura 18

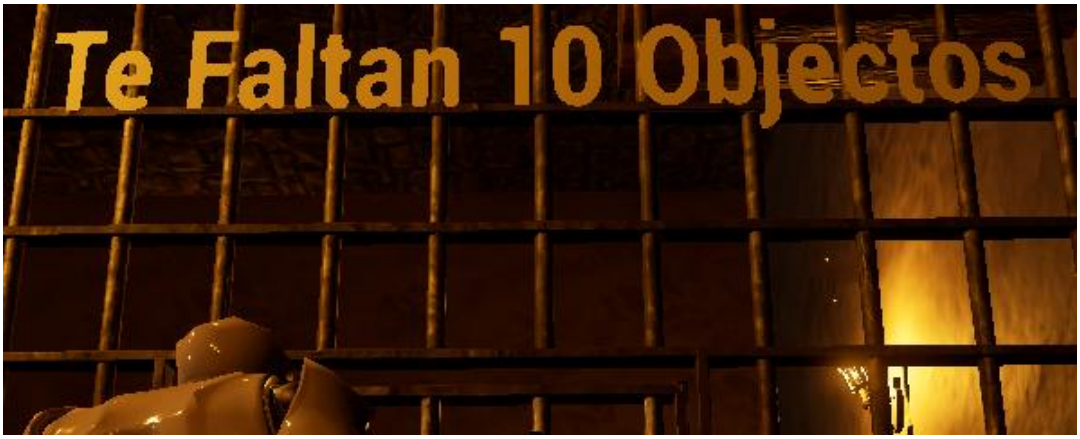
Gárgola.

Nota: Objeto que interactuara con el jugador durante la partida. Elaborado por: Los autores.

- **Mensajes de ayuda** (primer nivel): Texto escrito en la parte superior de las puertas que indican que se debe hacer para abrir las puertas como se muestra en la figura 19.

Figura 19

Mensajes.



Nota: Texto que se mostrara encima de las puertas indicando el objetivo para abrirla. Elaborado por: Los autores.

- **Puertas** (primer nivel): Hay un total de tres puertas cuya función es impedir el paso del jugador hasta que se realice la misión que solicite cada puerta. Las puertas se muestran en la figura 20.

Figura 20

Puerta.

Nota: Objeto con el cual interactuara el jugador durante la partida. Elaborado por: Los autores.

- **Gárgola** (Segundo nivel): Objeto que da ambientación en la habitación, siempre estará mirando a la posición a donde se mueva el jugador. Figura 21.

Figura 21

*Gárgola de ambientación***.**

Nota: Objeto que da un ambiente más acore al mundo en que se desarrolla el juego, este seguirá con la mirada al jugador a donde se mueva. Elaborado por: Los autores.

- **Pueta de piedra** (Segundo nivel): La puerta se alzará una vez se coloque el objeto solicitado por el juego. Figura 22.

Figura 22

Puerta de piedra.

Nota: Puerta final del juego. Elaborado por: Los autores.

Otros elementos

- Antorchas: Objeto emite una luz tenue, se usa para iluminar el mapa, se encuentran en los dos niveles. El jugador no puede interactuar con ellas. Figura 23.

Figura 23

Antorchas.

Nota: Objeto que proporciona iluminación al nivel. Elaborado por: Los autores.

- Rejas: Tienen la función principal de no dejar pasar al jugador por el mapa, además dan la forma a los subniveles presentes en el primer nivel. Figura 24.

Figura 24

Rejas.

Nota: Objeto que limitara el área de los niveles. Elaborado por: Los autores.

- Columnas: Sirven como soporte y acabado estético para que el jugador tenga una mejor jugabilidad. Figura 25.

Figura 25

Columnas.

Nota: Objetos que proporcionan jugabilidad dentro del nivel. Elaborado por: Los autores.

- Jaulas: Su principal función es dar una mejor ambientación dentro del mapa. El jugador no puede interactuar con ellas. Figura 26.

Figura 26

Jaulas.

Nota: Objetos que dan un acabado estético al nivel. Elaborado por: Los autores.

2.1.3 Menús e Interfaces

La interfaz es conocida como HUD esta nos mostrara la información por pantalla durante la partida. Puede mostrarse siempre o mediante determinados eventos.

Menú principal. El menú principal se abre al ejecutar el juego donde se mostrará al personaje principal del juego encerrado en una jaula con una hoguera iluminando la habitación y contará con las siguientes opciones:

- **Jugar**: Esta opción iniciará una nueva partida.
- **Opciones**: Esta opción abrirá una nueva interfaz donde se podrá configurar algunas características del juego como son:

  Resolución de pantalla

  - Small
  - HD
  - Full HD
  - UHD

  Anti-Aliasing

  - Low
  - Med
  - High
  - Ultra

  Calidad de texturas

  - Low
  - Med
  - High
  - Ultra

  Distancia de dibujado

- Cerca
- Medio
- Lejos

Calidad del sombreado

- Low
- Med
- High
- Ultra

- **Como jugar**: Esta opción abrirá una nueva interfaz donde se mostrarán una imagen con los controles del juego, adicional a eso nos mostrara consejos para distintas situaciones que puedan suceder al jugar. Esta opción tiene como objetivo que el jugador aprenda los controles del juego.
- **Salir**: Esta opción termina la ejecución del juego por lo cual se cierra inmediatamente.

Menú de juego. Este menú se podrá visualizar dentro del juego, una vez se seleccione la opción jugar en el menú principal y de comienzo la partida. Para tener acceso a él se deberá presionar la tecla P lo que detendrá el juego y mostrará en el primer plano el menú con las siguientes opciones.

- **Reiniciar**: Esta opción comienza una nueva partida de manera inmediata sin salir al menú principal.
- **Fin del juego**: Esta opción termina la partida actual y nos regresa al menú principal.
- **Salir del programa**: Esta opción termina la ejecución del juego por lo cual se cierra inmediatamente.

Interfaz durante la partida. Esta interfaz se mostrará en cuanto se inicie la partida estará colocada en la parte superior derecha de la pantalla su principal función será mostrar el número de llaves que hayamos encontrado y tomado.

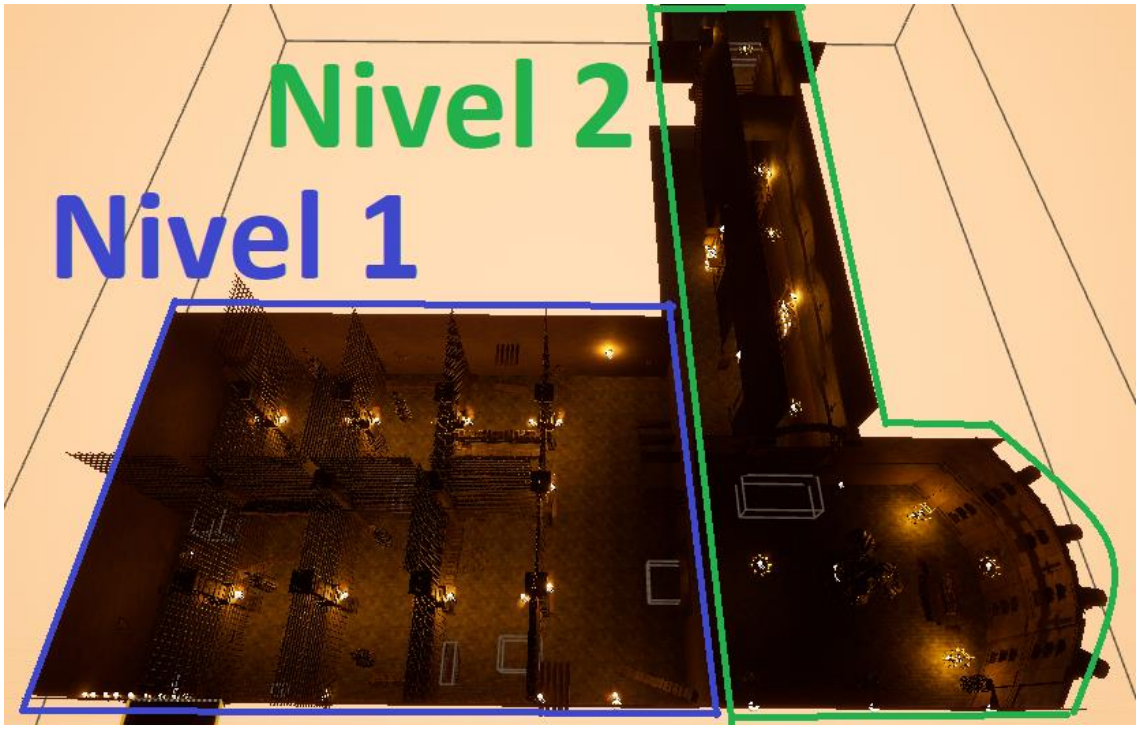
Interfaz fin del juego. Esta interfaz se mostrará en cuanto el jugador llegue a la parte final del juego, mostrara en la pantalla información de los creadores del juego y algunos créditos del material utilizado como son los sonidos e iconos al finalizar se retornará al menú principal. Mientras la interfaz este activa, el jugador no podrá moverse.

2.1.4 Mapa de juego

El mundo en el que se desarrollará el juego será un mapa dividido en dos niveles como se muestra en la figura 27. El cual contara con diversos objetos los cuales darán una ambientación de una mazmorra.

Figura 27

Mapa general del juego.

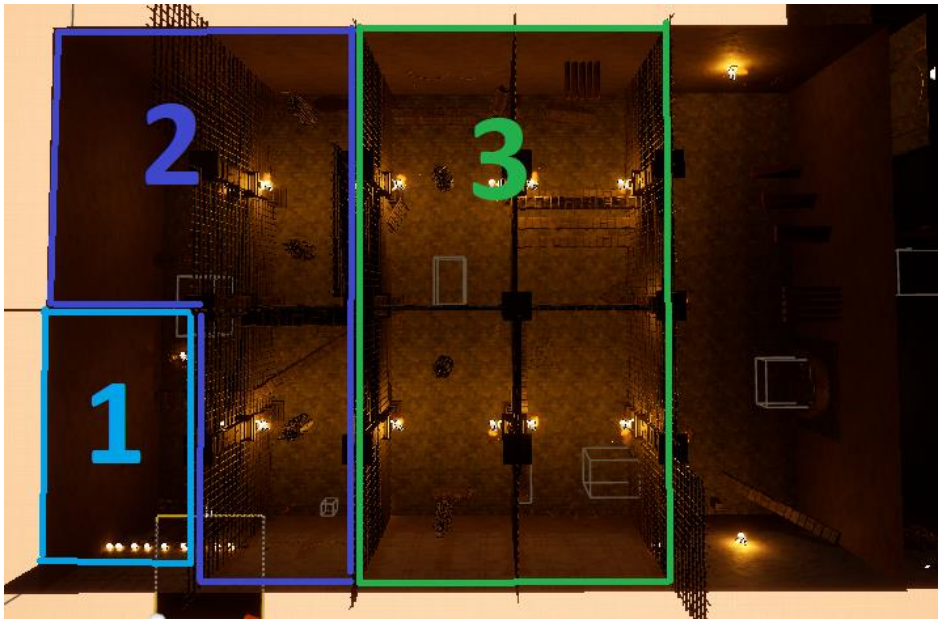


Nota: Visión general de los dos niveles que presenta el juego. Elaborado por: Los autores.

Primer nivel del mapa. El primer nivel del mapa cuenta con 3 subniveles. Cada subnivel está separado por la puerta que se debe abrir, la figura 28 muestra como esta subdividido el primer nivel.

Figura 28

Primer nivel.



Nota: vita de los tres subniveles del primer nivel. Elaborado por: Los autores.

1. En el primer subnivel la puerta se abrirá de manera automática, el jugador solo se acercará a la puerta sin realizar ninguna otra acción.
2. El segundo subnivel se encontrará la puerta 2, la cual se abrirá realizando la acción definida en los objetivos del juego. Para poder pasar al siguiente nivel se tendrá que realizar la mecánica establecida por el juego.
3. El tercer subnivel contará con la puerta 3, en este subnivel se deberá utilizar los recursos de los anteriores subniveles para poder alcanzar el objetivo.

Segundo nivel del mapa. El segundo nivel del mapa contará con una sola habitación la cual contendrá una gárgola como se muestra en la figura 29. En la figura 30 se aprecia todo el segundo nivel.

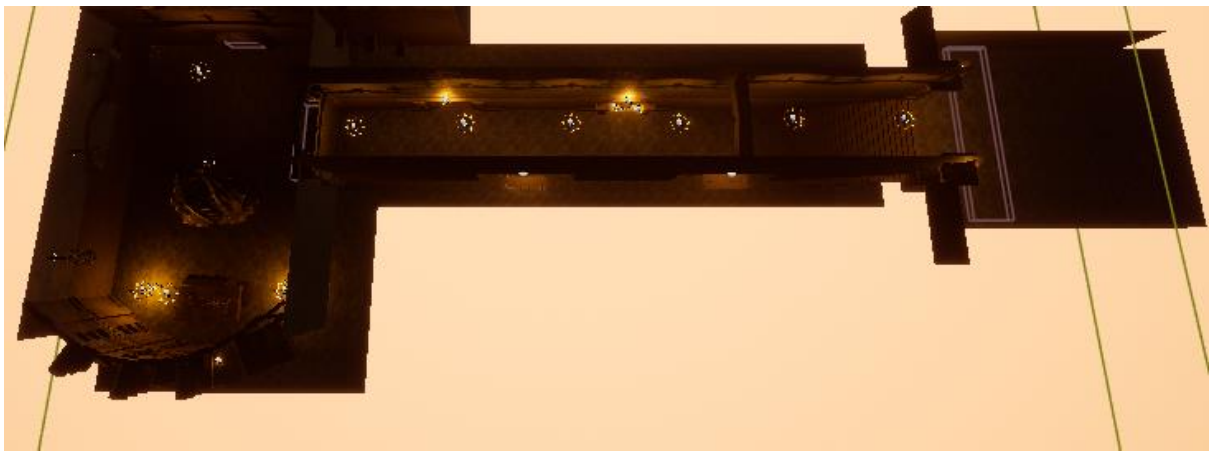
Figura 29

Gárgola

Nota: Gárgola que seguirá con la mirada al personaje. Elaborado por: Los autores

Figura 30

Segundo nivel.



Nota: Vista general del segundo nivel. Elaborado por: Los autores.

La última fase del juego consiste en recorrer un pasadizo que llevará a la salida definitiva de la mazmorra como se muestra en la figura 31. Al culminar se mostrará algunos créditos e información de los autores del juego durante un lapso de menos de 10 segundos. Una vez concluido se regresará de manera automática al menú principal.

Figura 31

Pasadizo final.

Nota: Pasadizo final para terminar el juego. Elaborado por: Los autores.

2.2 Desarrollo E Implementación

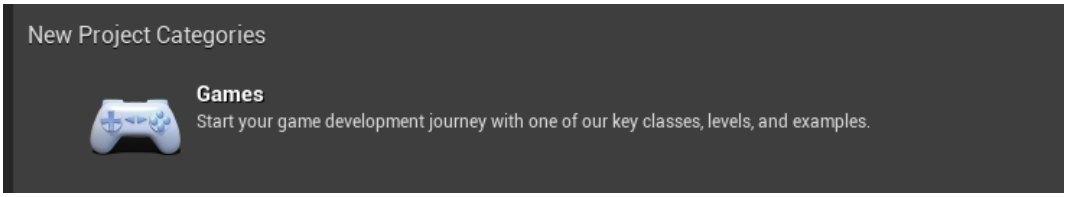
En esta sección se abarcará como se desarrolló e implemento lo mencionados en la sección 2.1 mediante el uso de Blueprints y programación en C++ de Unreal Engine 4.

2.2.1. Creación del juego.

Selección del proyecto Unreal Engine 4. La creación de proyectos en Unreal Engine es sencilla figura 32, lo que debemos tomar en cuenta es que clase de proyecto se desea realizar. El motor ofrece plantillas por defecto las cuales crearan un proyecto ya sea completamente vacío o con implementaciones iniciales ya definidas como objetos ya cargados, esto dependerá de la plantilla seleccionada.

***Figura** 32*

Creación de proyecto.



Nota: Creación de nuevo proyecto en Unreal Engine 4. Elaborado por. Los autores.

Además, se podrá visualizar el aspecto de que tendrá el proyecto en la pantalla de la derecha como se muestra en la figura 33.

Figura 33

Plantillas.

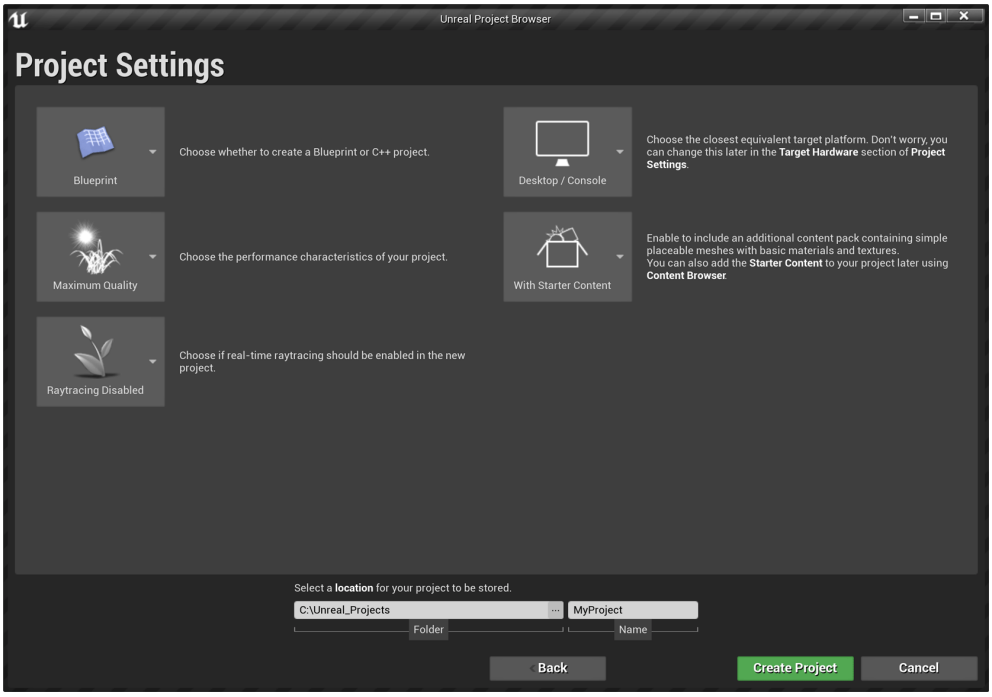
Nota: Selección de plantilla para el proyecto. Elaborado por: Los autores.

Para el presente proyecto se hará uso del template Third Person, con ello ya vendrá implementado nuestra clase en C++ ThirdPersonCharacter la cual contendrá toda la programación para el control del personaje.

Configuración del proyecto. En la página Configuración del proyecto figura 34, puede elegir el nivel de calidad/rendimiento, la plataforma de destino, si desea incluir contenido de inicio y más.

Figura 34

Configuraciones del proyecto.

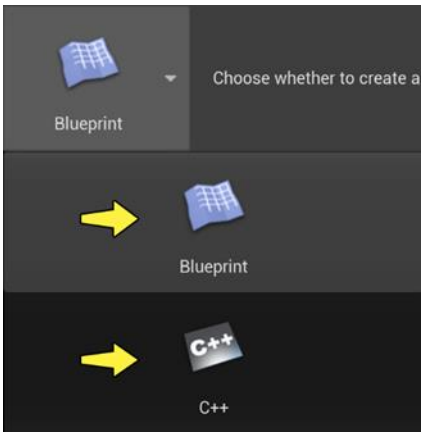


Nota: Pantalla para configuración del proyecto. Elaborado por: Los autores.

- **Blueprint**: En esta sección se seleccionará el modo de trabajo si en Blueprint o C++ como se muestra en figura 35.

Figura 35

Configuración Blueprint.



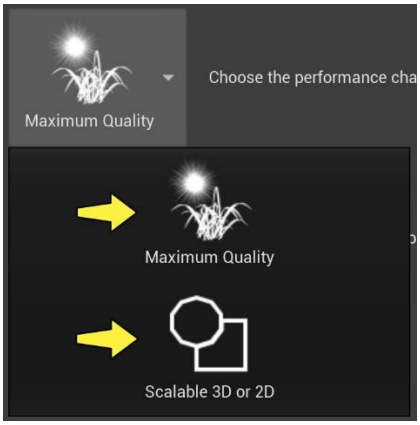
Nota: Opciones de configuración para el proyecto. Elaborado por: Los autores.

Seleccionaremos C++ ya que se quiere compilar el proyecto programando con C++ en Visual Studio.

- **Calidad máxima**: En esta sección se seleccionará si está desarrollando su proyecto para verlo en una computadora o consola de juegos figura 36.

Figura 36

Configuración calidad máxima.



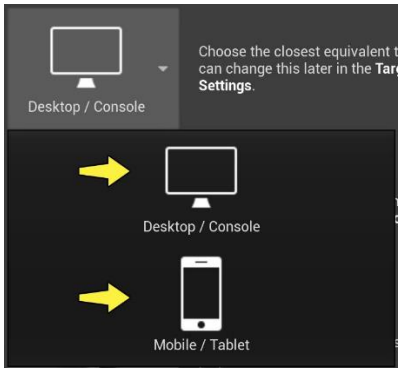
Nota: Opciones de configuración para el proyecto. Elaborado por: Los autores.

Seleccionaremos Escalable 3D o 2D ya que haremos uso de una computadora.

- **Escritorio/Consola**: Seleccionamos la opción de escritorio ya que el juego está destinado para la plataforma Windows y Linux, se muestra la opción en la figura 37.

Figura 37

Configuración de escritorio o consola.

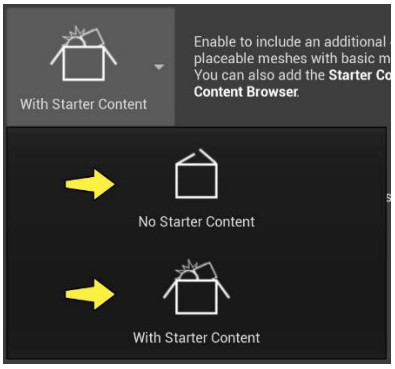


Nota: Opciones de configuración para el proyecto. Elaborado por: Los autores.

- **Contenido de inicio**: Seleccionamos no starter content figura 38, ya que comenzamos sin ningún recurso básico.

Figura 38

Configuración contenida de inicio.

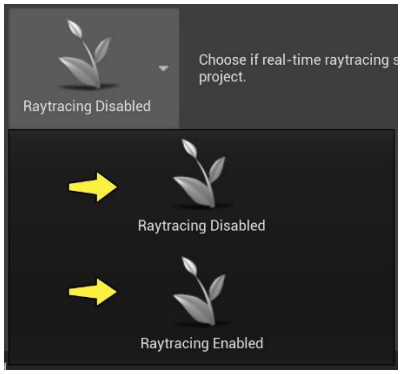


Nota: Opciones de configuración para el proyecto. Elaborado por: Los autores.

- **Raytracing deshabilitado:** Seleccionamos disable como se muestra en la figura 39.

Figura 39

Configuración raytracing.

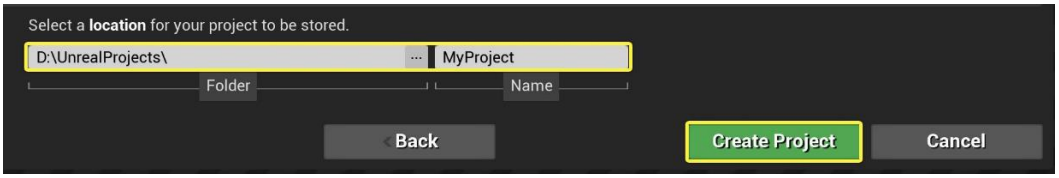


Nota: Opciones de configuración para el proyecto. Elaborado por: Los autores.

Finalmente, se seleccionará dónde almacenaremos el proyecto y asignaremos un nombre como se muestra en la figura 40, clic en crear proyecto para finalizar.

Figura 40

Guardar el proyecto.



Nota: Se seleccionará una ubicación para guardar el proyecto. Elaborado por: los autores.

Agregación de elementos del Marketplace al proyecto. Para la creación del proyecto se seleccionó del Marketplace elementos que contaban con las características requeridas en la especificación del diseño del juego inicial, se descargó todos los elementos con el fin de integrar al proyecto recién creado.

Para la implementación de los assets es necesario dirigirse a la pantalla principal de Epic Games en el apartado de biblioteca, ubicamos los assets Medieval Dungeon y posteriormente seleccionamos el proyecto al cual queremos añadirlo como lo muestra la figura 41. Finalmente, los assets empezarán a cargar dentro de nuestro proyecto este proceso se muestra en la figura 42.

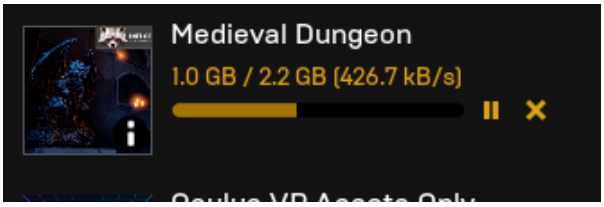
Figura 41

Selección del proyecto.

Nota: Selección del proyecto donde se agregarán los assets. Elaborado por: Los autores.

Figura 42

Assets Medieval Dungeon.

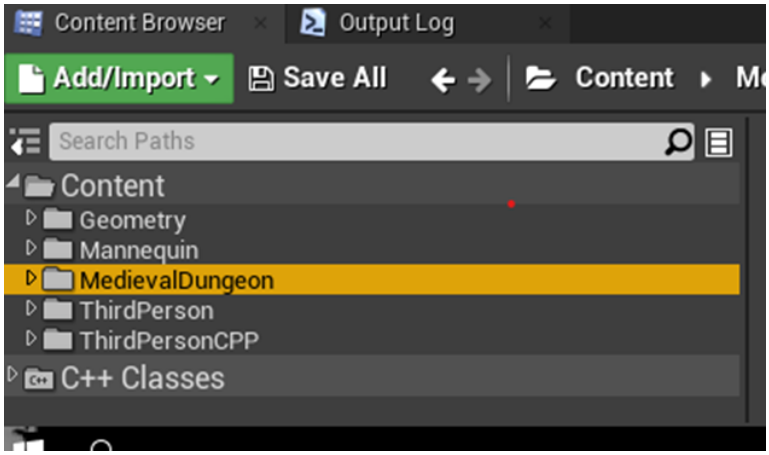


Nota: Carga de los assets al nuevo proyecto. Creado por: Los autores.

Creación del nivel. Al agregar los elementos del Marketplace al proyecto se crea una carpeta con el mismo nombre donde estarán todas las texturas como lo muestra la figura 43.

Figura 43

Carpeta de texturas.

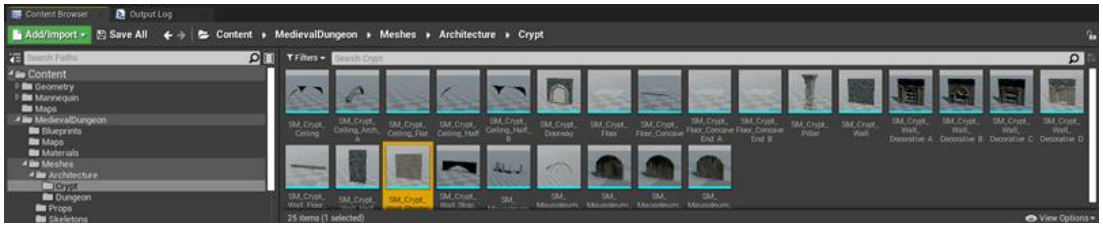


Nota: Lugar donde se almacenan los nuevos assets cargados. Elaborado por: Los autores.

En el editor de Unreal en la parte inferior se muestran divididos por carpetas todos los assets implementados en el proyecto como lo muestra la figura 44, a partir de este momento podemos hacer uso de cualquier asset que necesitemos para la creación del nivel como se muestra en la figura 45.

Figura 44

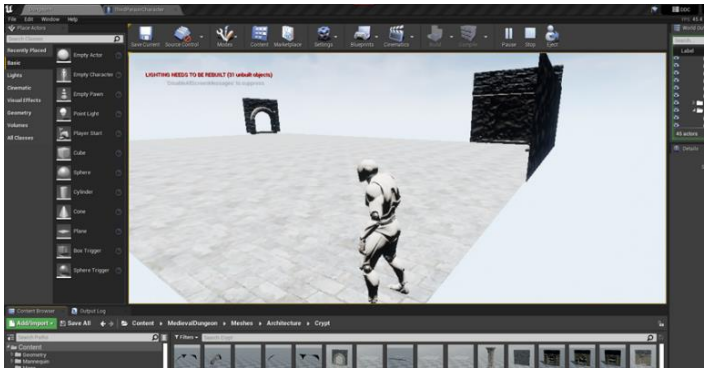
Paleta de Shaders.



Nota: Paleta de shaders según la carpeta seleccionada. Elaborado por: Los autores.

Figura 45

Implementación del diseño del mapa.

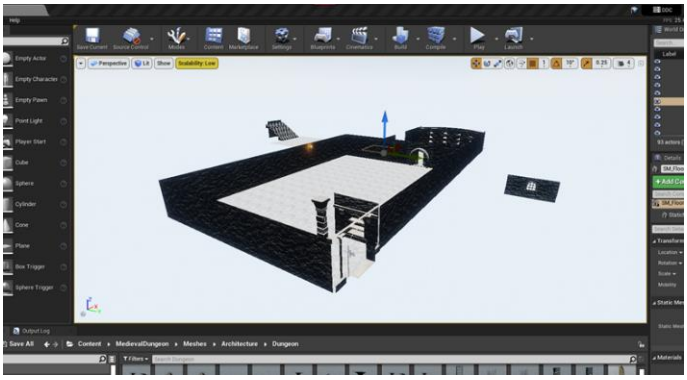


Nota: Implementación del diseño del mapa. Creado por: Los autores.

Se comienza realizando la estructura externa que corresponde al primer nivel del juego como se muestra en la figura 46, posteriormente comienza con la estructura de los tres subniveles correspondientes como se muestra en la figura 47.

Figura 46

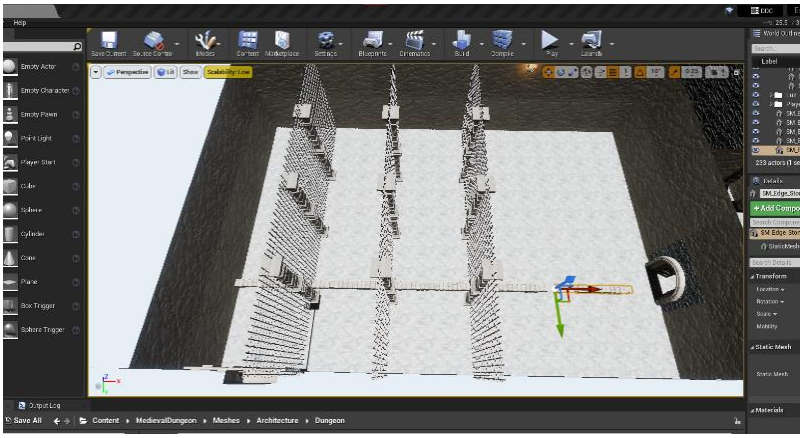
Estructura externa del mapa nivel 1.



Nota: Vista general de la estructura externa del mapa nivel 1. Elaborado por: Los Autores.

Figura 47

Estructura de los subniveles.

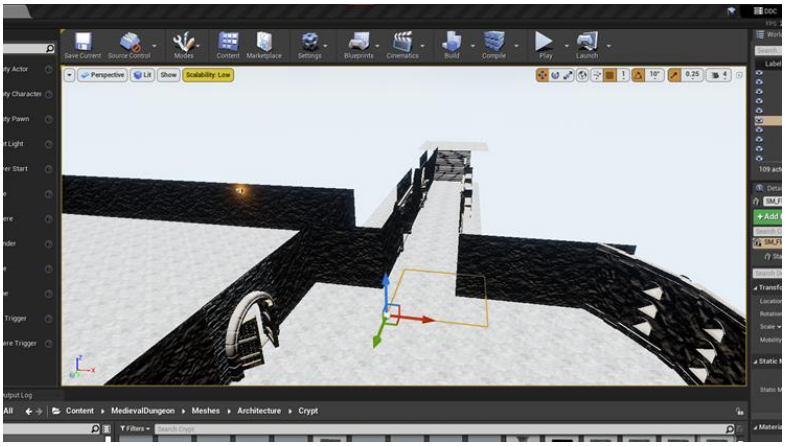


Nota: Vista general de la estructura para los 3 Subniveles del primer nivel. Elaborado por: Los autores.

Para la estructura del segundo nivel se comienza con el área externa como se aprecia en la figura 48, además se define dónde estará ubicada la última puerta que lleva al pasadizo para finalizar el juego.

Figura 48

Estructura externa del mapa nivel 2.

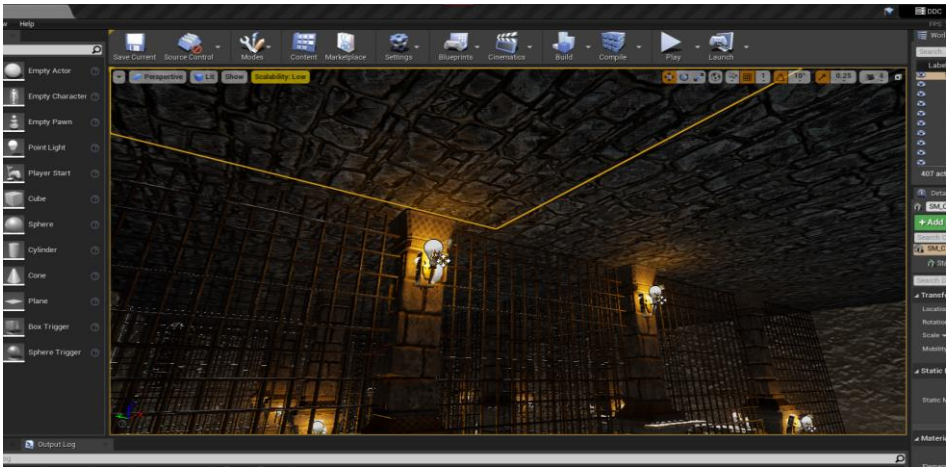


Nota: Vista general de la estructura externa del mapa nivel 2. Elaborado por: Los autores.

Una vez ya establecido la estructura de todo nuestro mapa se comienza con la agregación de diferentes elementos que dan ambientación necesaria para que el juego se parezca a una mazmorra. Se agregan detalles como antorchas, esqueletos, baúles, féretros, estatuas, etc. Finalmente se coloca el techo como se muestra en la figura 49 dejando el mapa del juego culminado.

Figura 49

Techo del mapa.

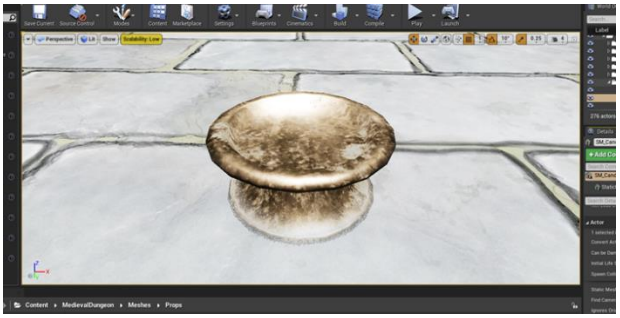


Nota: Colocación del techo del mapa. Elaborado por: Los autores.

Los objetos que con los cuales interactuara el jugador, serán configurados para que puedan ser agarrados, esto se realizará mediante la asignación de una etiqueta (TAG) con un valor “agarrable”. Un ejemplo es el objeto que se muestra en la figura 50.

Figura 50

Agregación de tag.



Nota: Objeto que se pude tomar en el juego. Elaborado por: Los autores.

2.2.2. Implementación de los menús e interfaces.

Menú principal. El menú principal del juego está diseñado para brindar al usuario la posibilidad de interactuar con el juego directamente. Este posee cuatro opciones diferentes entre estas opciones se encuentran, el botón jugar, el botón de opciones o configuraciones, el botón cómo jugar y el botón salir como se muestra en la figura 51.

F*igura 51*

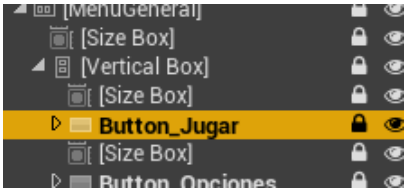
Widget Blueprint Menú Principal.

Nota. *Visualización del widget MenuPrincipal dentro del juego, Elaborado por: Los autores.*

Para la creación de los *widgets* primero se tiene que crear en el editor Unreal un *widget Blueprint*, una vez creado en él código (C++) se busca esa clase creada y se enlazan los componentes.

Figura 52

Widget Blueprint MenuPrincipal botones.



Nota. Despliegue de los botones dentro del editor UE, Elaborado por: Los autores.

Se crea un botón *Button_Jugar* en el *widget Blueprint y* en él código (C++) se busca ese botón utilizando el comando *UPROPERTY (meta = (BindWidget))* el cual permite acceder aún macro con el que se puede especificar ciertas propiedades al editor de Unreal.

Figura 53

MenuPrincipalC++.

```
private:
    //MenuPrincipal
    UPROPERTY(meta = (BindWidget))
    UWidgetSwitcher* MainWidgetSwitcher;

    UPROPERTY(meta = (BindWidget))
    UWidget* MenuGeneral;

    UPROPERTY(meta = (BindWidget))
    UButton* Button_Jugar;

    UPROPERTY(meta = (BindWidget))
    UButton* Button_Opciones;

    UPROPERTY(meta = (BindWidget))
    UButton* Button_ComoJugar;

    UPROPERTY(meta = (BindWidget))
    UButton* Button_Salir;
    //End MenuPrincipal
```

Nota. Componentes de tipo UButton desplegados en la clase MenuPrincipal.h, Elaborado por: Los autores.

En este caso a la propiedad *meta = BindWidget*, la cual permite enlazar la propiedad creada en el *widget Blueprint* con la propiedad creada en el él código (C++), la única condición es que ambas instancias deben tener el mismo nombre. Como se muestra en las figuras 52 Y 53

Figura 54

Inicialización Componentes Widget.

```
bool UMenuWidget::Initialize()
{
    bool FatherSuccess = Super::Initialize();

    (!Button_Jugar ? FatherSuccess = false : FatherSuccess);
    Button_Jugar->OnClicked.AddDynamic(this, &UMenuWidget::OnClickJugar);

    (!Button_Opciones ? FatherSuccess = false : FatherSuccess);
    Button_Opciones->OnClicked.AddDynamic(this, &UMenuWidget::OnClickOpciones);

    (!Button_ComoJugar ? FatherSuccess = false : FatherSuccess);
    Button_ComoJugar->OnClicked.AddDynamic(this, &UMenuWidget::OnClickComoJugar);

    (!Button_Atras ? FatherSuccess = false : FatherSuccess);
    Button_Atras->OnClicked.AddDynamic(this, &UMenuWidget::OnClickAtras);

    (!Button_Salir ? FatherSuccess = false : FatherSuccess);
    Button_Salir->OnClicked.AddDynamic(this, &UMenuWidget::OnClickSalir);

    (!Button_Atras_Opciones ? FatherSuccess = false : FatherSuccess);
    Button_Atras_Opciones->OnClicked.AddDynamic(this, &UMenuWidget::OnClickAtrasOpciones);
```

Nota. A cada botón se les asigna la propiedad OnCLicked (). Elaborado por: Los autores.

Como se muestra en la figura 54, los componentes ya se encuentran previamente creados en el *Widget Blueprint* por lo cual no se tienen que inicializar, pero se tiene que comprobar que estos no sea un *nullpointer* (hace referencia aún espacio de memoria vacío), después de eso se procede asignarle una funcionalidad con el método *onclicked ()* el cual es un método propio del motor y este permite detectar cuando se ha dado un clic en ese botón y poder ejecutar una función específica.

Figura 55

Ejemplo de Función Play.

```
void UMenuWidget::OnClickJugar()
{
    if (!MenuInterface) return;
    MenuInterface->Play();
}
```

Nota. Al presionar el botón Jugar se ejecuta el método de la imagen. Elaborado por: Los autores.

Como se muestra en la figura 55. Estas funciones se llaman al aplastar el botón se utiliza un puntero que hace referencia a una clase de tipo *UGameInstance* la cual tiene la particularidad de que se ejecuta relativamente antes de la creación del juego y no se destruye hasta que el juego se cierre esto permite crear e inicializar el menú, el cual posibilita ejecutar la funcionalidad requerida al aplastar el botón.

Figura 56

Función Play.

```
void UDungeonGameInstance::Play()
{
    UWorld* World = GetWorld();
    if (!World) return;
    if (!MenuWidget) return;
    MenuWidget->ChangeInputMode();
    World->ServerTravel("/Game/Maps/Dungeon");
}
```

Nota. Se accede a la clase de tipo GameInstance para ejecutar el método Play(). Elaborado por: Los autores.

Este método permite invocar a una función llamada ServerTravel() la cual faculta la cargar de un mapa específico ingresando como parámetro el path o ruta del mapa al cuál queremos dirigirnos. Ver figura 56.

Menú de configuraciones. En este submenú se detallan los apartados gráficos que el jugador puede cambiar, estos cambios se van a ver efectuados al momento de aplastar el botón y su configuración quedará guardada dentro del juegos para las siguientes veces que se abra el motor. Ver figura 57

Figura 57

Widget Blueprint Menú Opciones.

Nota. Visualización del widget de Opciones dentro del juego. Elaborado por: Los autores.

Se utiliza un puntero de la clase *UWidgetSwitcher* la cual accede a contener *UWidgets* (es la clase del *WidgetBlueprint*). Con este puntero podemos intercambiar entre *UWidgets. D*e igual manera estos punteros no requieren inicialización ya que se encuentra con la propiedad *meta = BindWidget. Ver figuras 58 y 59.*

Figura 58

Menú Opciones 1.

```
UPROPERTY(meta = (BindWidget))
UWidgetSwitcher* MainWidgetSwitcher;
```

Nota. Este puntero permite hacer el cambió entre UWidget. Elaborado por: Los autores.

Figura 59

Menú Opciones 2.

```
//SubMenuConfiguraciones
UPROPERTY(meta = (BindWidget))
UWidget* MenuSettings;

UPROPERTY(meta = (BindWidget))
UButton* Button_Atras_Opciones;
//End SubMenuConfiguraciones
```

Nota. Creación del submenú Opciones de tipo UWidget . Elaborado por: Los autores.

En el caso de dar clic al botón opciones se comprueba que nuestro *UWidgetSwitcher* no esté referenciado a un espacio de memoria vacío y luego de esto se procede a llamar a la función *SetActiveWidget.* La cual intercambia los *widgets* del menú principal por el puntero al *widget* de nombre *MenúSettings*.Ver figura 60.

Figura 60

Funcionalidad Botón Opciones.

```
void UMenuWidget::OnClickOpciones()
{
    if (!MainWidgetSwitcher) return;
    if (!MenuSettings) return;
    MainWidgetSwitcher->SetActiveWidget(MenuSettings);
}
```

Nota. *Método que se ejecuta al dar Clic al botón Opciones. Elaborado por: Los autores.*

Igual manera al dar clic en el botón *atrás* dentro del submenú hacemos el proceso inverso y utilizamos nuestro *UWidgetSwitcher* para utilizar la función *SetActiveWidget* y como parámetro ingresamos el puntero del menú principal de nombre *MenuGeneral*. Ver figura 61.

Figura 61

Funcionalidad Botón Atrás Opciones.

```
void UMenuWidget::OnClickAtrasOpciones()
{
    if (!MainWidgetSwitcher) return;
    if (!MenuGeneral || !MenuSettings) return;
    MainWidgetSwitcher->SetActiveWidget(MenuGeneral);
}
```

Nota. *Método que se ejecuta al dar Clic al botón atrás dentro del Submenú Opciones. Elaborado por: Los autores.*

Menú de como jugar. En este submenú se detallan todos los diferentes controles que existen en el juego, además que se incluye todas las misiones que tiene que hacer el jugador para poder terminar el juego también contiene un apartado de *tips (Consejos)* para guiar al jugador con todas las misiones que tienen que cumplir. Ver figura 62.

Figura 62

WidgetBlueprint Como Jugar.

Nota. *Visualización del widget dentro del juego del menú Como Jugar. Elaborado por: Los autores.*

En este menú se utiliza la misma lógica del widget opciones, utilizamos nuestro puntero a la clase *UWidgetSwitcher* y llamamos a la función *SetActiveWidget ()* para enviar como parámetro nuestro puntero del menú instrucciones. Ver imagen (Como Jugar C++).

Figura 63

Como Jugar C++.

```
void UMenuWidget::OnClickComoJugar()
{
    if (!MainWidgetSwitcher) return;
    if (!MenuGeneral || !MenuInstrucciones) return;
    MainWidgetSwitcher->SetActiveWidget(MenuInstrucciones);
}
```

Nota. *Método que se ejecuta tras dar clic al botón Como Jugar. Elaborado por: Los autores.*

Menú de juego. Este menú (*Widget Blueprint)* interactúa directamente con el jugador dentro del juego ya que este se puede acceder una vez el boton *Button_Jugar* se presiona, Ver imagen (Widget Blueprint Menú Principal Botones), posee los botones y comandos reiniciar, fin del juego y salir del programa. Ver figura 64.

Figura 64

Widget Blueprint MenuInGame.

Nota. *Visualización del widget MenuInGame dentro del juego. Elaborado por: Los autores.*

En esta clase *UWidget* (La clase del *Widget Blueprint* en C++ es *UWidget*) Se utiliza la misma lógica de las anteriores clases ya que principalmente solo tiene botones a estos se les verifica si son punteros válidos y después de esto se les asigna una función al recibir la acción de ser cliqueados esto gracias al método *onclicked()*. Ver figura 65.

Figura 65

MenuInGame C++.

```
bool UInGameMenuWidget::Initialize()
{
    bool ValidWidget = Super::Initialize();
    (!Button_Reiniciar ? ValidWidget = false : ValidWidget);
    Button_Reiniciar->OnClicked.AddDynamic(this, &UInGameMenuWidget::OnClickReiniciar);
    (!Button_FinJuego ? ValidWidget = false : ValidWidget);
    Button_FinJuego->OnClicked.AddDynamic(this, &UInGameMenuWidget::OnClickFinJuego);
    (!Button_Salir ? ValidWidget = false : ValidWidget);
    Button_Salir->OnClicked.AddDynamic(this, &UInGameMenuWidget::OnClickSalir);

    return ValidWidget;
}


void UInGameMenuWidget::OnClickReiniciar()
{
    if (!MenuInterface) return;
    MenuInterface->Reiniciar();
}

void UInGameMenuWidget::OnClickFinJuego()
{
    if (!MenuInterface) return;
    MenuInterface->SalirJuego();
}

void UInGameMenuWidget::OnClickSalir()
{
    if (!MenuInterface) return;
    MenuInterface->Exit();
}
```

Nota. *Se muestra la clase InGameMenuWidget en la cual se muestran varios métodos OnClick(). Elaborado por: Los autores.*

Interfaz durante la partida. Este menú cumple con la funcionalidad de ayudar al jugador para que pueda seguir el progreso de las misiones. Este es el único que permanece fijo

en la pantalla es decir acompañará al jugador desde el inicio del juego hasta el final de este. Ver figura 66.

Figura 66

Widget Blueprint PlayerHud.

Nota. Visualización del widget PlayerHud en él juego. Elaborado por: Los autores.

Esta clase tiene dos métodos importantes los cuales son SetText() y SetVisibleImage() El primero permite cambiar el valor de los números que se encuentran ubicados en la esquina superior derecha de la pantalla.

Figura 67

PlayerHud C++.

```
void UPlayerHud::SetText(FString Keys)
{
    TB_Llaves->SetText(FText::FromString(Keys));
}

void UPlayerHud::SetVisibleImage(bool IsGrabbing)
{
    if (IsGrabbing)
    {
        NotGrab->SetVisibility(ESlateVisibility::Hidden);
        IsGrab->SetVisibility(ESlateVisibility::Visible);
    }
    else
    {
        IsGrab->SetVisibility(ESlateVisibility::Hidden);
        NotGrab->SetVisibility(ESlateVisibility::Visible);
    }
}
```

Nota. *Se encuentran los métodos para cambiar el contador de las llaves y la imagen al agarrar o soltar objetos. Elaborado por: Los autores.*

El segundo permite cambiar la visibilidad dependiendo si se encuentra agarrando un objeto o no, ambos métodos son propios de cada componente el primero corresponde a la clase *UTextBlock* (esta clase permite manipular texto) y el segundo a la clase *UImage* (Esta clase permite manipular imágenes). Ver figura 67.

2.2.3. Implementación acciones del personaje.

Acción agarrar y soltar. La funcionalidad de Agarrar empieza al aplastar la tecla "E" con ello se ejecuta el código para detectar un si un objeto dentro del mapa puede ser agarrado o no, para esto se utiliza *LineTraceSingleForObjects ()* este método propio de la librería *UKismetSystemLibrary* traza una línea de colisión. Ver figura 70.

Figura 68

Acción Agarrar 1.

Nota. *El personaje se encuentra agarrando un objeto dentro del juego. Elaborado por: Los autores.*

Tras trazar la línea de colisión esta retornará un valor positivo si golpeo con algo y un valor negativo si no lo hiso además que también retornara el primer objeto golpeado,

dependiendo de esto se accede a la etiqueta *TAG* (esta etiqueta o Tag de valor *agarrable* se les asignó a ciertos objetos específicos distribuidos dentro del mapa al momento de su construcción como esqueletos, estatuas, velas, etc.) del objeto golpeado, con este resultado se comprueba si el objeto tiene de etiqueta el valor *agarrable* el objeto se agarrará Ver Imágenes (Acción Agarrar 1 y Acción Agarrar 2), *si* no tiene este valor, la etiqueta puede tener el de valor "llave" Ver figura 85, y sin ninguna de estas ocurre no se agarraría nada.

Figura 69

Acción Agarrar 2.

Nota. *El personaje soltó el objeto agarrado dentro del juego. Elaborado por: Los autores.*

En toda esta interacción existe un booleano de nombre *IsGrab* que permite detectar cuando se aplasta la misma tecla, si hay un objeto agarrado *IsGrab = true* el objeto se suelta, cuando *IsGrab = false* se hace el proceso contrario el cual autoriza agarrar otro objeto. Ver figura 70.

Figura 70

Acción Agarrar C++.

```
void AEscapeDungeonCharacter::Agarrar()
{
    FHitResult HitResult;
    FVector CameraLocation;
    FRotator CameraRotation;
    //Para que el personaje apunte con el moviminto del mouse(Camera) Activar bUseControll
    GetController()->GetPlayerViewPoint(CameraLocation, CameraRotation);
    //Bone Location
    FVector BoneLocation = GetMesh()->GetBoneLocation(FName("Head"));//Validara
    FVector StartPoint = BoneLocation + CameraRotation.Vector();
    FVector EndPoint = CameraLocation + CameraRotation.Vector() * ArmLength;//Vector() ==
    ETraceTypeQuery MyQuery = UEngineTypes::ConvertToTraceType(ECC_Visibility);//Ojito
    TArray<AActor*> ActorsToIgnore;
    ActorsToIgnore.Add(this);
    if (UKismetSystemLibrary::SphereTraceSingle(Mundo, StartPoint, EndPoint, 8.f, MyQuery,
```

Nota. *El método Agarrar () el cual tiene la funcionalidad para agarrar los objetos del mapa. Elaborado por: Los autores.*

Con la variable *IsGrab igual a* falsa se ejecuta la función *Agarrar, de*ntro del método *Agarrar* revisar imagen (Acción Agarrar C++) se guarda el *AActor* que fue agarrado con ello se puede conseguir su *ActorPrimitiveComponent* de nombre *ActorAgarrado* la cual es una clase la cual maneja las físicas de los *AActors.* Ver figuras 70 y 71.

Figura 71

Acción Agarrar función SetGrab.

```
void AEscapeDungeonCharacter::SetGrab(bool Grab)
{
    IsGrab = Grab;
    MyGameInstance->SendIsGrabbing(Grab);
}
```

Nota. El método encargado de cambiar la variable booleana cada vez que se agarra o se suelta un objeto del mapa. Elaborado por: Los autores.

Una vez agarrado el objeto utilizamos la función *AgregarFisicas* con ella se activan las físicas esto faculta mover el objeto en los tres ejes XYZ y que este reaccione con la gravedad, con las colisiones. Después se bloquea la rotación para que el objeto no esté rotando libremente según el movimiento del ratón. Ver figura 72.

Figura 72

Acción Agarrar Físicas 1.

```
void AEscapeDungeonCharacter::AgregarFisicas(UPrimitiveComponent* ActorAgarrado, FVector ActorLocation)
{
    //Comparo Si volvio a Agarrar el mismo Actor q soltó
    if (ActorsToDisassembleFisics.Num() > 0 && ActorsToDisassembleFisics.Contains(ActorAgarrado))
    {
        //No Le agarro si ya esta en el array IsGrab = false
        SetGrab(false);
        UE_LOG(LogTemp, Warning, TEXT("Agarraste el mismo wey >:V"));
        return;

    }
    ActorPrimitiveComponent = ActorAgarrado;
    if (!ActorPrimitiveComponent) return;
    //Configuro al Actor como Movible
    ActorPrimitiveComponent->SetMobility(EComponentMobility::Movable);
    //Configuro el evento Overlap
    ActorPrimitiveComponent->SetGenerateOverlapEvents(true);
    //Configuro el Colission Presets
    ActorPrimitiveComponent->SetCollisionProfileName(FName("Agarrable"));
    //Activo las Fisicas
    ActorPrimitiveComponent->SetSimulatePhysics(true);
```

Nota. El método que se encarga de activar las físicas una vez que el objeto es agarrado. Elaborado por: Los autores.

Con la variable *ActorAgarrado* se utiliza un componente para manejar sus físicas esté componente tiene el nombre *PhysicsHandle* con esto en el método *Tick() (este método es llamado cada frame es decir que es dependiente a la velocidad de procesamiento de nuestra GPU.)* se traslada al objeto agarrado cada frame de acuerdo a la posición del jugador, la rotación de la cámara y una distancia fija la cual es la variable *ArmLength* Ver figuras 70 y 73.

Figura 73

Acción Agarrar Físicas 2 función Tick.

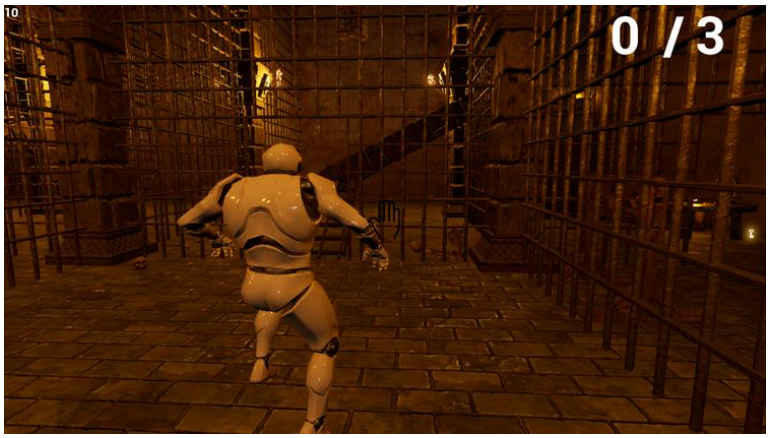
```
void AEscapeDungeonCharacter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (PhysicsHandle->GrabbedComponent)
    {
        FVector TempCameraLocation;
        FRotator TempCameraRotation;
        //Ubicación de la camara
        GetController()->GetPlayerViewPoint(TempCameraLocation, TempCameraRotation);
        //Rotacion de la Camara
        FVector TempFinalPoint = TempCameraLocation + TempCameraRotation.Vector() * (ArmLength-40);

        //Rotacion del Personaje
        //FVector TempFinalPoint = TempBoneLocation + GetActorForwardVector() * 2.f;
        PhysicsHandle->SetTargetLocation(TempFinalPoint);
    }
```

Nota. El método Tick () importante para realizar acciones en bucle. Elaborado por: Los autores.

Acción saltar. Se ejecuta la acción de saltar al aplastar la tecla “Espacio”. Ver figura 74.

Figura 74

Acción Saltar.



Nota. El personaje se encuentra ejecutando la funcionalidad de saltar. Elaborado por: Los autores.

Se utiliza el método *Jump ()* de la clase *ACharacter* la cuál es la clase padre del personaje, esta funcionalidad ya se encuentra programada en el motor. Ver figura 75.

Figura 75

Saltar clase padre.

```
void ACharacter::Jump()
{
    bPressedJump = true;
    JumpKeyHoldTime = 0.0f;
}

void ACharacter::StopJumping()
{
    bPressedJump = false;
    ResetJumpState();
}
```

Nota. La funcionalidad de saltar que se encuentra en la clase ACharacter. Elaborado por: Los autores.

Estos métodos corresponden a la clase ACharacter revisar imagen (Saltar ClasePadre) los cual se encarga del movimiento del personaje (ACharacter)y una de esas funcionalidades que ya tiene implementadas es la de saltar. Ver figura 76.

Figura 76

Función Saltar.

```
else
{
    ACharacter::Jump();
}
```

Nota. *La funcionalidad de saltar la cual se ejecuta al presionar la tecla "espacio".*
Elaborado por: Los autores.

Acción trepar. Esta acción se encuentra en la función de saltar *AEscapeDungeonCharacter::Jump* es decir la función *Jump* de la clase del personaje controlado por el jugador, al aplastar la letra "espacio" ejecuta una condición, si la condición se cumple el personaje trepara el objeto que tiene de frente, pero si la condición es falsa el personaje ejecutará *ACharacter::Jump()* Ver figuras 76 y 77

Figura 77

Función Trepar.

Nota. El personaje ejecutando la funcionalidad de trepar dentro del juego. Elaborado por: Los autores.

La condición a evaluar es un *LineTraceSingleForObjects ()* para detectar si existe algún objeto al frente del personaje con el que colisionamos, exactamente a 150 unidades al frente con el fin de encontrar a este objeto a 150 unidades al frente del personaje se utiliza la posición actual del jugador dada por la función *GetActorLocation(),* la rotación utilizando la función *GetActorForwardVector(). E*ntonces se utiliza la posición más la rotación multiplicado por la distancia para conseguir el punto final de la línea a trazar. Ver figura 78 línea 4.

Figura 78

Funcionalidad Trepar C++.

```
void AEscapeDungeonCharacter::Jump()
{
    if (bIsRolling) return;
    FVector Start = GetActorLocation();
    Start.Z += 28.f;
    FVector End = Start + GetActorForwardVector() * 150.f;
    TArray<TEnumAsByte<EObjectTypeQuery>> Array;
    Array.Add(UEngineTypes::ConvertToObjectType(ECollisionChannel::ECC_WorldStatic));
    TArray<AActor*> IgnoreActors;
    IgnoreActors.Add(this);
    FHitResult HitResult;
    if (UKismetSystemLibrary::LineTraceSingleForObjects(Mundo, Start, End, Array, true,
    {
```

Nota. El método el cual tiene la funcionalidad de trepar si no cumple las condiciones para el mismo ejecuta Jump (). Elaborado por: Los autores.

En el caso de que se encuentre una colisión ejecutamos un segundo *LineTraceSingleForObjects* para verificar la altura del objeto que se intenta trepar, en el caso de que detecte una colisión, se calcula la altura de ese objeto y procede a realizarla acción de Trepar, caso contrario se ejecuta la función saltar *Jump ()* Ver figura 76.

Acción rodar. Tras aplastar la tecla “V” se ejecuta el método *Rodar* el cual incrementa una variable entera de nombre *NumOfRolls* Hasta un máximo de 5 veces es decir de cero a cuatro. Ver figura 80.

Figura 79

Acción Rodar.

Nota. *El personaje ejecutando la funcionalidad de rodar. Elaborado por: Los autores.*

Figura 80

Rodar C++.

```
void AEscapeDungeonCharacter::Rodar()
{
    if (NumOfRolls < 5)
    {
        NumOfRolls++;
    }
}
```

Nota. *El método se ejecuta cuando se presiona la tecla "v". Elaborado por: Los autores.*

Con la variable NumOfRolls conteniendo un mayor a cero dentro del método *Tick ()* se ejecuta una validación múltiple con la finalidad de rodar el número de veces que se aplastó la tecla, Para realizar esta acción se utiliza un método propio del motor de nombre PlayAnimMontage*()* el cual permite ejecutar una animación ingresando como parámetro el puntero a nuestra animación. Ver figura 81.

Figura 81

Rodar utilizando Tick C++.

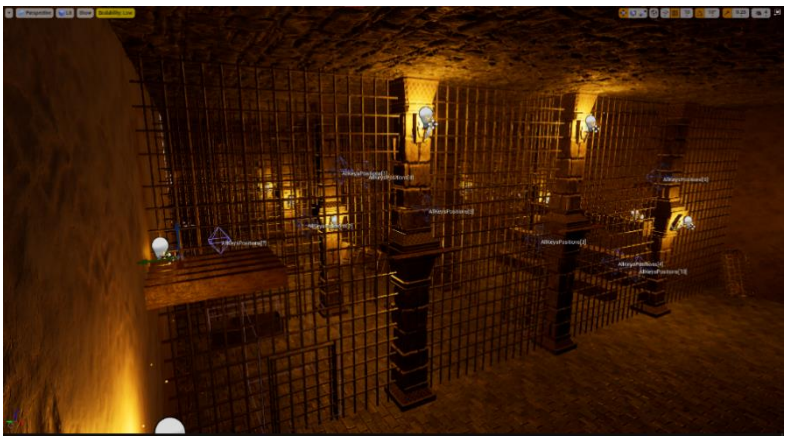
```
if (NumOfRolls > 0 && !bIsRolling && !IsPlayingRootMotion())
{
    bIsRolling = true;
    SetupCharacterToRoll(true);
    PlayAnimMontage(RollMontage);
}
```

***Nota**. Esta validación se ejecuta en el método Tick (), la cual efectúa la animación de rodar.*
Elaborado por: Los autores.

Implementación de las llaves. Las llaves se generan gracias a un Actor de nombre *ARandomKeys* que se ejecuta al iniciar el nivel utilizando el método *BeginPlay ()*, este actor contiene once ubicaciones predeterminadas de entre las cuales se ocuparan tres sin repetición. Ver figuras 82 y 83.

Figura 82

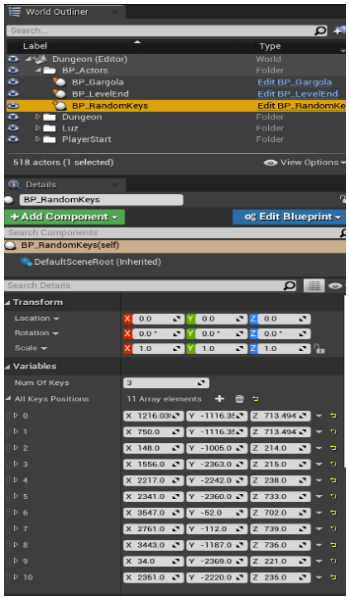
ARandomKeys en el editor.



***Nota.** Se visualiza en el nivel las once posiciones donde las llaves pueden aparecer.*
Elaborado por: Los autores.

Figura 83

BP_RandomKeys.



Nota. Se visualiza en el editor las once posiciones mostrando sus respectivos valores en los tres ejes. Elaborado por: Los autores.

El método *RandomKeys()* permite conseguir 3 posiciones aleatorias, para esto se utiliza el arreglo *AllKeysPositions* el cual almacena todas las posiciones posibles en las que la llave puede aparecer, con ello se genera un *Random*(Número Aleatorio) que sorteé una de estas posiciones almacenadas en el arreglo, a continuación se almacena en otro arreglo de nombre *NoSamePos* esta posición y se procede a comprobar si esta posición estuvo ingresada previamente en el arreglo *NoSamePos* en el caso de que sí volvemos a iterar hasta conseguir una posición que no se encuentre en el arreglo *NoSamePos*, y en caso contrario esta posición se guarda en el arreglo *NoSamePos*. Ver figura 84.

Figura 84

RandomKeys C++.

```
void ARandomKeys::RandomizeKeys()
{
    if (!IsValidArray) return;
    TArray<int32> NoSamePos;
    for (int32 i = 0; i < NumOfKeys; i++)
    {
        int32 Rand = FMath::RandHelper(AllKeysPositions.Num());
        if (NoSamePos.Num() > 0 && NoSamePos.Find(Rand) != INDEX_NONE)
        {
            i--;
        }
        else
        {
            NoSamePos.Add(Rand);
            UE_LOG(LogTemp, Warning, TEXT("Numeros Random -> %i"), NoSamePos[i]);
        }
    }

    for (int32 KeyPos : NoSamePos)
    {
        FVector WorldPos =  GetActorRotation().RotateVector(AllKeysPositions[KeyPos]);
        GetWorld()->SpawnActor<AActor>(Key, WorldPos, FRotator(0.f,0.f,0.f));
        UE_LOG(LogTemp, Warning, TEXT("World Position  -> %s"), *WorldPos.ToString());
```

Nota. Se muestra el método en el cual se ejecuta la funcionalidad de colocar las llaves de manera aleatoria. Elaborado por: Los autores.

Abrir puerta. Cuando el jugador encuentre las 3 llaves distribuidas en el mapa de manera aleatoria se utiliza el objeto de la clase *AChildDoorWithKeys* el cual es la puerta a abrir Ver imagen (Acción Abrir Puerta C+), una vez encontrado se lo utiliza llamando a la función SetBothDoorMovement*()* este pertenece a la clase *AChildDoorWithKeys*, la cual accede a *SetDootMovement()* la cual es una función creada en su clase padre.

Figura 85

Acción Abrir Puerta C++.

```
else if (Tag.IsEqual(FName("Llave")))
{
    SetGrab(false);
    if (LlavesAgarradas < 3)//Block in 3
    {
        LlavesAgarradas++;
        HitResult.Actor->Destroy();
        MyGameInstance->SendKeys(FString::FromInt(LlavesAgarradas));
        if (LlavesAgarradas == 3)
        {
            //Encuentro el objeto que pertenece a la clase AChildDoorWithKeys
            TArray<AActor*> FoundActors;
            TSubclassOf<AChildDoorWithKeys> ClassToFind = AChildDoorWithKeys::StaticClass();
            UGameplayStatics::GetAllActorsOfClass(Mundo, ClassToFind, FoundActors);
            if (FoundActors.Num() > 0)
            {
                AChildDoorWithKeys* TempActor = Cast<AChildDoorWithKeys>(FoundActors[0]);
                if (TempActor)
                {
                    TempActor->SetBothDoorMovement();
```

Nota. Se muestra la sección del código la cual cuenta las llaves agarradas y cumple la función de abrir la puerta si las llaves tienen el valor de tres. Elaborado por: Los autores.

Seguidamente de utilizarlo se cambia el valor de la variable *IsOpen* con esto la puerta se abre es decir gira 90º. Ver figuras 86 y 87.

Figura 86

ClaseHijo OpenDoor C++.

```
void AChildDoorWithKeys::SetBothDoorMovement(bool Move)
{
    PuertaHelp = "OjO";
    SetDoorMovement(Move);
}
```

Nota. El método el cual se encarga de cambiar el booleano y generar el mensaje que aparecerá en la puerta. Elaborado por: Los autores.

Cada puerta dentro del juego hereda su funcionalidad y componentes principales de la clase *AFatherDoor*, y cada una de ellas se abre de diferente manera.

Figura 87

Clase PadreDoor C++.

```
void AFatherDoor::SetDoorMovement(bool bDoor)
{
    IsOpen = bDoor;
}
```

Nota. El método que recibe el booleano el cual permite abrir la puerta. Elaborado por: Los autores.

Acciones de las puertas

Implementación funcionalidad puerta 1. La puerta se abre automáticamente al acercarse a ella como se muestra en la figura 88.

Figura 88

Puerta ChildDoorLevel0.

Nota. Visualización de la puerta ChildDoorLevel0 dentro del juego. Elaborado por: Los autores.

Utiliza un método propio de la clase *UBoxComponent* el cual es *OnComponentBeginOverlap, y* permite detectar las colisiones específicamente las colisiones que se generan por atravesar a su componente de nombre *ColisionComponent.* Ver figura 89.

Figura 89

Puerta ChildDoorLevel0 C++.

```
UPROPERTY(EditAnywhere, Category ="Componentes")
UBoxComponent* ColisionComponent;
```

Nota. El componente de colisión de la puerta ChildDoorLevel0 dentro del header de cpp. Elaborado por: Los autores.

Una vez que se detecta una colisión se ejecuta nuestra función *BeginOverlap* la cual efectuará la acción de abrir la puerta por medio de un booleano de nombre *IsOpen*. Ver figura 90. Si la variable *IsOpen es* verdadera ejecuta el método *MoverPuerta ()* esta va a girar a una rotación relativa a nuestro *mesh* (Modelo de puerta) para eso utilizamos la función *SetRelativeRotation ()* de la clase *AActor*.

Figura 90

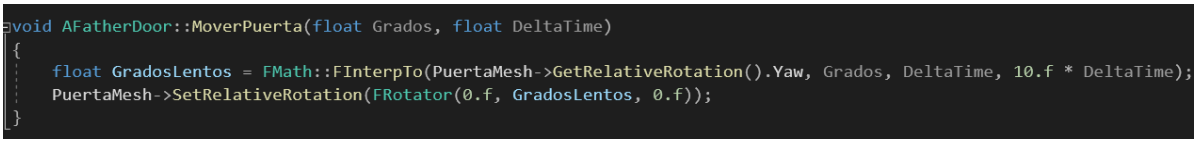
Puerta ChildDoorLevel0 Overlap C++.

Nota. El método BeginOverlap () el cual inicia la funcionalidad de abrir la puerta si detecta que un objeto traspaso al componente de colisión. Elaborado por: Los autores.

Para realizar un movimiento progresivo se utiliza una función de la librería *FMath* la cual tiene el nombre de *FInterpTo().* Esta sirve para hacer una interpolación lineal de un punto "inicial" hasta otro punto "final" y nos retorna un número aproximado al valor del punto final. Ver figuras 88 y 91.

Figura 91

Puerta ChildDoorLevel0 MoverPuerta C++.



Nota. El método MoverPuerta () el cual es el encargado de mover la puerta, haciendo una rotación de noventa grados. Elaborado por: Los autores.

Implementación funcionalidad puerta 2. Esta puerta requiere aplastar una placa para poder abrirla.

Figura 92

Puerta ChildDoorLevel1.

Nota. Visualización de la puerta ChildDoorLevel1 la cual requiere que el jugador aplaste una placa para poder abrirse. Elaborado por: Los autores.

Utiliza el método *OnComponentEndOverlap* el cuál es propio de la clase *UBoxComponent*, este detecta cuándo un objeto deja de estar dentro del componente *UBoxComponent* de nombre *PlacaCollision*. Ver figura 93.

Figura 93

Puerta ChildDoorLevel1 C++.

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Componentes")
UBoxComponent* PlacaCollision;
```

Nota. El componente PlacaColission el cual se encargará de detectar los eventos de colisión que se pueden generar en la puerta ChildDoorLevel1. Elaborado por: Los autores.

Con esto cuando un objeto esté dentro de nuestra Placa Colisión se llama la función *SetDoorMovement* la cuál es la encargada de abrir la puerta. Ver figura 94.

Figura 94

Puerta ChildDoorLevel1 Overlap C++.

```
void AChildDoorLevel1::OnPlacaBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent
{
    if (!Aplastado)
    {
        TriggerActor = OtherActor;
        FVector PlacaPos = Placa->GetComponentLocation();
        PlacaPos.Z = PlacaPos.Z - 8.f;
        Placa->SetWorldLocation(PlacaPos);
        SetDoorMovement(true);
        Aplastado = true;
    }

}

void AChildDoorLevel1::OnPlacaEndOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
{
    if (Aplastado && TriggerActor == OtherActor)
    {
        FVector PlacaPos = Placa->GetComponentLocation();
        PlacaPos.Z = PlacaPos.Z + 8.f;
        Placa->SetWorldLocation(PlacaPos);
        SetDoorMovement(false);
        Aplastado = false;
    }
}
```

Nota. Los métodos BeginOverlap y EndOverlap los cuales detectan cuando un objeto colisiona con la placa y cuando el objeto deja de colisionar con la placa. Elaborado por: Los autores.

Implementación funcionalidad puerta 3. En esta puerta se genera un número aleatorio de entre cinco hasta trece sin tomar el trece y se guarda en una variable entera de nombre *IntNum*.

Figura 95

Puerta ChildDoorLevel2.

Nota. Visualización de la puerta ChildDoorLevel2 dentro del juego. Elaborado por: Los autores.

Esta variable especifica la cantidad de objetos que tiene que ingresar el jugador en el féretro para que esta puerta se abra. Ver figura 96.

Figura 96

Puerta ChildDoorLevel2 C++.

```
void AChildDoorLevel2::BeginPlay()
{
    Super::BeginPlay();
    float Num = FMath::FRandRange(5.f, 13.f);
    IntNum = FMath::FloorToInt(Num);
    PuertaHelp = "Ingresa " + FString::FromInt(IntNum) + " Objectos";
    UE_LOG(LogTemp, Warning, TEXT("Mi rand -> %i"), IntNum);
}
```

Nota. El método BeginPlay ejecuta la funcionalidad para calcular el número máximo de objetos necesarios para abrir la puerta. Elaborado por: Los autores.

Con esto se utiliza la función el método *OnComponentBeginOverlap* para registrar en la variable *NumOfActors* los objetos que ingresaron al componente de colisión y utilizamos el método *OnComponentEndOverlap* para disminuir la variable en el caso de que el jugador saque los objetos del componente de colisión. Ver figura 97.

Figura 97

Puerta ChildDoorLevel2 Overlap C++.

```
void AChildDoorLevel2::OnBaulBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor, UPrimitiveComponent*
{
    if (OtherActor)
    {
        NumOfActors++;
        if (NumOfActors >= IntNum)
        {
            SetDoorMovement(true);
        }
        else
        {
            SetDoorMovement(false);
        }
```

Nota. El método BeginOverlap el cual ejecuta la funcionalidad para detectar cuando abrir la puerta y cuando mantenerla cerrada. Elaborado por: Los autores.

Implementación funcionalidad puerta 4

Figura 98

Puerta ChildDoorWithKeys.

Nota. Visualización de la puerta ChildDoorWithKeys la cual es la puerta final del nivel uno. Elaborado por: Los autores.

Después de obtener todas las llaves se llama la función *SetBothDoorMovement()* la cual cambia el valor de la variable *IsOpen* a true con lo que podemos abrir la puerta utilizando las funciones heredadas, pero esta puerta al ser doble tiene dos *Mesh* de cada puerta es decir que el segundo *Mesh* se gira en el método *Tick()* de la clase *AChildDoorWithKeys*. Ver figura 99.

Figura 99

Puerta ChildDoorWithKeys C++.

```
void AChildDoorWithKeys::SetBothDoorMovement(bool Move)
{
    PuertaHelp = "Oj0";
    SetDoorMovement(Move);
}

void AChildDoorWithKeys::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    if (IsOpen)
    {
        float RotatorZ = FMath::FInterpTo(OtherPuertaMesh->GetRelativeRotation().Yaw, 90.f, DeltaTime, 10*DeltaTime);
        OtherPuertaMesh->SetRelativeRotation(FRotator(0.f,RotatorZ, 0.f));
    }
}
```

Nota. Se muestran los métodos encargados de enviar el booleano para abrir la puerta y el encargado para ejecutar la rotación relativa de la puerta. Elaborado por: Los autores.

Implementación funcionalidad puerta de piedra

Figura 100

Puerta ChildDoor.

Nota. Visualización de la puerta ChildDoor la cual se abre de manera vertical. Elaborado por: Los autores.

Esta puerta edita nuestro método MoverPuerta Ver figura 101, y en vez de rotar el *mesh* de la puerta lo que hace es mover el *mesh* en el eje Z (Yaw) para ello se utiliza la función *SetRelativeLocation.*

Figura 101

Puerta ChildDoor C++.

```
void AChildDoor::MoverPuerta(float Grados, float DeltaTime)
{
    float PosicionZ = FMath::FInterpTo(PuertaMesh->GetRelativeLocation().Z, MeshLocation.Z + 180.f , DeltaTime, 10.f * DeltaTime);
    FVector TemLocation = PuertaMesh->GetRelativeLocation();
    TemLocation.Z = PosicionZ;
    if ((MeshLocation.Z + 180.f) - TemLocation.Z <= 50.f)
    {
        UE_LOG(LogTemp, Warning, TEXT("Hijo VFinal -> %f "), TemLocation.Z);
        return;
    }
    PuertaMesh->SetRelativeLocation(TemLocation);
}
```

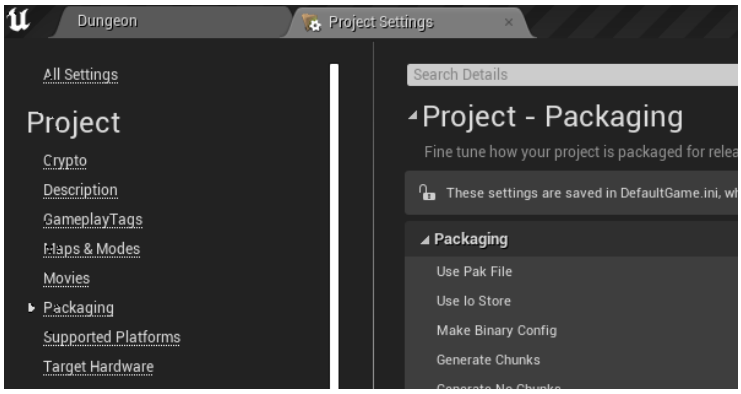
Nota. El método MoverPuerta() el cual es el encargado de mover la puerta, haciendo un leve movimiento en el eje Z utilizando una interpolación lineal para conseguir ese objetivo. Elaborado por: Los autores.

2.2.4. Empaquetamiento

Para configurar el empaquetamiento abrir el apartado de *Project settings* o configuraciones del proyecto y en la parte izquierda de la barra de navegación se encuentra la opción empaquetada. Ver figura 102.

Figura 102

Ventana ProjectSettings.

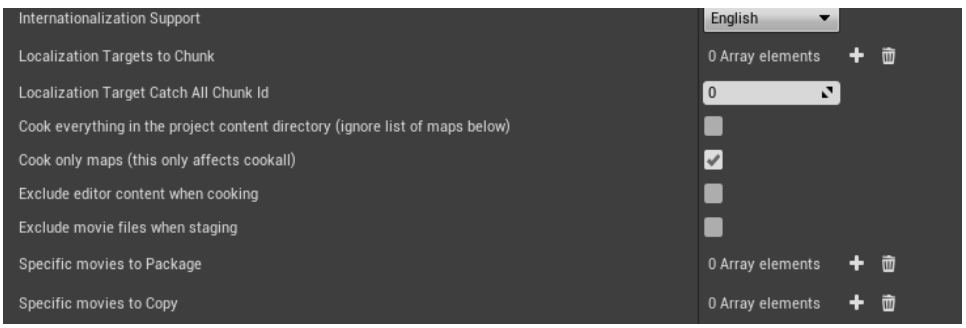


Nota. Ventana de las opciones del proyecto dentro del editor del motor UE. Elaborado por: Los autores.

Señalamos la opción que dice *Cook Only Maps* la cual guarda ciertos mapas seleccionados mas no todos los mapas que puedan existir, gracias a esto mejoramos la velocidad con la que se empaqueta el juego y el peso del ejecutable. Ver figura 103.

Figura 103

Cook only maps.



Nota. Dentro del apartado Packaging se encuentra las opciones necesarias para empaquetar él juego con el menor peso posible. Elaborado por: Los autores.

Para que el paso anterior tenga validez se necesita agregar en la lista que se muestra en la figura 104, la ruta de los mapas necesarios para el juego.

Figura 104

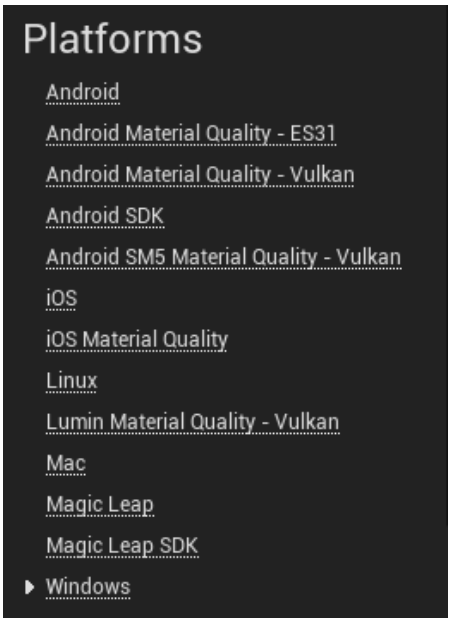
Agregar mapas.

Nota. Se agregan los mapas al empaquetamiento del juego, y se ignora todos los assets que no estén dentro de estos mapas agregados a la lista. Elaborado por: Los autores.

En la ventana de configuraciones del proyecto encontramos las plataformas en las que Unreal empaqueta los proyecto, este proceso se realiza con el fin de generar el ejecutable del juego. Ver figura 105.

Figura 105

Plataformas por defecto.



Nota. La ventana de opciones del proyecto donde se muestra el apartado plataformas dentro del editor UE. Elaborado por: Los autores.

Windows 10. Se seleccionan los drivers Vulkan y DirectX. Ver figura 106.

Figura 106

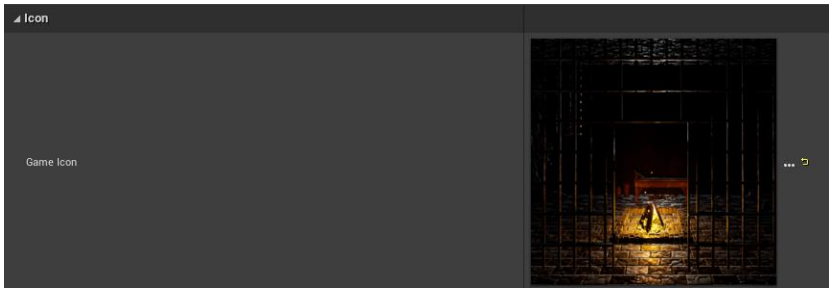
Selección drivers.

Nota. Selección de los controladores de video que podrán ejecutar el videojuego, los cuales se añaden a los prerrequisitos y se instalan automáticamente antes de ejecutar el juego empaquetado. Elaborado por: Los autores.

También se selecciona el icono de la aplicación en este caso tiene que ser máximo 256px * 256px. Ver figura 107.

Figura 107

Imagen Ico.

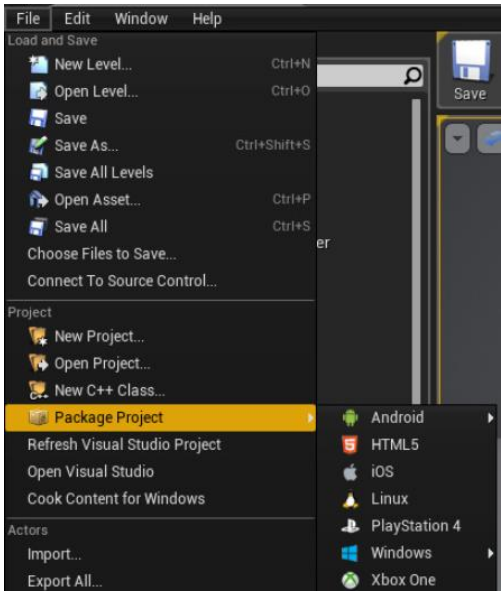


Nota. Apartado para seleccionar el icono de la aplicación dentro de la plataforma Windows los mismos pasos funcionan para Linux. Elaborado por: Los autores.

Tras haber configurado el empaquetamiento para Windows abrir y dar clic a *File->Package Proyect->Windows.* Luego de esto se mostrará el proceso de empaquetamiento, al terminar generará una carpeta de nombre *WindowsNoEditor* dentro de esta se encontrará el ejecutable del juego. Ver figura 108.

Figura 108

Guía Empaquetado Windows.



Nota. Dentro del editor de UE se selecciona la plataforma Windows para empezar a empaquetar el juego. Elaborado por: Los autores.

Linux. Unreal Engine permite que el videojuego se pueda ejecutarse en diversas plataformas, para esto es necesario realizar Cross-compiling (compilación cruzada) que hace posible que se desarrollen juegos para Linux desde un entorno Windows.

Actualmente la compilación cruzada desde Windows es compatible con las plataformas Linux-X86_64 y Linux-ARM. Para esto existen la herramienta toolchain binaries, se puede obtener en la página principal de Unreal figura 109, y cuenta con diferentes versiones.

Figura 109

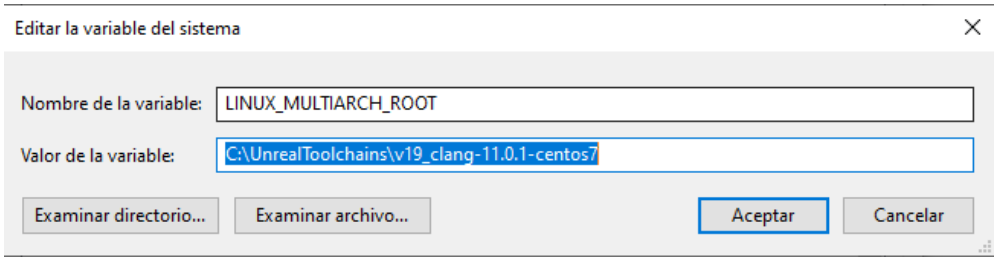
*Clang***.**

Nota: Herramienta para la compilación cruzada clang. Elaborado por: Los autores.

Al descargar e instalar, es necesario configurar las variables de entorno de Windows con el nombre de "LINUX_MULTIARCH_ROOT" como se muestra en la figura 110. El valor debe ser la ruta al directorio clang de la versión instalada que contiene los múltiples directorios de arquitectura de Linux. Se recomienda reiniciar el sistema tras la agregación de la variable de entorno.

Figura 110

Variables del sistema Windows 10.

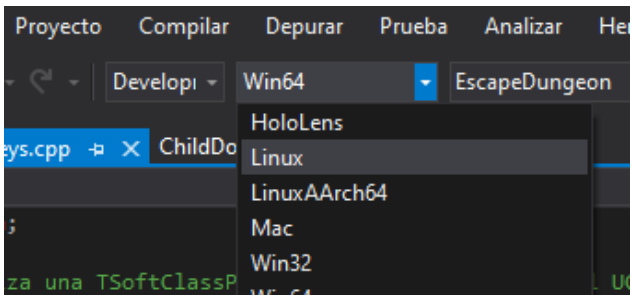


Nota: Configuración para la instalación de la herramienta clang. Creado por: Los autores.

Después de que la variable esté configurada, se volverá a generar los archivos de proyecto de UE4 (usando GenerateProjectFiles.bat) además de reiniciar Visual Studio. Una vez realizado esto la opción "Linux" estará disponible en las configuraciones de Win32/Win64 de Visual Studio como se muestra en la figura 111.

Figura 111

Verificación de instalación de la herramienta.

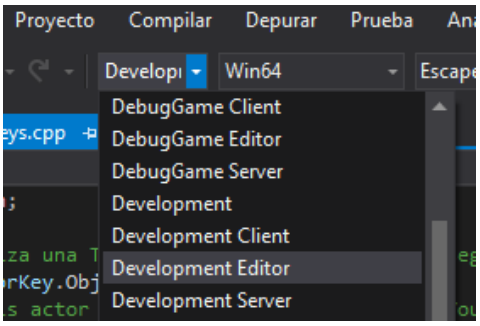


Nota: Verificación para la compilación cruzada. Elaborado por: Los autores.

Se debe configurar “Development Editor” y “Win64” como se muestra la figura 112, finalmente se compila el proyecto figura 113. Ver figura 114.

Figura 112

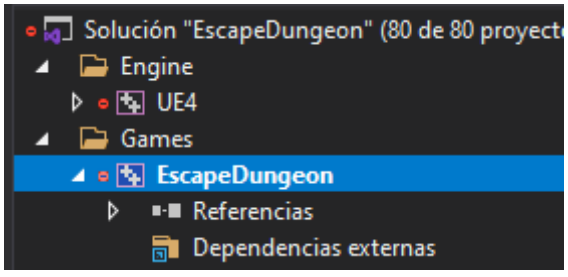
Configuración de visual Studio



Nota: Configuración de visual studio para la compilación cruzada. Elaborado por: Los autores.

Figura 113

Proyecto



Nota: Selección del proyecto para compila. Elaborado por: Los autores.

Figura 114

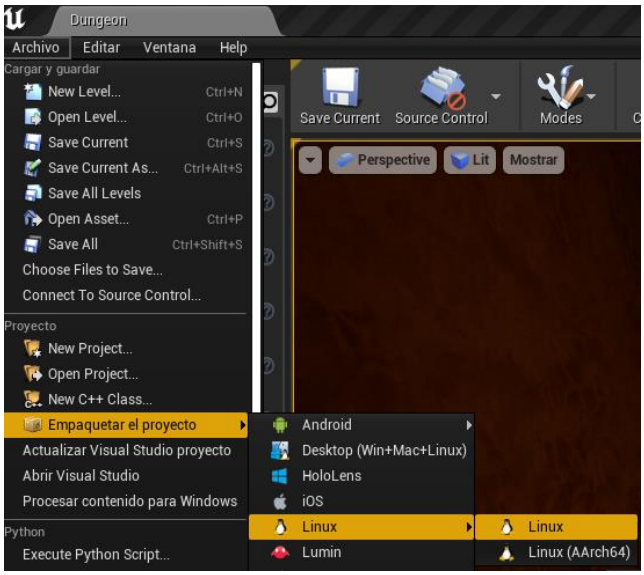
Opción compilar

Nota: Opción para compilación del proyecto. Elaborado por: Los autores.

Para finalizar abriremos el proyecto ya compilado con las configuraciones echas y dentro del editor de Unreal Engine en File->Package Proyect->Linux como lo muestra la figura 115, seleccionaremos el lugar donde se guardará y al finalizar nos quedara el proyecto empaquetado para la plataforma Linux.

Figura 115

Package proyect



Nota: Selección Linux para empaquetamiento. Elaborado por: Los autores.

3. CAPITULO 3: RESULTADOS.

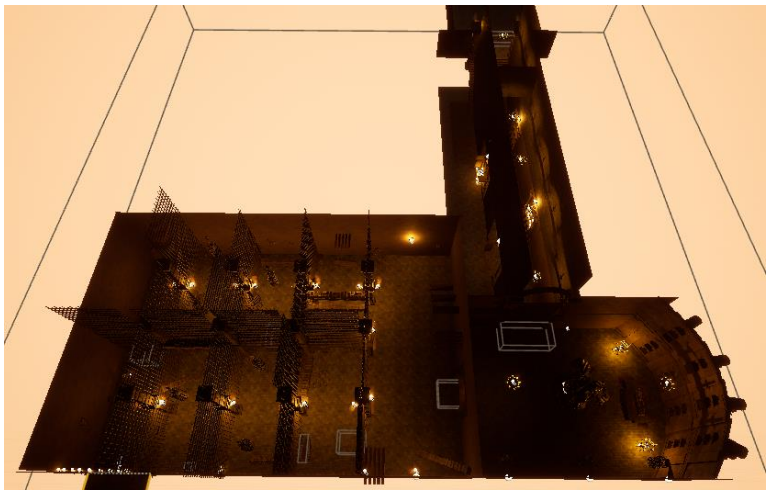
En la siguiente sección se especificarán los resultados que se lograron con la finalización y distribución del videojuego, con la ayuda de las métricas especificadas en la **calidad de un juego en base a su jugabilidad**. Además, se explicará cómo estos resultados permiten cumplir con los objetivos del proyecto planteados.

Para el objetivo “Estudiar el funcionamiento del motor UE” se consiguió desarrollar el videojuego con programación en C++ y utilizando el concepto de Blueprint para diseñar parte de la funcionalidad.

Para el objetivo “Diseñar el mapa del juego” se obtuvo como resultado todo el mundo del videojuego contando este con dos niveles como se muestra en la figura 116.

Figura 116

Mundo del videojuego



Nota: Resultado final del mundo del videojuego. Elaborado por: Los autores.

Para el objetivo “interfaz de usuario” como resultado se consiguió dar interactividad al momento de jugar, representando en el videojuego la facilidad de uso.

Para el objetivo “Implementar la funcionalidad del Juego” se obtuvo como resultado final toda la funcionalidad descrita en el diseño del juego.

Para el objetivo “Realizar pruebas del juego” se realizó una encuesta a un total 20 personas que aceptaron formar parte del proyecto utilizando el juego, se les proporciono el enlace de acceso al videojuego y una encuesta que se llevaría a cabo durante o después de probar el juego. La versión proporcionada a los jugadores fue la versión beta final.

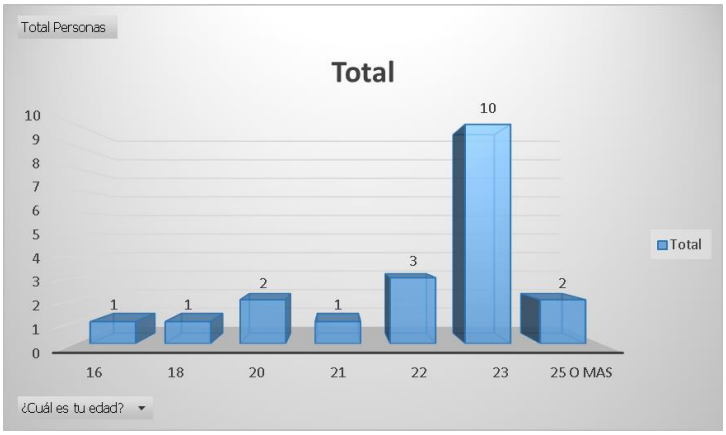
Para la evaluación de los resultados se tomó en cuenta el género de la persona, su edad y su experiencia previa con videojuegos. Arrojando los siguientes datos:

Clasificación por edad figura 117.

- Entre 16 a 19 años con un total de 2 personas.
- Entre 20 a 23 años con un total de 16 personas.
- Entre 24 a 25 años con un total de 2 personas.

Figura 117

Clasificación por edad



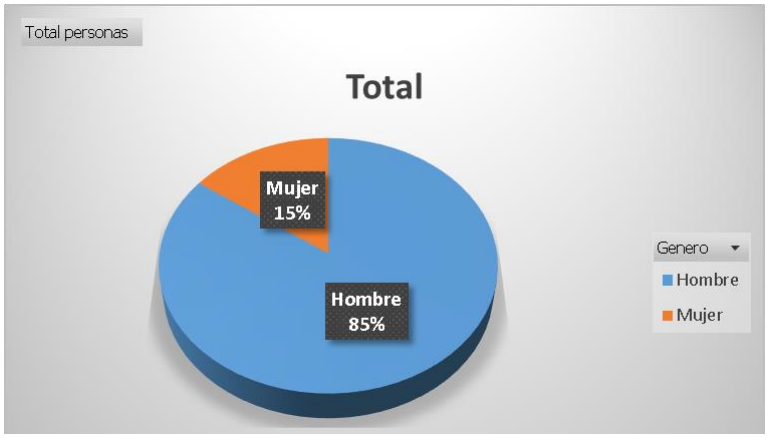
Nota: Gráfico por edad de los participantes que evaluaron el juego. Elaborado por: Los autores.

Clasificación por género figura 118.

- 17 personas fueron hombres
- 3 personas fueron mujeres.

Figura 118

Clasificación por genero



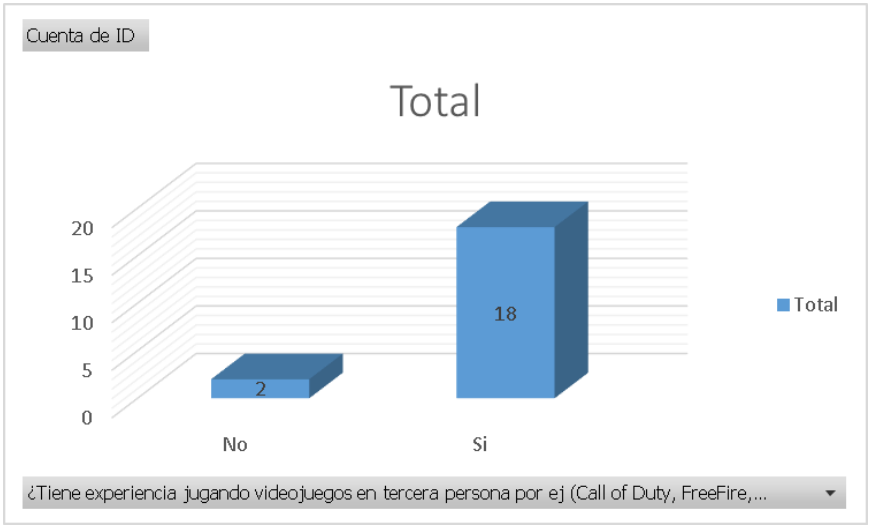
Nota: Grafica de porcentajes de clasificación por genero de los participantes. Elaborado por: Los autores.

Clasificación cono experiencia previa con videojuegos figura 119.

- 18 personas contaban con experiencia previa.
- 2 personas no tenían experiencia previa.

Figura 119

Clasificación por experiencia previa.



Nota: Grafica de participantes que contaban con experiencia previa en videojuegos.
Elaborado por: Los autores.

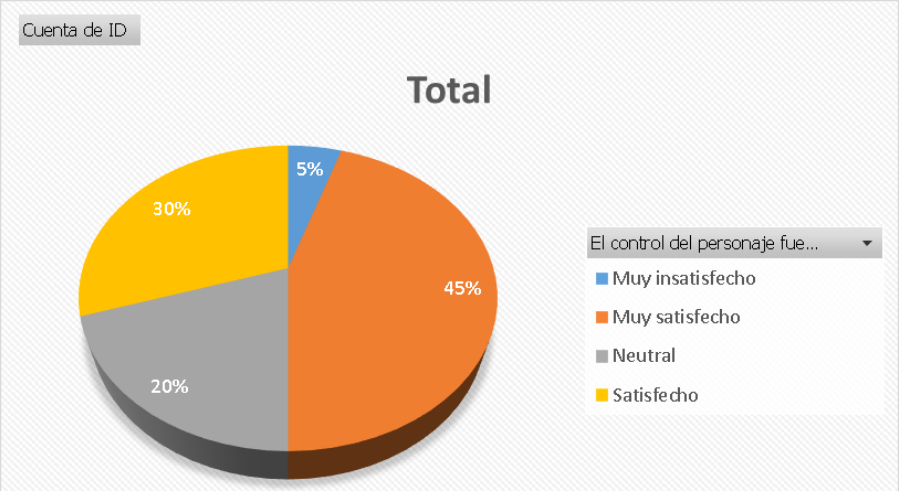
Los resultados obtenidos con base en la experiencia de usuario (UX) una vez utilizado el videojuego, se clasificarán en base a la jugabilidad y rendimiento previamente definidos en el apartado de calidad de juego.

3.1. Jugabilidad

En cuanto a la eficacia se observó que, al momento en que se iba incrementando progresivamente la dificultad, los usuarios las iban completando de manera satisfactoria las metas, de igual manera en la eficiencia y productividad el tiempo que demoraron para aprender los controles fue bajo obteniendo los datos de la figura 120.

Figura 120

Eficiencia del juego



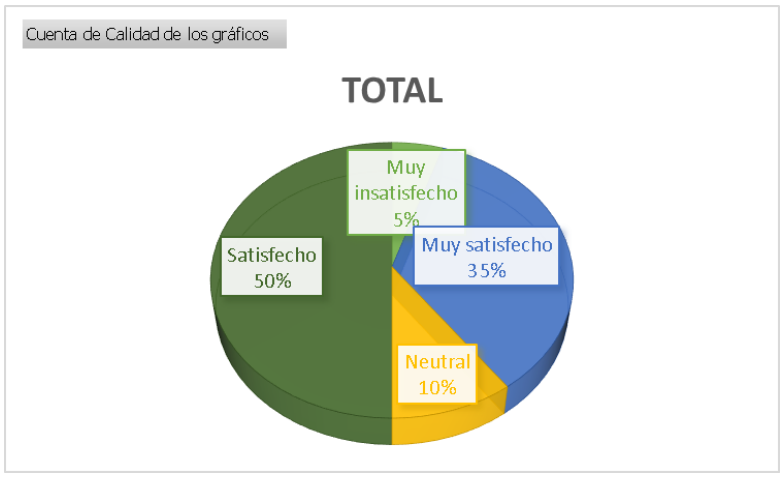
Nota: Resultados de la eficiencia en cuanto al control del juego. Elaborado por: Los autores.

En cuanto a la seguridad y prevención el videojuego no representa un riesgo para el jugador, ya que no cuenta con alguna exigencia física que demande un esfuerzo corporal, por lo que el uso de este solo influye de manera mental lo que prolonga el tiempo en que se utiliza.

En cuanto a la satisfacción del jugador con el juego, se obtuvo en base a la calidad de los gráficos como se muestra en la figura 121, la animación del personaje al realizar determinadas acciones con los resultados que se muestra en la figura 122. Y finalmente una vez culminado el juego se midió la satisfacción en general arrojando los resultados de la figura 123.

Figura 121

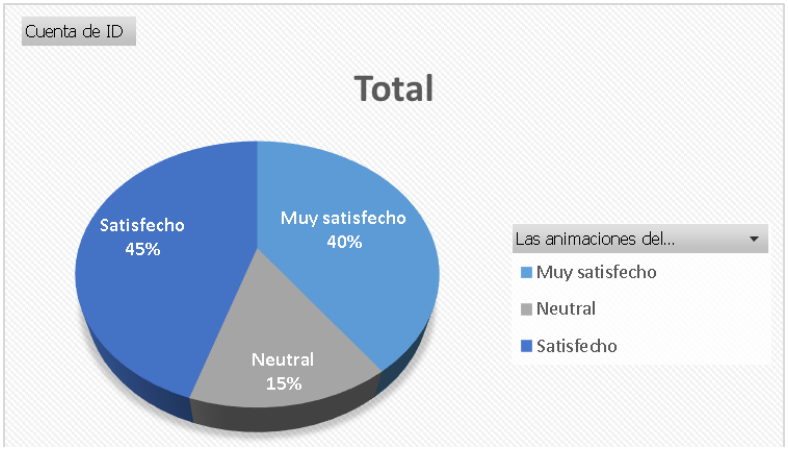
Calidad de los gráficos



Nota: Resultados de satisfacción obtenidos en base a la calidad de los gráficos. Elaborado por: Los autores.

Figura 122

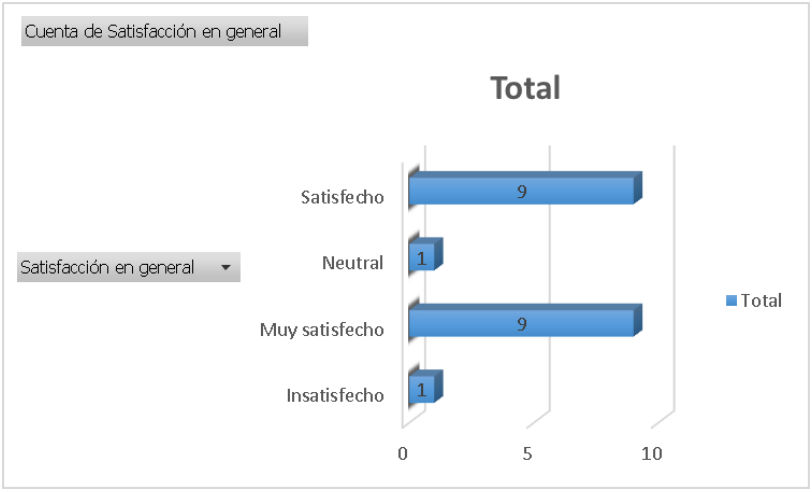
Animación del personaje



Nota: Resultados de satisfacción obtenidos en base a la animación del personaje. Elaborado por: Los autores.

Figura 123

Satisfacción en general



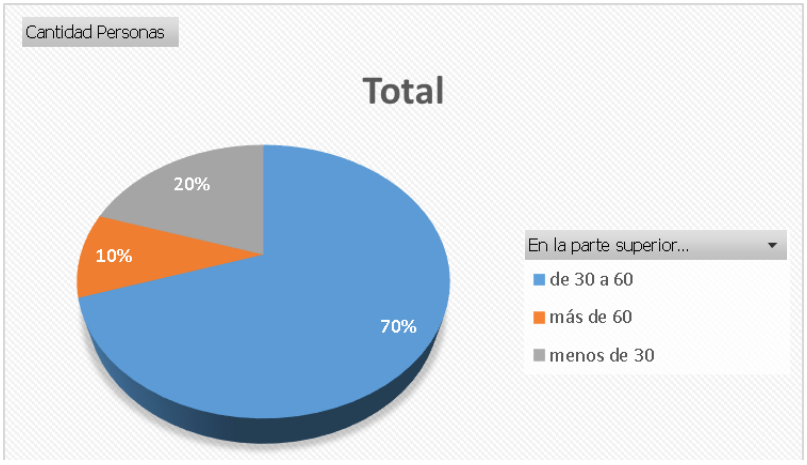
Nota: Resultados en cuanto a la satisfacción en general del jugador. Elaborado por: Los autores.

3.2. Rendimiento

Como se indicó en el capítulo 1 sección de videojuegos una tasa de 45 FPS es la ideal para obtener un buen rendimiento, esto también aplica a valores mayores de 30 FPS

Figura 124

Rango de FPS



Nota: Resultados obtenidos en cuanto al rango de FPS. Elaborado por: Los autores.

Como se puede observar en la gráfica el 70% de personas que ejecutan el juego entre 30 y 60 FPS nos quiere decir que el videojuego tienen un excelente rendimiento, para el 20% de personas que tienen menos de 30 FPS el videojuego se ejecutara con un rendimiento medio a bajo, esto puede ser debido a la ausencia de una tarjeta de video dedicada, a una saturación en la memoria RAM, procesador o disco duro, en general los videojuegos siempre consumen una gran cantidad de recursos de hardware de la computadora por ello a menos que el computador cuente con una tarjeta de video dedicada no se recomienda tener abiertas otras aplicaciones al ejecutar el juego. Pero enfatizamos que a la gran mayoría de los encuestados es decir al 80% ejecuta el videojuego con un rendimiento bueno a excelente.

El objetivo de estos resultados no solo es comprobar que el videojuego sea rápido, eficaz, eficiente, etc. Sino también que se proporcione ese valor añadido que es que el usuario se sienta satisfecho utilizando el videojuego.

CONCLUSIONES.

En definitiva, después de haber creado el videojuego y al observar que los resultados de las encuestas fueron óptimos. Se puede afirmar que mediante la utilización del motor gráfico Unreal Engine 4 se consiguió desarrollar un videojuego de buena calidad, de manera sencilla, rápida y económica esto es gracias a todas las facilidades que el motor ofrece, entre estas la más sólida es el hecho de que una gran parte de funcionalidad se encuentra programada en el motor y este al utilizar el modelo OOP (Object Orientated Programming) genera la facultad de heredar o utilizar funcionalidad de estas clases.

Con respecto a la metodología de desarrollo implementada, se pude concluir que scrum se adapta muy bien al software de videojuegos, ya que con esto se controló desde la idea y concepción inicial hasta su versión final. Esto conllevo que al momento del desarrollo se tuviera un orden en cada etapa logrando con éxito la creación del juego.

En cuanto a la funcionalidad del videojuego en su última versión, se puede decir que Unreal Engine 4 proporciona herramientas las cuales dotan de la interactividad necesaria en base a la funcionalidad diseñada para el juego, lo que provoca como resultado que los usuarios finales se adentren en el mundo en el cual están jugando.

Para terminar, mediante las pruebas que se realizaron con la última versión del juego en base a la jugabilidad y los resultados obtenidos por parte de los usuarios finales, se concluye que el juego tiene calidad y una alta probabilidad que sea recomendado.

RECOMENDACIONES.

Al momento de comenzar con la idea del diseño y la funcionalidad del juego es recomendable guiarse de metodologías que previamente fueron adaptadas a la creación de un software en específico, y con esto ver cual se asemeja más al tipo de trabajo que se quiere implementar. Y si en caso de que no se logre asemejar ninguna, se pude ir adaptando una metodología con el fin que obtener el resultado a conseguir.

Unreal Engine, al tener una gran cantidad de herramientas que facilitan el desarrollo es recomendable consultar el funcionamiento en la documentación oficial. Por otra parte, si se desea implementar una funcionalidad en concreto sin tener mucho éxito, la mejor opción es consultar en la documentación oficial de Unreal Engine o en diferentes foros.

Es recomendable que, al momento de finalizar con el desarrollo del videojuego, este se someta a pruebas por parte de un sujeto externo al proyecto, con el fin de determinar si existen posibles bugs y ver las primeras impresiones que este genera, con ello se consigue mejorar notablemente la experiencia que deja el juego para su versión final.

Para utilizar el motor Unreal Engine al máximo rendimiento se necesita tener un conocimiento básico del lenguaje de programación C++.

Recomendamos la utilización de los blueprints ya que facilitan la manipulación de los componentes de los Actores dentro del editor Unreal Engine.

GLOSARIO DE TÉRMINOS

ASSET: El término Game assets hace referencia a los recursos que utiliza un videojuego, son los Sprite, las animaciones, los paquetes de sonido todo lo que forman parte del juego en el momento de su creación.

BETA: Cuando se habla de la versión Beta de un videojuego, se refiriere a la primera versión completa de un videojuego.

FPS. Es la unidad básica de medida que indica cómo funciona una GPU para un determinado videojuego y se refiere al número de imágenes que muestra por segundo.

GPU: Es la unidad de procesado de los gráficos.

HUD: Estructura grafica que representa datos importantes del personaje en la pantalla de juego.

MUNDO. es un contenedor para todos los Niveles que componen tu juego.

LAUNCHER: es el archivo que arranca el juego, es aplicación que lo ejecuta y archiva.

PLATAFORMAS DE VIDEOJUEGOS: Son sistemas que se crean como bases para hacer funcionar determinados hardware y software concretos.

SHADER: Es un programa o aplicación informática que se ejecuta en la tarjeta gráfica o GPU, y que permite realizar las sombras de los gráficos en un videojuego.

SPRITE: En el desarrollo de videojuego se utiliza para denominar el mapa de bits que creaba en la pantalla un hardware gráfico especializado sin necesidad de usar la CPU del ordenador.

TEXTURAS. Son imágenes que se utilizan en materiales.

LISTA DE REFERENCIAS.

Andres, N. (2018). Videojuegos, la carrera boom, ¿por qué ninguna universidad de Ecuador la ofrece? Retrieved from https://dialoguemos.ec/2018/08/videojuegos-la-carrera-boom-por-que-ninguna-universidad-de-ecuador-la-ofrece/

Arce, L. J. J. M. U. d. A. (2011). Desarrollo de videojuegos.

Belli, S., & Raventós, C. L. J. A. D. R. d. p. e. i. s. (2008). Breve historia de los videojuegos. (14), 159-179.

Cleto, V. D. y. M. (2013). *Desarrollo de Videojuegos: Arquitectura del Motor*. In G. C. y. V. David (Ed.), (pp. 314). Retrieved from https://issuu.com/eslibre.com/docs/desarrollo_de_videojuegos_1__arquit

David Villa, C. G., David Vallejo, Francisco Jurado, Francisco Moya, Javier Albusac, Cleto Martin, Sergio Perez, Felix J. Villanueva, Cesar Mora, Jose Jesus Castro, Miguel Algel Redondo, Luis Jimenez, Jorge Lopez, Miguel Garcia, Manuel Palomo, Guillermo Simmross, Jose Luis Gonzales. (2015). *Desarrollo de videojuegos: Un enfoque practico*. In C. G. y. D. V. David Villa (Ed.). Retrieved from https://issuu.com/freddypenafiel7/docs/desarrollo_videojuegos_volumen3

Department, S. R. (2021a). Leading video game markets in Latin America in 2020, by revenue. Retrieved from https://www.statista.com/statistics/500035/gaming-revenue-countries-latin-america/

Department, S. R. (2021b). Number of employees in the video game industry in selected Latin American countries in 2019. Retrieved from https://www.statista.com/statistics/1230193/workforce-video-game-latin-america-country/

Department, S. R. (2021c). Number of video game studios in selected Latin American countries in 2019. Retrieved from https://www.statista.com/statistics/1229940/number-video-game-studios-latin-america-country/

Engine, U. (2020). Unreal Engine 4 Documentation. Retrieved from https://docs.unrealengine.com/4.27/en-US/

González-Sánchez, J.-L., Montero-Simarro, F., & Gutiérrez-Vela, F.-L. J. P. d. l. I. (2012). Evolución del concepto de usabilidad como indicador de calidad del software. *21*(5), 529-536.

Iglesias, A. A., & Blanque, J. (2011). Desarrollo de videojuegos. In: Universidad Nacional de Luján, Buenos Aires.

Janzen, B. F., & Teather, R. J. (2014). Is 60 fps better than 30? the impact of frame rate and latency on moving target selection. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems* (pp. 1477-1482).

Moltó, A. C., Pellicer, J. L., & Pérez, J. V. M. J. c. T. c. d. d. a. a. l. T. (2012). Gráficos a la máxima potencia: Una comparativa entre motores de juego. *1*(2), 3.

Patiño, A. (2016). Visita Marte en 2016 gracias a la realidad virtual. https://parentesis.com/noticias/videojuegos/Visita_Marte_en_2016_gracias_a_la_Realidad_Virtual

Pekka, A., Outi, S., Jussi, R., & Juhani, W. J. F. V. P. (2002). Agile software development methods-Review and analysis.

Primicias.ec. Una mirada el universo 'gamer'.

S.L., B. (2020). Unreal Engine 4, el motor gráfico que ofrece realismo al máximo. Retrieved from https://www.baboonlab.com/blog/noticias-de-marketing-inmobiliario-y-tecnologia-1/post/unreal-engine-4-el-motor-grafico-que-ofrece-realismo-al-maximo-23

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum* (Vol. 1): Prentice Hall Upper Saddle River.

statistics. (2022). Video Game Industry Statistics, Trends and Data In 2022. Retrieved from https://www.wepc.com/news/video-game-statistics/

Thomsen, M. (2012). History of the Unreal Engine. https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine