UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Programación en Unreal y Distintos Desarrollos de Programación

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Alberto Alonzo Lona López

ASESOR(A) DE INFORME

Ing. Alberto Templos Carbajal

Ciudad Universitaria, Cd. Mx., 2022

Índice

1. Resumen

Desarrollo de un videojuego en Unreal Engine 4 que consiste en la elaboración de distintas mecánicas con el uso de *blueprints* para así ajustarse a los requerimientos de la empresa.

En la actualidad con el uso de los motores gráficos se pueden realizar varios tipos de producciones como películas, diseño de ambientes, producciones virtuales, simulación de arquitectura y construcciones, simulaciones de Realidad Virtual y para este proyecto, videojuegos.

La empresa a la que se le desarrolló venía trabajando productos de valor para la marca, produciendo y financiando un par de cortos con sus propios recursos, *iO Inner Self* (2018) y *Kizuato* (2019), para el año de 2020 se plantearon la idea de realizar un videojuego que llevó por nombre "*Hannah The Game*", por lo cual fui contactado para reunir un grupo de programadores que trabajaran en conjunto el proyecto.

Debido a que el proyecto se elaboró en tiempos de pandemia (julio 2020 - marzo 2021) se trabajó de manera remota, con el uso de repositorios virtuales y definiendo las tareas a realizar semana con semana.

2. Nombre del proyecto.

PROGRAMACIÓN EN UNREAL Y DISTINTOS DESARROLLOS DE PROGRAMACIÓN.

3. Objetivo.

Elaborar el producto demo de un videojuego propuesto por la empresa y subirlo a una plataforma de distribución de videojuegos.

4. Marco teórico.

El proceso de desarrollo de videojuegos es una actividad multidisciplinaria que involucra a profesionistas de las diferentes áreas que abarca. Este *pipeline* se divide en 3 principales etapas, preproducción (etapa en la cual se concibe la idea, se planea, se diseña y se empieza a prototipar), producción (etapa en la que se desarrolla el contenido, se integran los elementos, se hacen pruebas y se lanza el videojuego) y postproducción (etapa posterior al lanzamiento del producto en la que se le da mantenimiento a corto, mediano o largo plazo según el proyecto).

Con los conocimientos adquiridos al cursar la carrera de Ingeniería en Computación y mi participación en la Sociedad de Desarrollo en Videojuegos (SODVI), ambos en la Facultad de Ingeniería de la Universidad Nacional Autónoma de México (UNAM), obtuve los elementos que me permitieron cumplir con los objetivos establecidos por la empresa de desarrollar el demo de un videojuego respetando y siguiendo lo establecido por el *storyboard*, así como las ideas de los directores; a través de un proceso autodidacta con el cual generar soluciones a distintos problemas en un tiempo definido y proponer ideas que ayudaran a complementar lo ya establecido. Para esto usé un enfoque teórico-práctico con la finalidad de encontrar soluciones a los distintos problemas que se presentan a través de un análisis previo, aterrizando las características con las que debe contar la solución del problema.

5. Antecedentes del tema.

Los videojuegos son un medio de entretenimiento interactivo que ha ido creciendo durante los últimos años llegando cada vez a más a una mayor audiencia global a través de distintas plataformas como las consolas, las computadoras, los celulares o los servicios de *streaming* en la nube. Y detrás de cada videojuego existe un proceso de desarrollo de este, creado en conjunto por un equipo (frecuentemente) de personas multidisciplinarias que se desempeñan en distintos roles de desarrollo, como lo son: el diseñador de juegos, el programador, el modelador 3D, el artista 2D, etc. Actualmente la

industria del desarrollo de videojuegos en México se encuentra en crecimiento con algunos juegos *indies* (término que se usa en la industria para referirse a videojuegos desarrollados por estudios independientes, un grupo reducido de personas o a veces por una sola persona) ya posicionados a nivel global como son: *KleptoCats* de *HyperBeard*, *Mulaka* de *Lienzo, Neon City Riders* de *Mecha Studios*, *Pato Box* de *Bromio*, por mencionar algunos.

Si bien la Facultad no enseña directamente desarrollo de videojuegos en la carrera, hay temas de algunas asignaturas a lo largo del plan de estudios que son de ayuda para esto, lo cual se puede ver desde primer semestre con la materia de cálculo y geometría analítica donde el álgebra vectorial es una de las bases para trabajar con espacios tridimensionales abstractos, también todas las asignaturas afines a la programación dentro de la división de ciencias básicas como fundamentos de programación, estructuras de datos y algoritmos I y II, programación orientada a objetos y estructuras discretas, esto porque enseñan las bases de los lenguajes de programación, así como el razonamiento lógico y abstracto que debe existir para la resolución de problemas computacionales antes de empezar a escribir código haciendo uso de diagramas de flujo, pseudocódigo, lógica proposicional, tablas de verdad, y las bases de cómo depurar código.

Dentro del conjunto de ciencias de la ingeniería en asignaturas como ingeniería de software o administración de proyectos de software se enseñan las bases de cómo crear, administrar y dar soporte a un proyecto contemplando el proceso de desarrollo, los requerimientos, la operación, la creación del software, la documentación del mismo, así como del proyecto y el soporte a corto, mediano o largo plazo que se le debe de dar. En otras materias como lenguajes formales y autómatas, se empiezan a ver distintas técnicas de diseño de lenguajes que se pueden aplicar en conjunto con el razonamiento lógico para la resolución de problemas a nivel teórico, como con las expresiones regulares, las gramáticas regulares y los autómatas de estado finito.

Llegando a las materias de ingeniería aplicada, con todas las bases que se han venido trabajando se empiezan a poner en práctica en asignaturas como computación gráfica e interacción humano-computadora, computación gráfica avanzada y temas selectos de ingeniería en computación II (antes, temas selectos de graficación). Estas materias son más enfocadas al proceso de gráficos generados por computadora aplicando en proyectos dentro de clase algoritmos, métodos y técnicas básicas para el dibujo de vectores, curvas, superficies, colisiones y principios de animación esto en *OpenGL*. En el caso de temas selectos de ingeniería en computación es donde considero que se nos da mayor libertad para explorar y experimentar con todo lo que hemos visto, con el uso del

motor gráfico de *Unity*, o software de modelado 3D como *Blender*, para crear un proyecto.

Todo esto en conjunto a la Sociedad de Desarrollo en Videojuegos, la cual busca formar un grupo multidisciplinario comprometido con el desarrollo de videojuegos para las nuevas tecnologías y difundir el desarrollo de videojuegos en México para fomentar la participación de la comunidad estudiantil a través de la creación de videojuegos; en la cual a lo largo de todo el periodo en el que forme parte de la misma fui conociendo las bases del desarrollo de videojuegos a nivel programación, utilizando el motor gráfico de *Unity* para hacer versiones propias de videojuegos sencillos como un *Pong*, un *Infinite Runner*, un *Tetris* y un *Space Invaders*. Además de esto conocí y me adentré en un panorama más general de la industria del videojuego en México, participando de manera activa en distintos eventos del medio y apoyando a organizar otros dentro de la facultad para ampliar mi visión sobre lo que es la industria, conociendo a distintas personas, estudios y desarrolladores del medio como por ejemplo Antonio "*Fáyer*" Uribe exdirector *Hyperbeard*, Oscar Toledo, Ernesto Ríos "*Draco*" y Arturo Nereu de *Unity*.

6. Definición del problema.

A lo largo de la carrera fui miembro activo de la Sociedad de Desarrollo en Videojuegos de la Facultad de Ingeniería de la UNAM, dentro de la sociedad como parte de la difusión y el aprendizaje en el desarrollo de juegos en México participamos en eventos como el *Global Game Jam* (un evento para desarrollar un videojuego completo en un plazo de 48 horas). En estos eventos solíamos repartir tarjetas de la sociedad para hacernos difusión, así como ampliar nuestra red de contactos. Años después gracias a las tarjetas que repartimos una empresa se contactó con la sociedad para tener una entrevista de manera virtual ya que ellos buscaban programadores enfocados en el desarrollo de videojuegos para realizar la demo de un videojuego que ellos querían hacer.

En el primer acercamiento les compartimos la lista de contactos que nosotros tenemos sobre estudios de desarrollo mexicanos, y posteriormente se planteó la idea de que los miembros de la sociedad fueran quienes desarrolláramos la demo con el detalle de que este tenía que ser en el motor gráfico de *Unreal Engine*, en un inicio aclaramos que nosotros poseíamos el conocimiento básico pero que sólo habíamos trabajado con el motor gráfico de *Unity* por la cantidad de recursos y accesibilidad que este nos proporcionaba ya que es más óptimo para todo tipo de equipos y no requiere de una tarjeta gráfica dedicada.

Una vez con esto establecido se me encargó que reuniera un equipo de cuatro o cinco integrantes de la sociedad cuyas capacidades conociera para someternos a una prueba con *Unreal Engine*.

7. Análisis y metodología empleada.

Para llevar a cabo el desarrollo de un videojuego existen distintos motores de videojuegos, para este proyecto, se optó por usar *Unreal Engine*, porque es la herramienta de creación 3D en tiempo real más avanzada para imágenes fotorrealistas y experiencias inmersivas en la actualidad; con una amplia documentación, foros y tutoriales en línea y con constantes actualizaciones. Además de que por la calidad de modelos y texturas que trabaja la empresa es la herramienta donde más se puede explotar el potencial visual.

Se aplicó una metodología basada en Scrum estableciendo bloques de tiempo de 1 a 2 semanas en la cual al inicio de la semana se establecían las tareas que hacer por proyecto, esto en el periodo de julio de 2020 a marzo de 2021.

Los directores del juego junto con la productora de la empresa establecían objetivos a trabajar semana con semana en las juntas de arranque; se hacían dos juntas a la semana, la primera el día viernes donde se establecían los temas a trabajar a lo largo de la siguiente semana, se asignaba las tareas al equipo y se veían los avances pasados; la otra junta era los días miércoles para revisar los avances y puntualizar correcciones que consideraban pertinentes de cara a la siguiente junta. Se continuó con este proceso por casi todo el desarrollo con excepción de finales de año debido a las vacaciones y fiestas decembrinas.

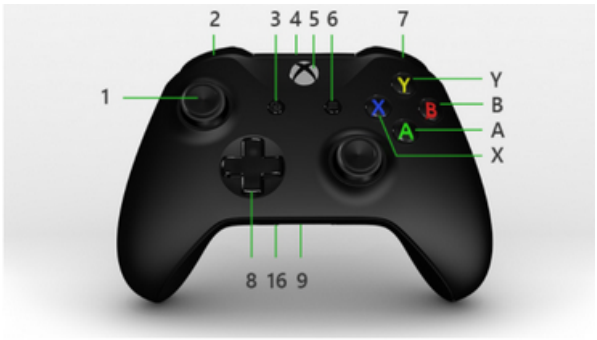
Conforme avanzó el proyecto y se llegaba a etapas más finales en la junta de los días viernes se empezó a generar el *build* (término que se le da al ejecutable del videojuego) del proyecto para probarlo y distribuirlo entre los *testers* (personas que prueban el juego antes que el público en general para detectar posibles fallos en el mismo), para así recibir su retroalimentación y seguir mejorando la experiencia de juego.

8. Participación profesional.

Como primera tarea asignada (a modo de prueba) después de un acercamiento se me encargó (junto con mis compañeros) realizar un sistema de cámaras similar a *Little Nigthmares*, ya que, si bien *Unreal Engine* cuenta por defecto con algunos sistemas de cámaras como en primera y tercera persona, el tipo de cámara que se nos pidió es una mezcla entre cámara fija, de riel y situacional. Para esto empezamos a hacer uso de los *blueprints* que es una alternativa de codificación visual que posee *Unreal*.

Se me proporcionó (junto con mis compañeros) varios recursos como un escenario, texturas, animaciones y modelos, así como un sistema de mecánicas inicial para el personaje principal, entre las cuales estaba, caminar, saltar, agacharse.

A modo de aprender más sobre el manejo de *Unreal Engine*, empecé a buscar información al respecto sobre cómo se debían configurar las entradas para no sólo usar las teclas *W*, *A*, *S*, *D*, para desplazarse sino también lograr conectar y configurar de manera responsiva un control de *Xbox One*, tal y como se ve en la *Figura 1* donde se muestra el nombre que tiene se le asigna a cada botón del control y la *Figura 2*, donde se muestra el mapeo que estos botones tienen en *Unreal*.



Original Xbox One Wireless Controller front

1 Left stick

2 Left bumper

3 View button

Figura 1. División de los botones del control de Xbox One en UE4.

Index	Item Name	Unreal Mapping	Input Type
1	Left Stick	(Move Horizontally) Gamepad_LeftX	Key, Axis
		(Move Vertically) Gamepad_LeftY	Key, Axis
		(Move Left Side More Than Deadzone) Gamepad_LeftStick_Left	Key
		(Move Up Side More Than Deadzone) Gamepad_LeftStick_Up	Key
		(Move Right Side More Than Deadzone) Gamepad_LeftStick_Up	Key
		(Move Down Side More Than Deadzone) Gamepad_LeftStick_Down	Key
		(Click) Gamepad_LeftThumbstick	Key
2	Left Bumper	Gamepad_LeftShoulder	Key

Imagen 2. Denominación de los botones para el mapeo en UE4.

Dentro de los foros y la documentación de *Unreal* encontré la información necesaria para configurar el mando de *Xbox*, esto a través de la ventana de Configuración del Proyecto, en la sección de “Entradas” (*Inputs*).

Esto se hace seleccionando el tipo de entrada que va a recibir para poder ejecutar una instrucción específica, como se muestra en la *Figura 3* donde se puede observar como para el movimiento del personaje se usan las teclas *W*, *A*, *S*, *D*, y a estas se le añade los *Gamepad Left* en el eje *x* y el eje *y*.

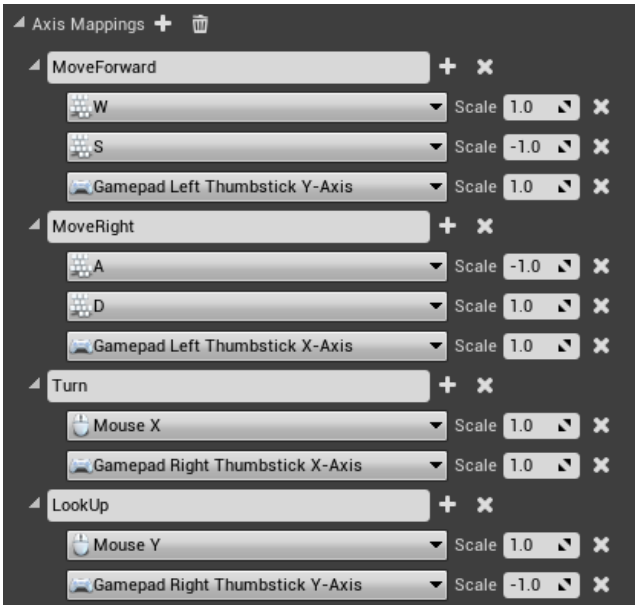


Figura 3. Configuración inicial del control de Xbox One para el movimiento del personaje.

Yo realicé un sistema de cámaras (al igual que mis compañeros), de los cuales la empresa decidió elegir el sistema de cámaras de otro compañero que tenía como base dos trazados de línea, una de ellas, la línea verde es aquella que hace una aproximación al movimiento del jugador a lo largo del espacio disponible y la otra, la línea rosa es la ruta que va siguiendo la cámara conforme se desplaza el jugador, tal y como se puede observar en la *Figura 4*, donde se ve una implementación sencilla de la cámara junto con las líneas de seguimiento.

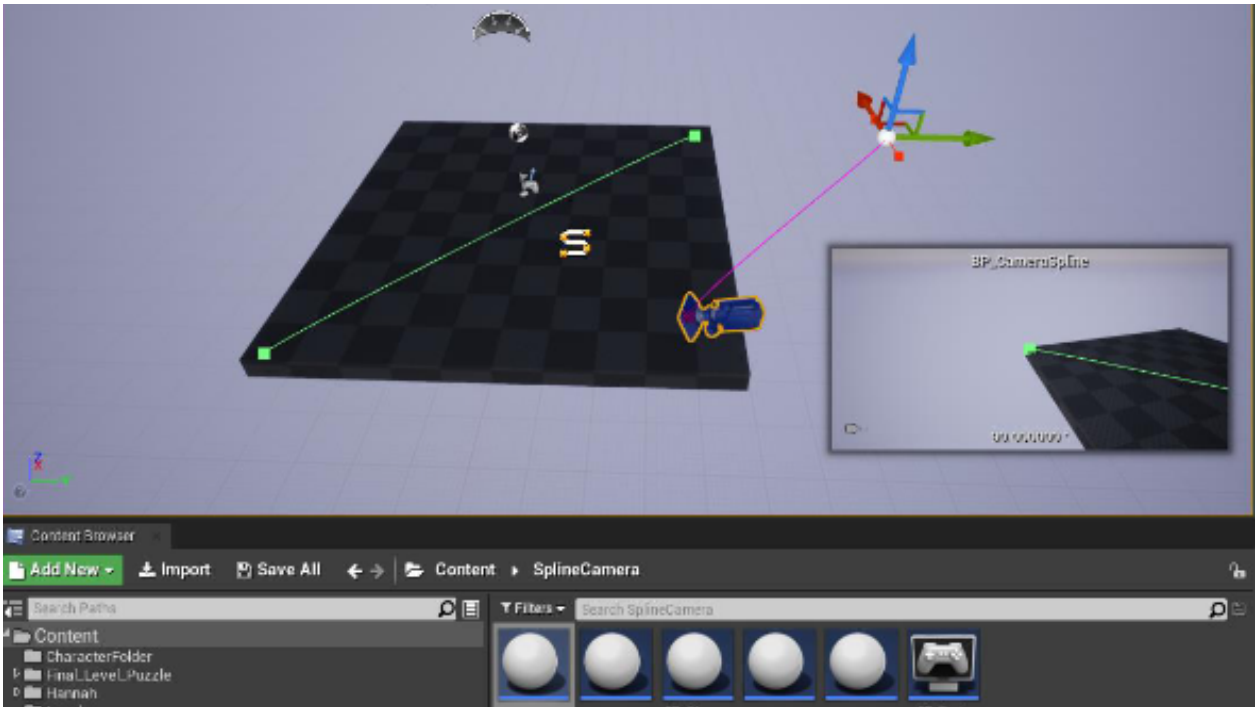


Figura 4. Primera entrega del sistema de cámara.

Como este resultado fue de agrado para la empresa, se empezó a hacer un *pipeline* de desarrollo en el cual, empezaría (junto con mis compañeros) a crear mecánicas de juego tanto para el jugador, los objetos, los enemigos, el entorno, la sucesión e implementación de los niveles.

Lo siguiente que realicé fue crear e implementar en *Unreal* el sistema de comportamiento de un enemigo denominado *larva*. El comportamiento del enemigo se basaba en que este se desplaza de manera aleatoria en un escenario fijo e iba a girar de un lado a otro para después continuar caminando, hasta el punto donde detecte al jugador, donde pasaría a un estado de alerta, preparándose para atacar al jugador, corriendo hasta él para después vibrar un par de segundos y explotar.

Tomando como base esta información que se me brindó y las características que este enemigo debía poseer, ya que al ser un enemigo tipo *minion*, se iba a usar en varias ocasiones dentro del juego; diseñé una máquina de estados que se ajustara a los requerimientos tal y como se muestra a continuación en la *Figura 5*.

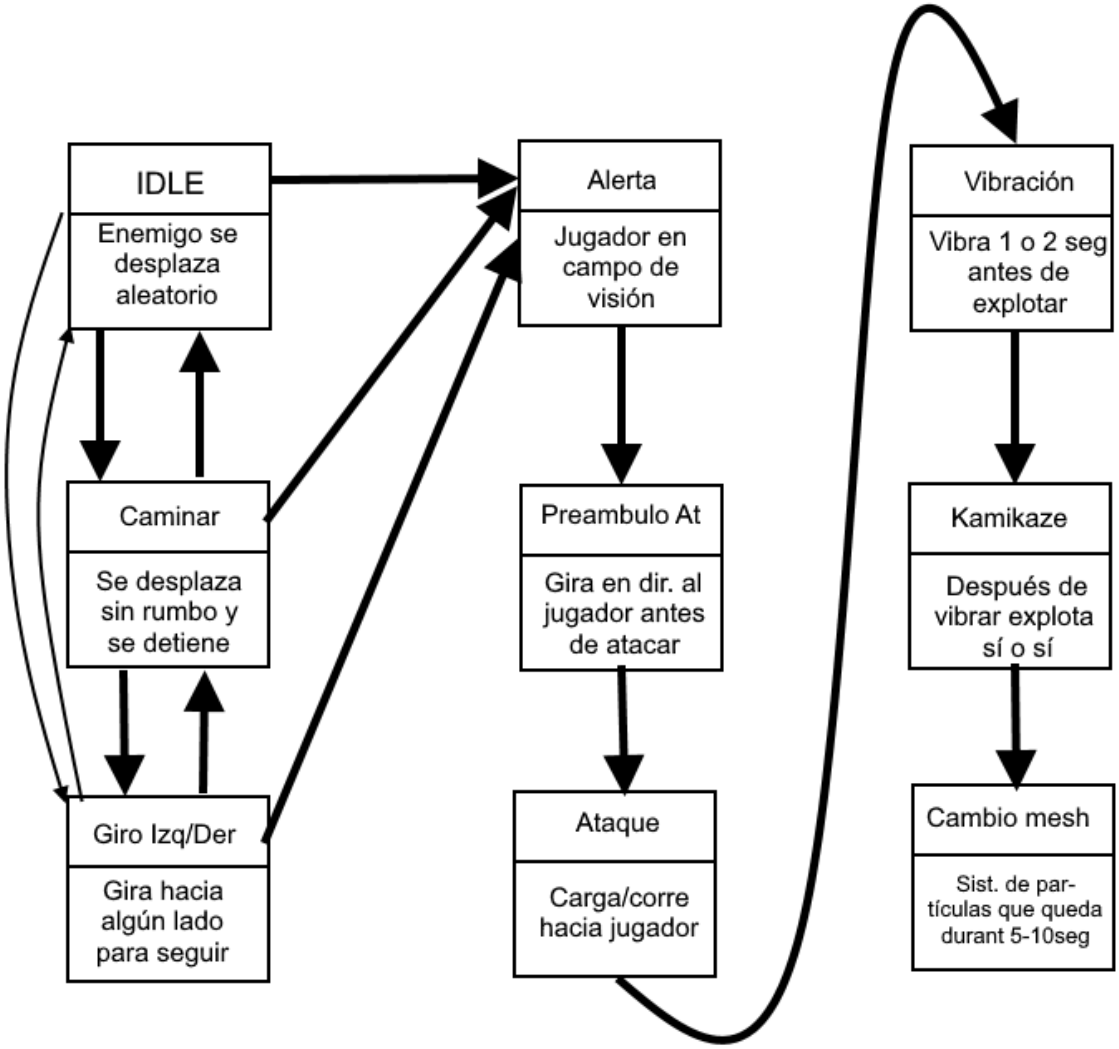


Figura 5. Máquina de estados de enemigo larva.

Una vez que establecí la máquina de estados del enemigo, investigué y seguí tutoriales para hacer este comportamiento dentro de *Unreal Engine*, esto con un *blueprint* de personaje, el uso de un *behavior tree* (árbol de comportamientos) y un *blackboard* (pizarra) en el cual se establecen funciones, para poder controlar su movimiento, su giro, el campo de visión para poder detectar al jugador, la función de ataque con su posterior muerte y los estados establecidos, así como la activación del cambio de un *static mesh* (malla que posee el actor) por un sistema de partículas, a continuación en la *Figura 6* se muestra parte del código del árbol de comportamiento.

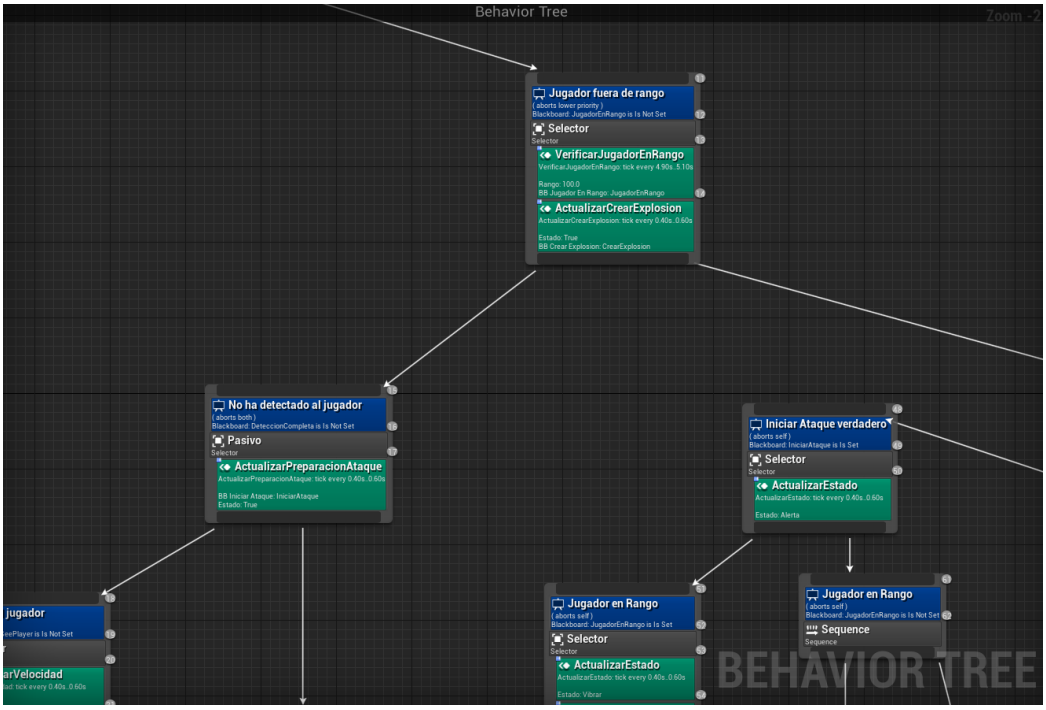


Figura 6. Parte del árbol de comportamiento del enemigo larva.

Junto con esto también trabajé (en colaboración con otro compañero) la máquina de estados para las animaciones del enemigo, conectando la lógica del árbol de comportamientos junto con las animaciones para que cambiara de una animación a otra, esto con un *Animation Blueprint* tal y como se muestra en la *Figura 7*.

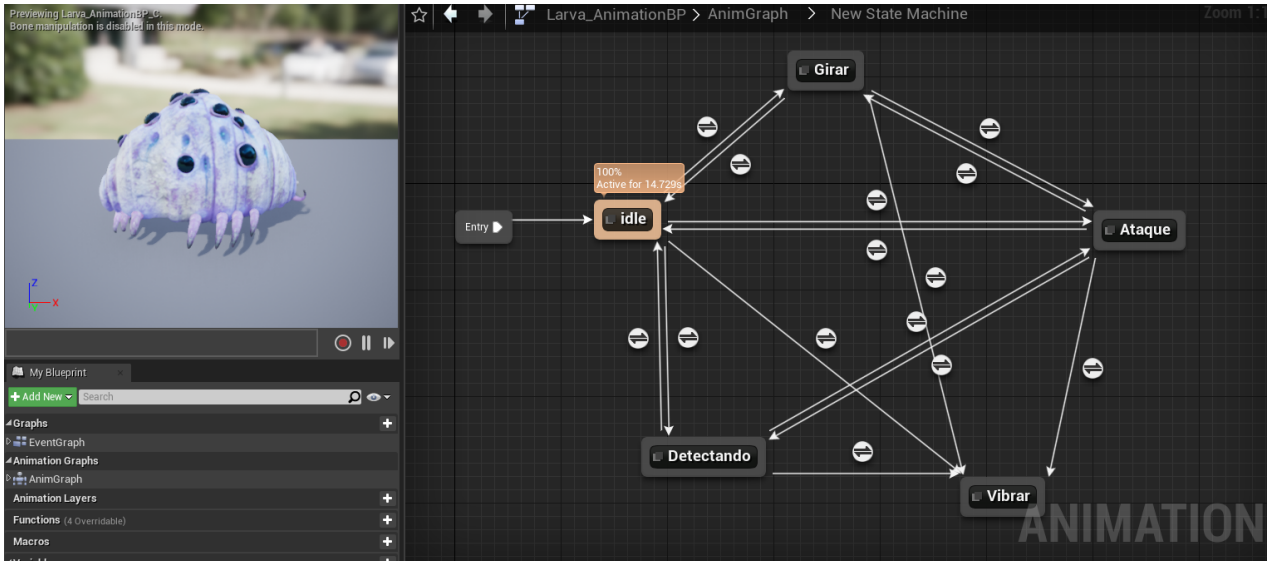


Figura 7. Máquina de estados de las animaciones.

Para hacer que el enemigo se moviera alrededor del nivel utilicé el actor de *Nav Mesh Bounds Volume*, este permite que cualquier otro actor que no sea el jugador (en este caso los actores enemigos) y cuente con un sistema de movimiento propio pueda desplazarse a lo largo del plano *XY*, y con la tecla "*P*" (dentro del *Engine*) se muestra el terreno que este abarca sobresaliendo en color verde como se muestra a continuación en la *Figura 8*.

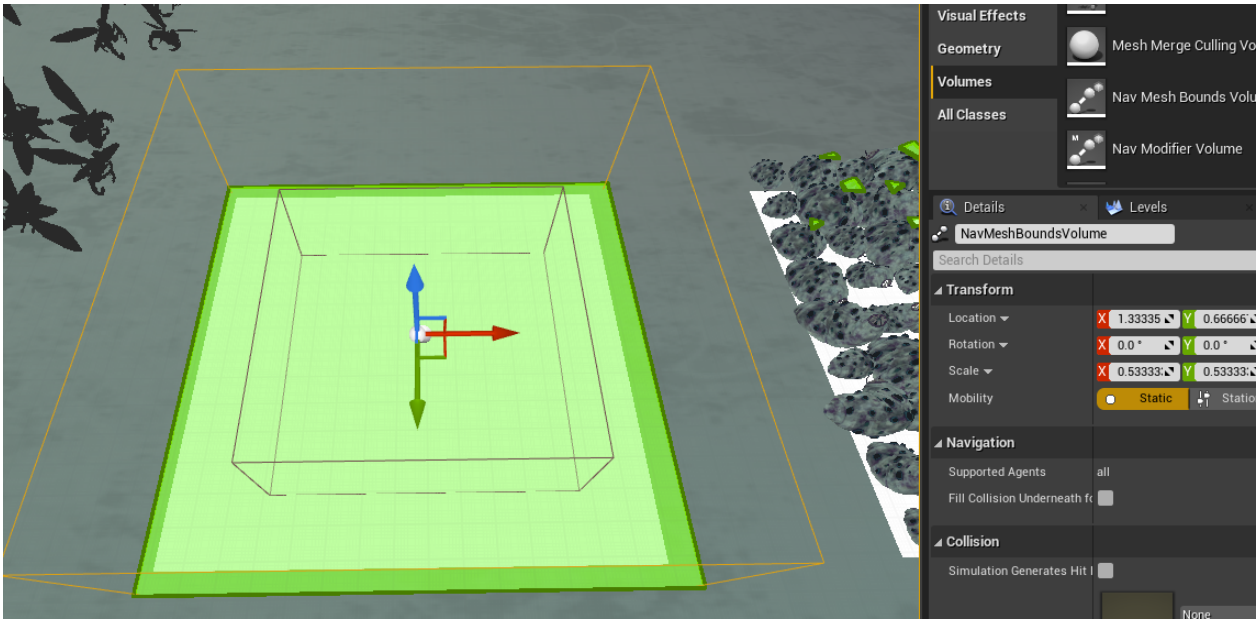


Imagen 8. Nav Mesh Bounds Volume.

Como características adicionales que nos solicitaron para este enemigo fue el instanciarlo varias veces en posiciones aleatorias alrededor de un segmento del mapa. Apoyándome en la documentación y tutoriales de la red, diseñe un *blueprint* el cual dependiendo del tamaño de la caja (o *bound*), va a tomar dos puntos de referencia, uno en el eje positivo de Z y otro en el negativo para verificar dónde está la superficie inferior en la cual pueda instanciar al actor *larva*, con una posición aleatoria en el plano *XY*, una rotación variable de 0° a 360° y una escala de entre 1.2 a 2.2 veces el tamaño de la *larva* original, esto para hacer que los enemigos instanciados se vean diferentes entre sí al aparecer en pantalla, en la *Figura 9* se muestra parte del código que se utilizó para implementar este sistema y en la *Figura 10* se muestra el resultado con un par de líneas rojas que sirven como puntos de depuración para comprobar que el programa se esté ejecutando como debe.

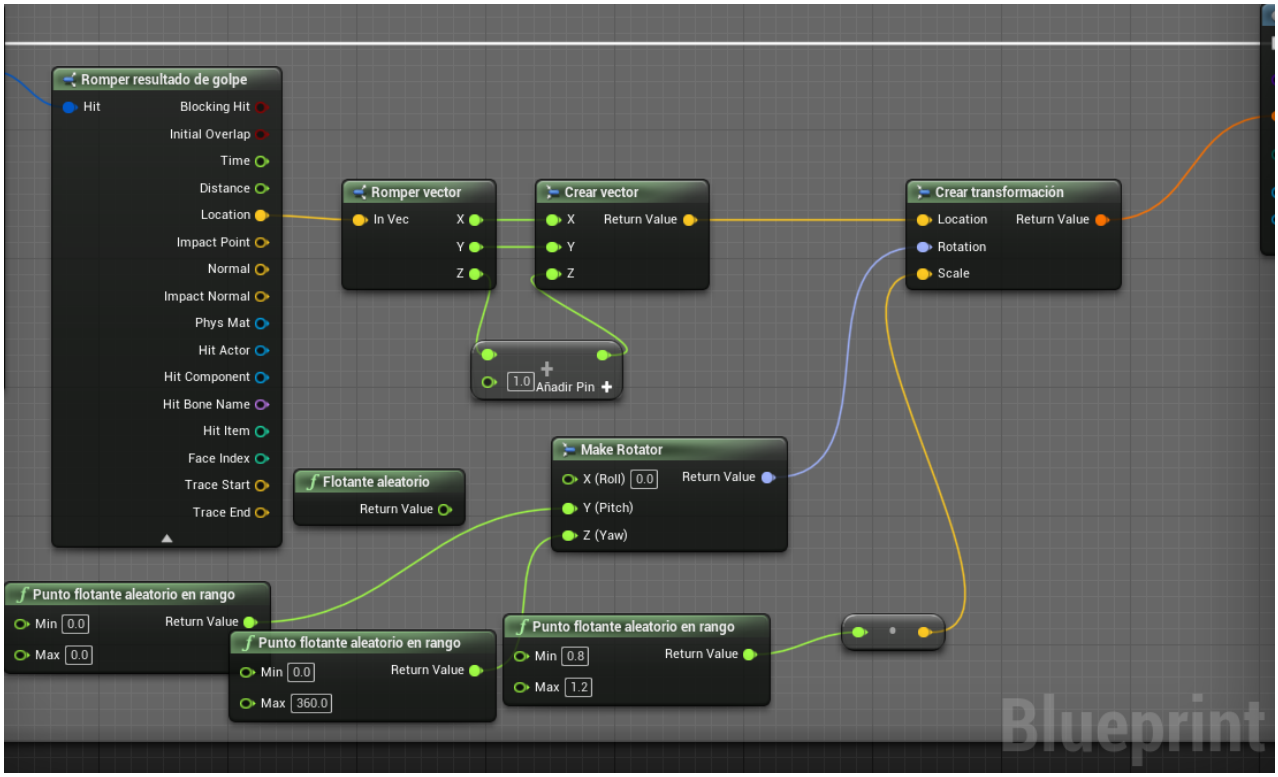


Figura 9. Sistema de instanciación de actor larva.

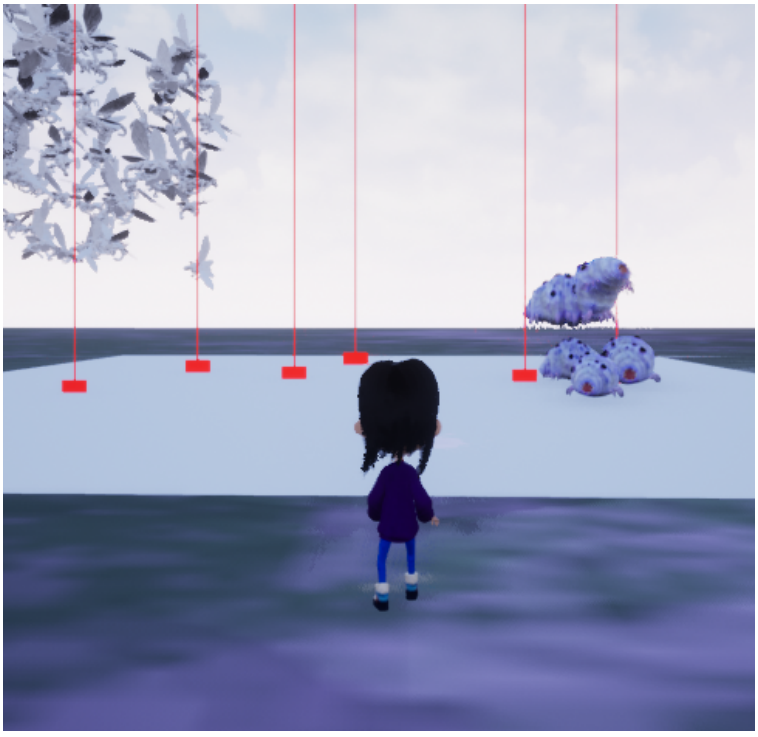
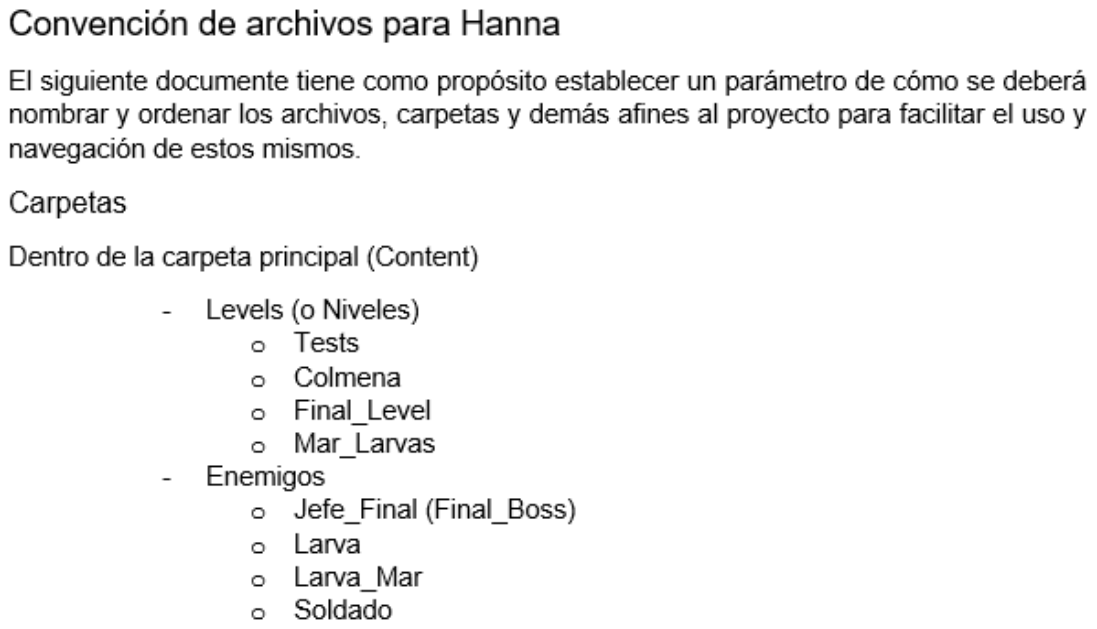


Figura 10. Sistema de spawn de larvas en tiempo de ejecución.

Al tiempo que estaba trabajando el actor denominado *larva*, mis compañeros estaban trabajando otros actores que se usarían posteriormente, así que tomé la iniciativa de crear una convención de los archivos para tener un mejor control y evitar tener elementos duplicados o en desuso una vez avanzado el proyecto, a continuación en la *Figura 11* se muestra parte del documento donde se tenía la propuesta para dicha convención.



Convención de archivos para Hanna

El siguiente documente tiene como propósito establecer un parámetro de cómo se deberá nombrar y ordenar los archivos, carpetas y demás afines al proyecto para facilitar el uso y navegación de estos mismos.

Carpetas

Dentro de la carpeta principal (Content)

- Levels (o Niveles)
  - Tests
  - Colmena
  - Final_Level
  - Mar_Larvas
- Enemigos
  - Jefe_Final (Final_Boss)
  - Larva
  - Larva_Mar
  - Soldado

Figura 11. Propuesta de convención de archivos.

Sin embargo, esta propuesta fue dejada a un lado posteriormente debido a que con el departamento de ilustración y modelado 3D no se llegó a un acuerdo por el avance que ellos ya tenían de varios elementos, con esto decidí (junto con mis compañeros) en crear un tablero de *Trello* para poder asignar, dividir y supervisar las tareas que se tenían que realizar dentro del área de programación, cómo se muestra en la *Figura 12*, y a su vez convencimos a los demás integrantes del proyecto (que no pertenecían al área de programación) de usar el sistema de control de versiones de *git* para así poder trabajar de manera simultánea en distintos actores, niveles y mecánicas de juego; todo esto con el fin de tener un mejor flujo de trabajo.

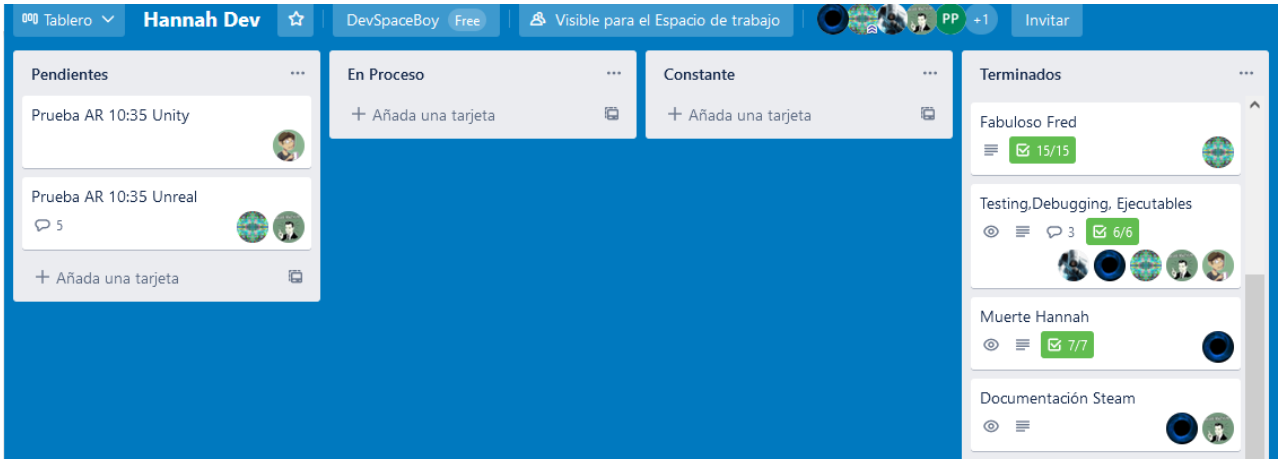


Figura 12. Trello del proyecto.

Una vez con esta organización continué con el desarrollo del comportamiento para otro actor denominado *larva mar*, ya que este es una variante del actor *larva* lo que hice fue tomar su árbol de comportamiento y reducirlo a algo más sencillo para las características que este nuevo actor necesitaba cumplir, como se muestra en la *Figura 13*.

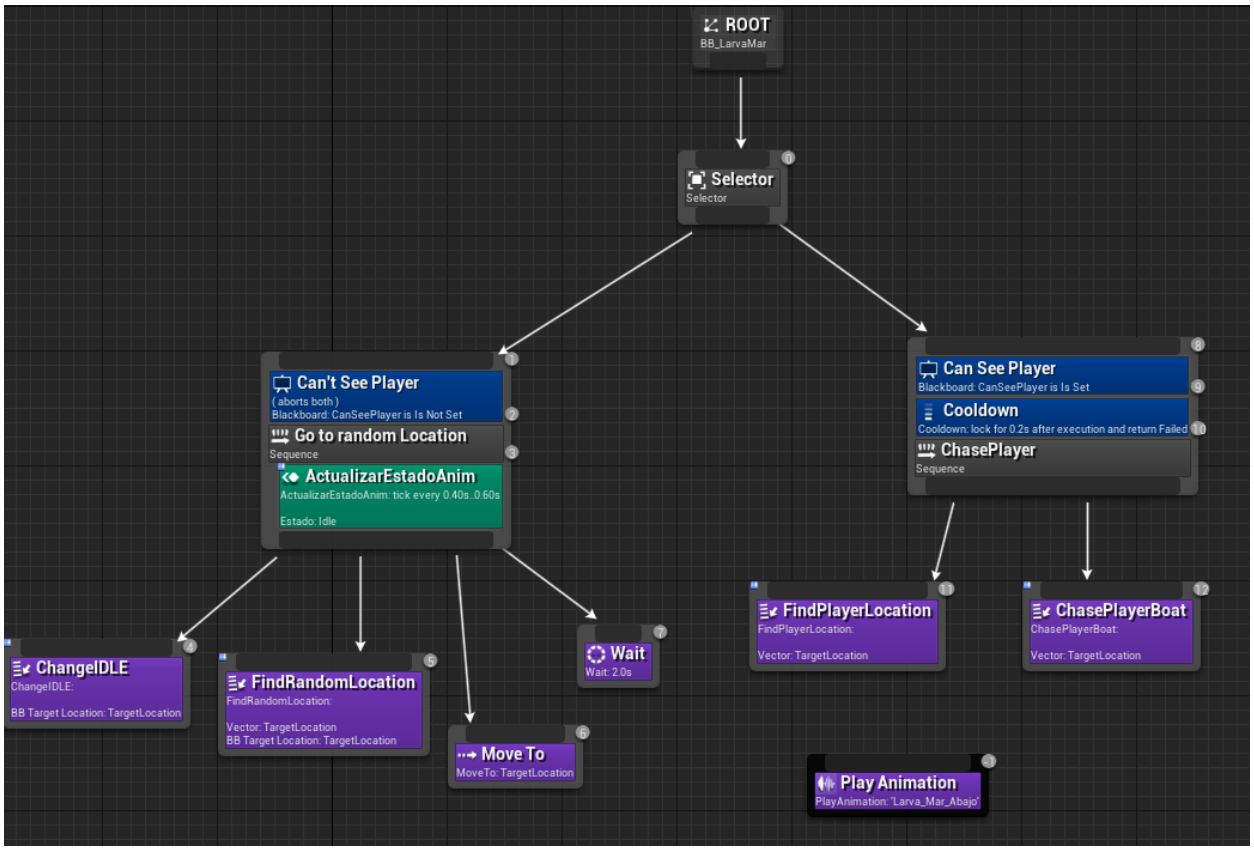


Figura 13. Árbol de comportamiento larva mar.

Con estos dos comportamientos se experimentó en primeras fases de prueba para cerciorarse de que interactuaran de forma correcta con el jugador, y así fue, sin embargo, como este nuevo actor se tenía que mandar a renderizar en una sección del juego de a cientos de unidades empecé a buscar distintos modos de hacer el llamado del objeto sin que este consumiera tanta memoria ni rendimiento, por lo que empecé a hacer pruebas con el sistema de partículas del motor.

De nueva cuenta apoyándome con la documentación oficial y varios tutoriales empecé a familiarizarme con el sistema de partículas, haciendo primero unos ejercicios para ver las capacidades y limitaciones del sistema llegando a buenos resultados como una cascada de la *larva mar*, o fracciones de la malla de la *larva mar* en pequeños pedazos simulando como si estas salieran de una trituradora para otro segmento del juego, como se visualiza en al *Figura 14*.

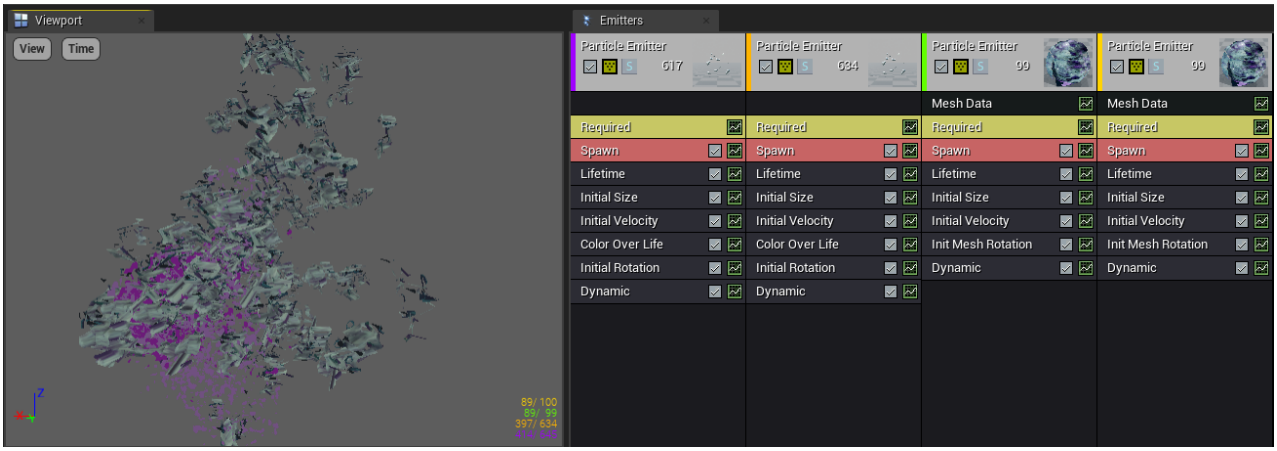


Figura 14. Fragmentos de larva mar “triturada”.

Después de estos y otros más ejercicios noté que en efecto el sistema de partículas era más ligero para mandar a llamar a varios actores (la *larva mar*) porque hacía una sola llamada de las partículas y dentro de esta llamada es donde se hacía el subproceso de estar llamando los actores como componentes de las partículas de los cuales se podía controlar y modificar su tiempo de vida, el número de “subactores” a instanciar, sus transformaciones, así como su escala, entre otros parámetros. Por lo que se volvieron a hacer las pruebas con el jugador y las *larvas mar* para ver el rendimiento en tiempo de ejecución y su interacción, y si bien ahora el rendimiento y la demanda era menor había un problema con el sistema de partículas, estas no poseen ninguna colisión para con otros objetos por lo que el jugador podía fácilmente atravesarlas cosa que no se tenía planificada que debía hacer.

Habiendo llegado a ese resultado e investigando dentro de la documentación, me hicieron llegar más modelos de objetos que se tenían que añadir a las escenas (o niveles) para seguir con las pruebas porque el actor *larva mar* se trabajó para una sección sobre la cual el jugador iría navegando sobre una balsa creada por circuitos y componentes electrónicos por lo que me di a la tarea de acomodar los materiales del modelo dentro del motor gráfico, como se ve en la *Figura 15*, porque al momento de importarlos hubo un error que desconfiguró todos los materiales, aunado a eso con un *Plug-in* de *Mesh Editing* edité el modelo (como su nombre lo dice) para poder separar la parte superior de la inferior. Porque posteriormente se necesitaría el modelo de la balsa por partes para hacerlo interactivo con el usuario.

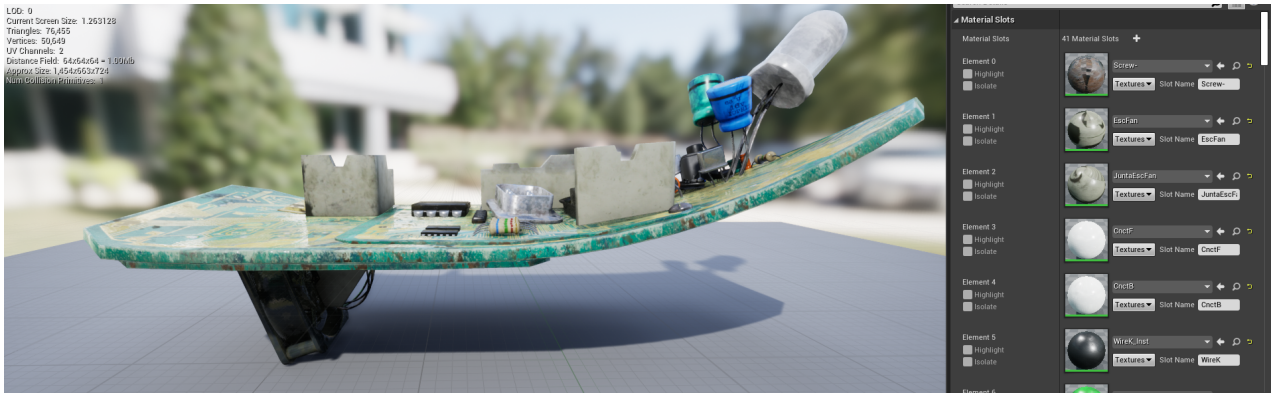


Figura 15. Balsa

Con los elementos de la balsa en su lugar, empecé a trabajar en conjunto a otro compañero que se encontraba realizando el sistema de movimiento para este actor, esto con el fin de empezar a hacer pruebas de interacción entre el actor *larva mar* con el actor balsa y buscar la mejor experiencia posible.

Como todo lo que últimamente estuve trabajando se relacionaba directamente con el nivel denominado “*Mar de larvas”*, se me asignó la tarea de ver el modo de crear aquello que daba nombre al nivel, un “mar de larvas”. Al inicio estaba pensando en cuál sería el mejor método para solucionarlo y cumplir con los requisitos de lo que querían que incluyera este nivel como: incluir las 5 animaciones que nos habían proporcionado para el actor *larva mar*, que los actores no se encimaran entre sí, que cubrieran la mayor área posible y que corriera a un buen rendimiento.

Al inicio opté por readaptar el *blueprint* creado para el actor *larva*, modificando algunos parámetros para mandar a llamar más actores en cada proceso, y si bien este proceso iba a saturar la memoria debido a la gran cantidad de actores que debían cubrir secciones del nivel, decidí intentarlo para conocer las limitaciones a las que me

enfrentaba y buscar una solución viable. En la *Figura 16* se ven bastantes líneas rojas que representan una por cada instancia de un actor *larva mar*.

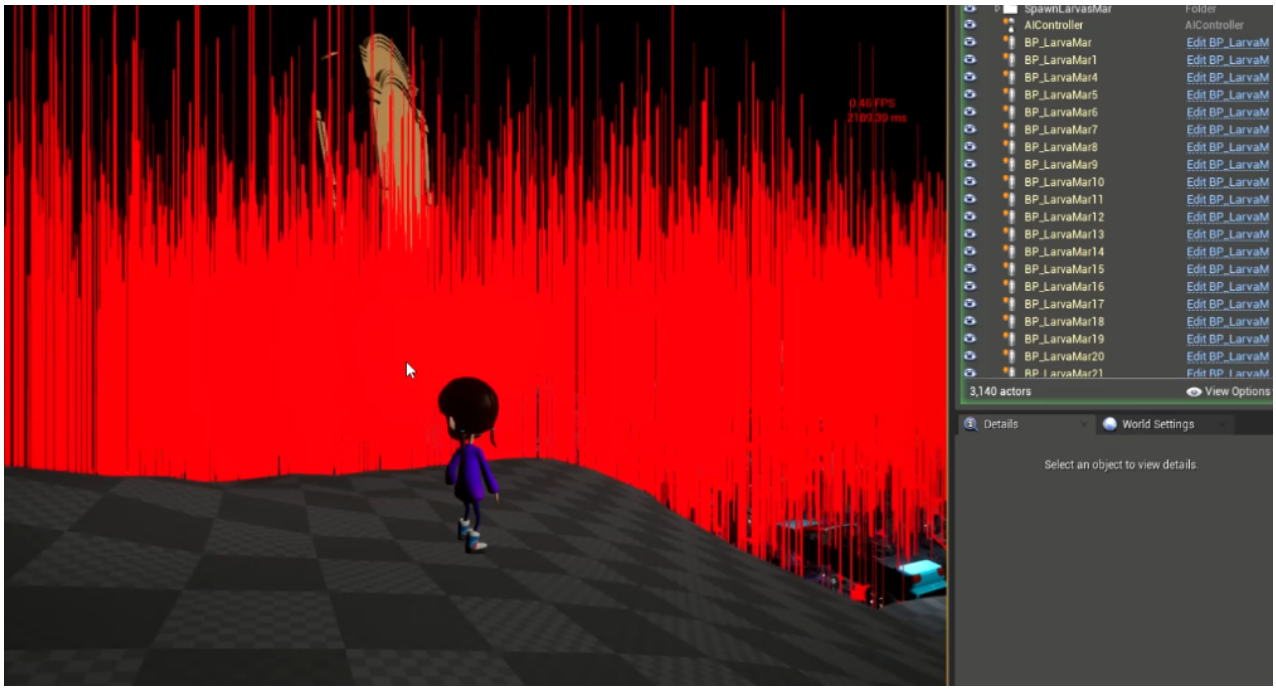


Figura 16. Instancia de actores larva mar a lo largo de todo el nivel.

Debido a que el modo de llamado es a través de un *Box Trigger*, al entrar en el volumen del cubo se crean instancias a los actores, se me ocurrió dividir el nivel en segmentos y en cada espacio poner un generador de actores. Y una vez que se hayan recorrido todos los segmentos no se genere una saturación de memoria como en el caso anterior donde se mandaban a llamar todos los actores de golpe, volví a modificar el *blueprint* manteniendo que al momento en que el jugador entre al volumen se generen los actores *larva mar*, pero al momento que se detecte que el jugador ha abandonado el volumen, estos actores previamente creados se van eliminando después de algunos segundos y así tener un mejor rendimiento en la experiencia de juego.

Pero al revisar más a detalle porque este actor era tan pesado de renderizar, noté que la complejidad del material era muy pesada para el motor gráfico ya que era un material *Depth Fade (desvanecimiento de profundidad)*, el cual evita que los objetos se vean como si estuvieran pegados los unos a los otros y desvanece ese cruce entre objetos opacos y traslúcidos, aunado a esto el material del objeto tenía la propiedad de transparencia lo cual también eleva el costo computacional del actor, esto se puede ver de mejor modo en la *Figura 17* donde se muestra la escena con actores con el modo de

complejidad de *Shader* donde entre más verde más óptimo es pero si es rojo o blanco es más complejo para procesar.

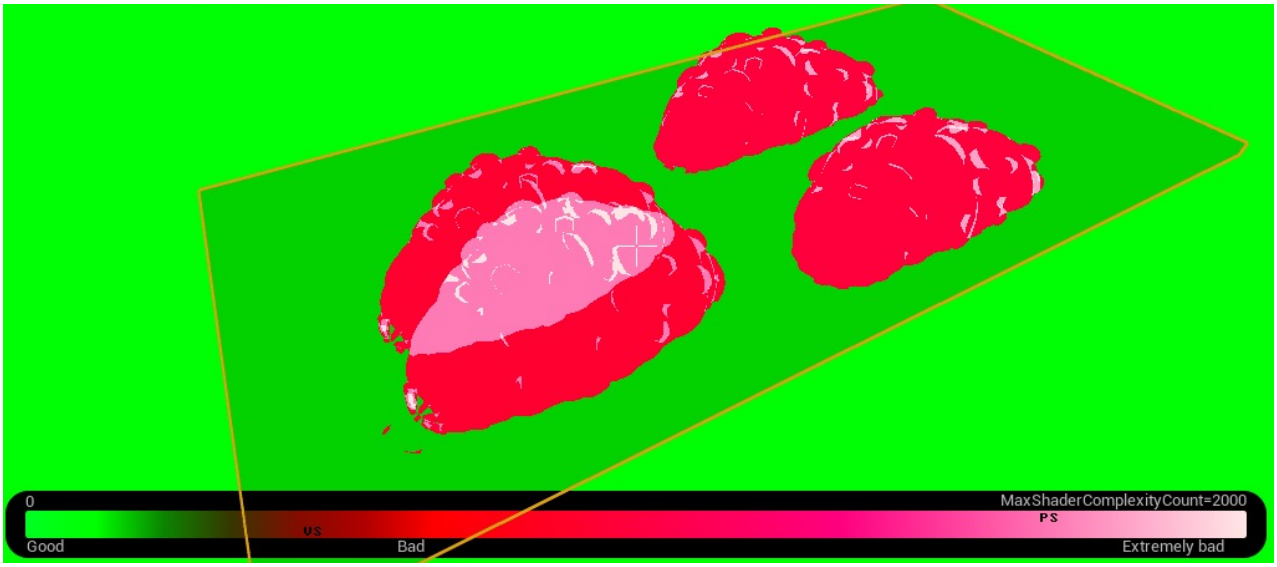


Figura 17. Shader Complexity del actor larva mar.

Como esta solución todavía generaba un consumo considerable de recursos y faltaba todo el contenido restante del nivel, se me solicitó buscar otra solución al problema, y me pidieron ver la posibilidad de resolverlo con un sistema de partículas, debido a que ya había experimentado con este anteriormente.

Por esas fechas un compañero con el que estaba trabajando me recomendó hacer uso del *Vertex Animation* (*Morph Target Animation* o *Blend Shapes*), el cual es un método de animación en el cual una versión deformada de una malla se almacena como una serie de posiciones de vértices dentro de un archivo de bitmap, donde cada *frame* de animación (que sería la deformación de la malla) es un vértice interpolado entre posiciones almacenadas. Siguiendo la documentación que me hizo llegar mi compañero y haciendo uso de *Blender* generé los 5 modelos con animación de la *larva mar*, obteniendo el modelo *".fbx"* de cada uno y dos archivos de bitmap uno *".bmp"* y otro *".exr"*, para importarlos al motor gráfico de *Unreal* como *blend* shapes para hacer uso de estos en un sistema de partículas. Además, que de este modo la complejidad del material y por consecuencia el cálculo computacional que se requiere es menor al método anterior.

Con todos estos elementos partí por generar un sistema de partículas usando como semilla la malla de cada uno de los *blend shapes* de la *larva mar*, a partir de la cual se iban a empezar a generar las partículas. Modificando las propiedades del sistema de partículas, logré hacer que se generaran a distintos tamaños, a cierta distancia las unas

de las otras, con un tiempo de vida correspondiente al tiempo con el que se generaban las nuevas partículas, y con un desplazamiento a lo largo del eje *Y* para dar la sensación de que era un entorno más “vivo”, esto se puede ver en la *Figura 18*.

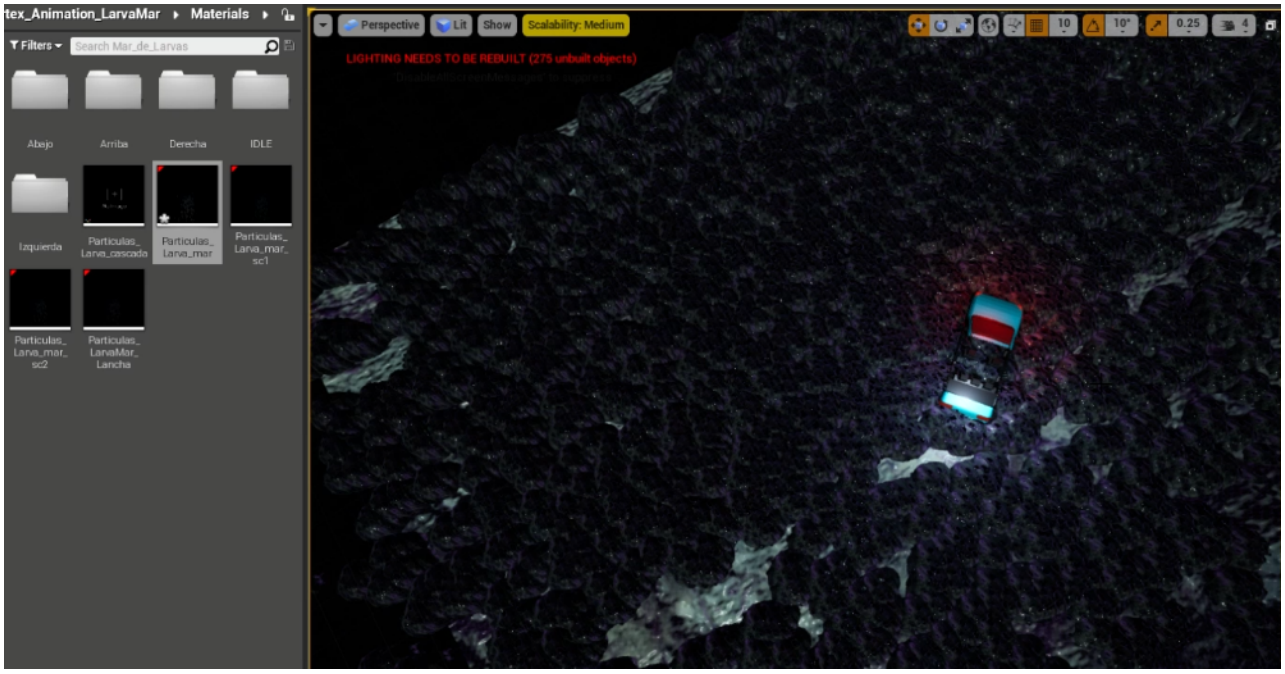


Figura 18. Prueba del Sistema de partículas larva mar dentro del entorno del nivel

Si bien este método del sistema de partículas tiene varias ventajas como el poder renderizar cientos de actores al mismo tiempo con un buen rendimiento y materiales óptimos, tiene la desventaja de que al ser un sistema de partículas cada partícula no posee las mismas propiedades que un objeto instanciado, en este caso alguna cápsula de colisión para evitar que los objetos se mezclen entre sí y como se genera su desplazamiento a través de darle diferentes parámetros y al momento de generarse es un movimiento aleatorio, es más difícil hacer que alguna partícula esté en algún punto en específico por lo que pueden llegar a mezclarse con geometría del nivel o con algún elemento no contemplado.

Debido a la ambición del proyecto en esta sección del demo en específico se requerían demasiados actores de *larva mar* en escena, por lo que me dispuse a buscar una solución óptima capaz de cargar tantos actores a la vez con el menor costo de rendimiento y memoria posible.

Investigando sobre los métodos de renderizado el tener que mandar a llamar un solo objeto que tiene una malla, uno o más materiales, física, colisiones, iluminación, posición, rotación y escala es un desperdicio cuando se quieren renderizar tantos objetos

similares al mismo tiempo, ya que se ejecuta un comando por llamado, ralentizando y saturando la memoria. De ese modo si se podía hacer sólo un llamado a un objeto dentro del cual se generaran todos los demás actores con sus materiales, mallas, físicas, colisiones y transformadas; si bien será un consumo de memoria mayor al del caso anterior, va a ser mucho menor que en caso de querer instanciar todos los objetos uno a uno saturando la memoria, porque sólo se harán unos cuantos llamados siendo que son varios actores combinados en un solo objeto.

Para esto hice un *blueprint* en el cual instanciar varios actores de *larva mar* en un arreglo, creé un par de arreglos donde los 5 tipos de *static mesh* de la *larva mar* creadas previamente con el *Vertex Animation* se iban a posicionar aleatoriamente en las coordenadas de un espacio "*X*" y "*Y*" (esto debido a que acomodé los actores larvas en forma de matriz de "*X*" por "*Y*" elementos), con una escala entre .35 y .8 y un valor de rotación en "*Y*" de 0 a 20 grados y de 180 a 360 grados en "*X*", parte de esto se puede apreciar en la *Figura 19*.

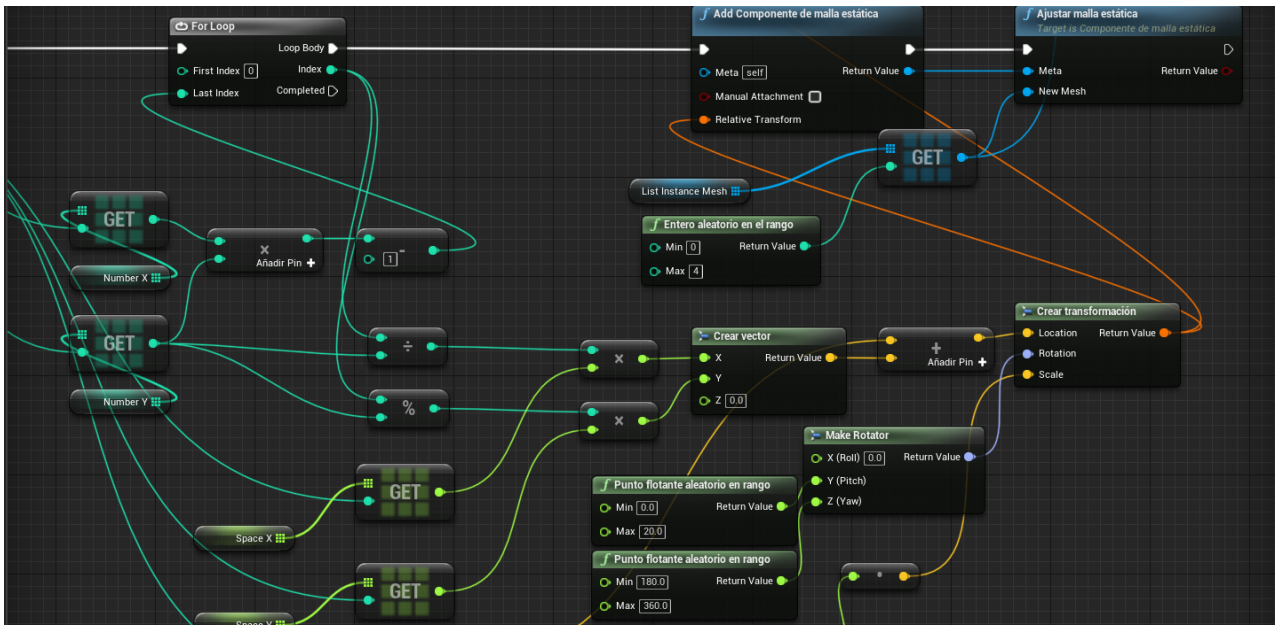


Figura 19. Blueprint del instanciador de larvas mar.

La decisión de darle tantas variables, es para posteriormente en tiempo de ejecución poder desde el *Engine*, modificar sencillamente los mismos parámetros para acomodar los actores *larva mar*, de mejor forma que se adapten al entorno deseado. Porque en un inicio sin estas variables el sólo hacer el llamado de varios actores dándoles un poco de distancia entre ellos daba una sensación de demasiada "artificialidad", cómo se ve en la *Figura 20* y el objetivo era lograr que el entorno (dentro del contexto artificial en que se desarrolla el videojuego) se viera natural o variado como se ve en la *Figura 21*.

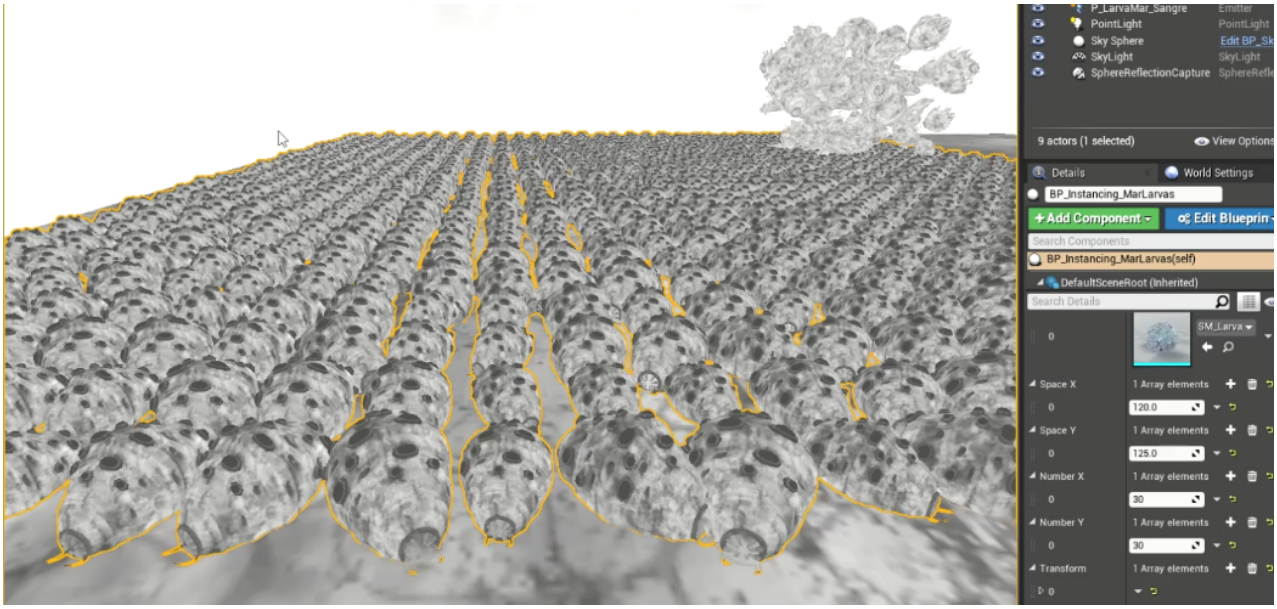


Figura 20. Instancia de actores larva mar alineados.

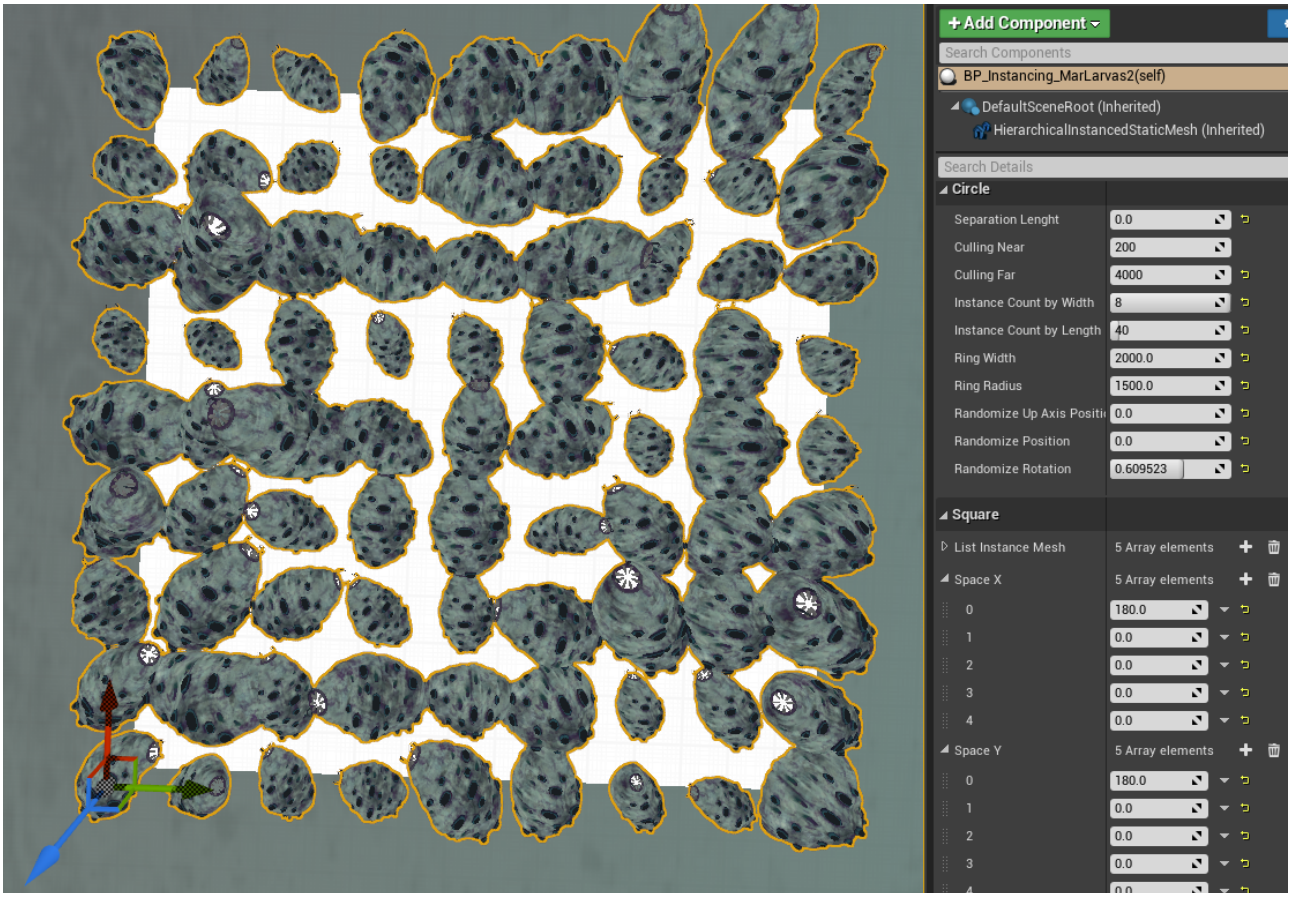


Figura 21. Instancia de actores larva mar con aleatoriedad.

Al mostrar esta solución, agradó al equipo (de momento), ya que contaba con la mayoría de los requisitos solicitados como incluir las 5 animaciones proporcionadas, abarcar la mayor área posible, que tuviera un buen rendimiento y que los actores no se encimaran (tanto) entre ellos, porque en este último punto al haber hecho el arreglo en que se instancian en forma de un rectángulo, todavía señalaban que existía cierta “artificialidad” en el acomodo de los actores, sin embargo, por cuestiones de tiempo se dejó como solución temporal en lo que nos proporcionaban los modelos finales del nivel.

Continuando con esto, me di a la tarea de revisar que otros sistemas principales de un videojuego hacían falta e inicié con el cambio de niveles. Para esto empecé haciendo pruebas entre el nivel del *mar de larvas* y el nivel de la *colmena*, esto creando un *TriggerBox*, dándole la función de *OnActorBeginOverlap* con la condicional de que, si el actor del jugador es el que entra dentro de esta “caja invisible” y se detecta que existe una colisión entre la geometría del actor jugador y el objeto caja, ejecutar la función *OpenLevel* para que se cambie a otro nivel.

Otro de los sistemas que faltaban y consideré pertinente ir creando era el sistema de vida, muerte y reaparición del jugador; de este modo con todos los niveles y mecánicas ya generadas se podría ir haciendo un primer aproximamiento a un *alfa* jugable de la demo. De modo casi intuitivo el motor gráfico de *Unreal* contiene un actor llamado *Kill Z Volume* en el cual al momento de que el jugador entra en el volumen de ese cubo invisible este es destruido y reaparece en el actor *Player Start* (la clase que indica donde el jugador va a aparecer o reaparecer cuando inicie el juego), así que con este actor se lograba cubrir todos aquellos espacios a los cuales el jugador no debería acceder o la muerte por altura del jugador. Sin embargo, los otros modos en que el jugador podría recibir daño y morir eran con los distintos enemigos por lo cual en el *blueprint* del jugador creé un par de variables que midieran los puntos de vida del jugador (*Health*), estableciendo esta variable en 100, colocándola en el evento de recibir daño y en caso de que el daño recibido se deje los puntos de vida del jugador igual o menor a 0, se destruye el actor del jugador para que este posteriormente reaparezca en el *Player Start* correspondiente, parte de este código se puede ver en la *Figura 22* donde se establece el daño por el actor *larva* y en la *Figura 23* donde se ve el código para la muerte por los actores en el *mar de larvas*.

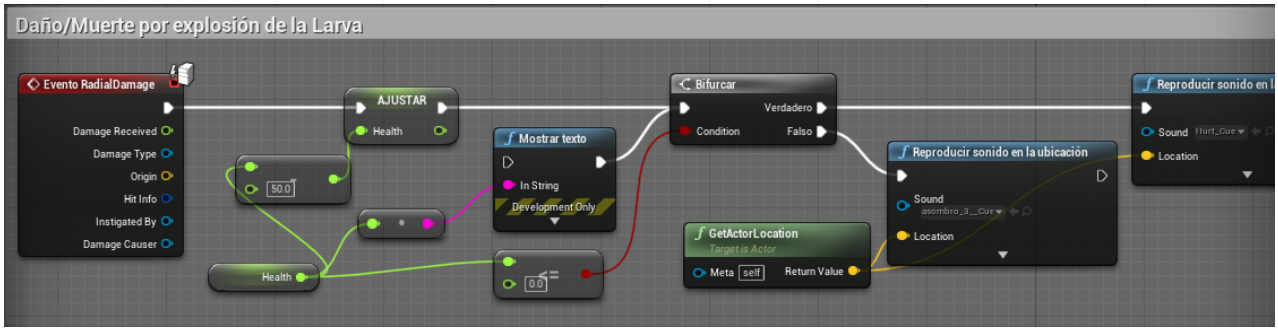


Figura 22. Blueprint de muerte por enemigos tipo larva.

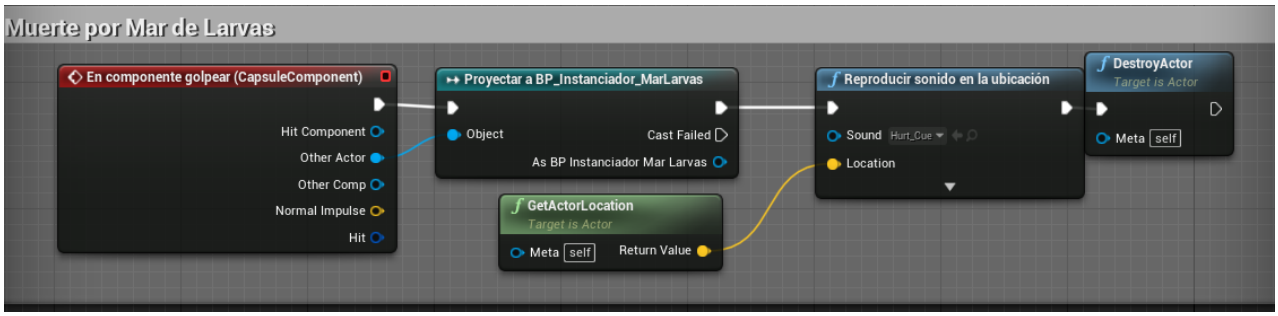


Figura 23. Blueprint de muerte por el mar de larvas.

Después de agregar y probar estas mecánicas básicas empecé, en conjunto con los demás desarrolladores, a hacer pruebas de la demo de principio a fin con lo que se tenía y nos percatamos de que había algo extraño cuando el jugador interactuaba en el escenario, se percibía como si este flotara ligeramente con respecto al suelo. Para entender porqué sucedió esto debo explicar que la mayoría de los objetos que interactúen entre sí deben poseer algún componente de colisión para que estos no se traspasen; lo que sucedió en lo antes mencionado es que el componente de colisión del jugador (que es en forma de cápsula) estaba ligeramente más grande en el eje "*Z*" que es por lo que se daba esta extraña interacción por lo que se ajustó el componente cápsula del jugador como se ve en la *Figura 24* donde de lado izquierdo se tenía una cápsula más corta y amplia debido a que el modelo del jugador en ese momento era más pequeño, en cambio de lado derecho como se cambió el modelo del personaje a uno un poco más alto se debió ampliar el largo de la cápsula y disminuir el diámetro de la misma.

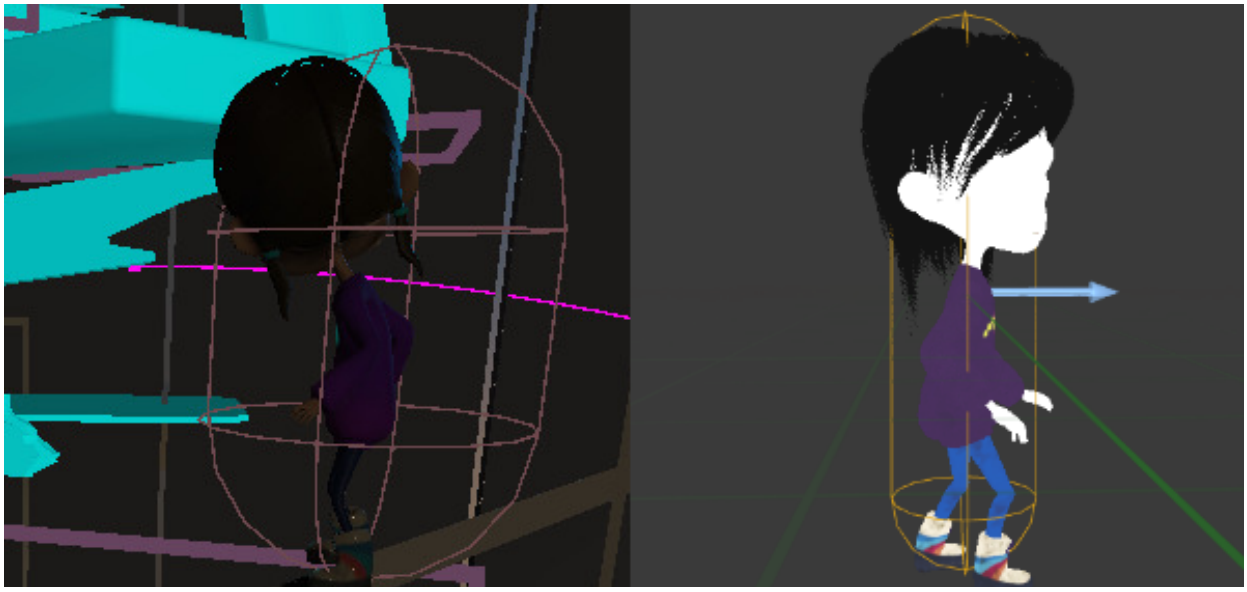


Figura 24. Cápsula de colisión original vs Cápsula de colisión modificada.

Si bien esto soluciona el problema de que el jugador ya no diera la sensación visual de estar flotando sobre el suelo, se generó el inconveniente de que al ajustar este componente descubrimos que ahora existían nuevos espacios por los cuales el jugador podía acceder y que no estaban contemplados en el acomodo original de las colisiones por lo cual se empezó a evaluar con detenimiento sección por sección de los niveles para reajustar las colisiones del entorno, en la *Figura 25* se muestra el modo de vista de colisión mostrando en color cyan las colisiones generadas por los propios modelos y las de color rosa las colisiones puestas desde el editor de *Unreal Engine*.

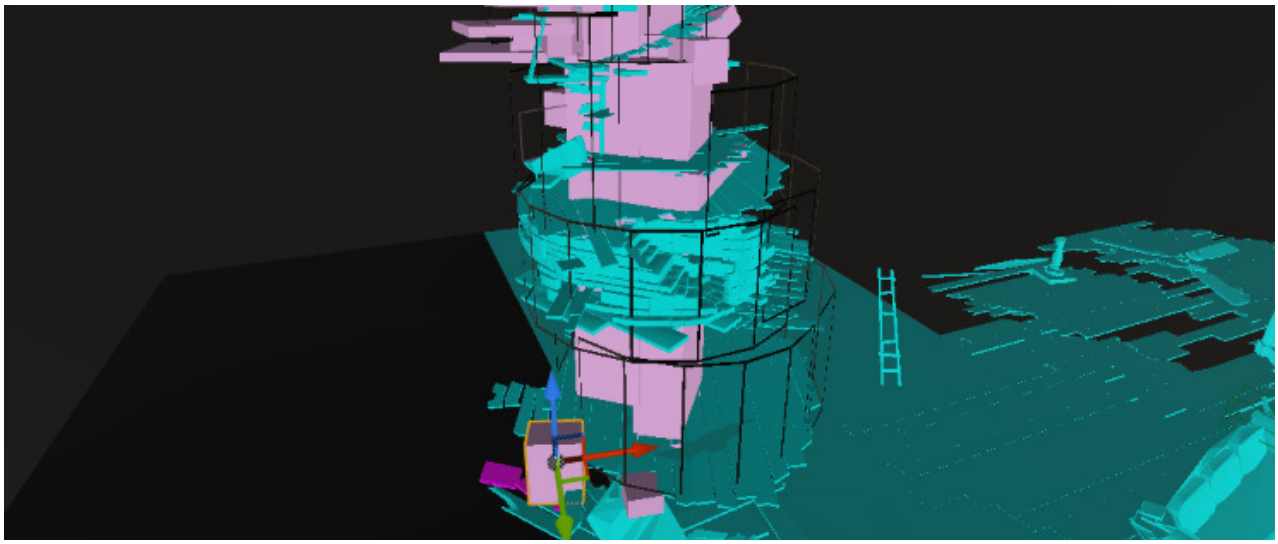


Figura 25. Proceso de reacomodo de colisiones en el nivel “Descenso”.

De manera simultánea a estar trabajando en todos estos reacomodos de colisiones en distintos niveles, me pidieron hacer una prueba con otro tipo de enemigo el *jefe final* con base al instanciador de actores en un solo llamado que había hecho originalmente para el actor *larva mar*. El requerimiento para este nivel es que, en él hay una pared caída por la cual se puede ver hacía el exterior y que en esta parte se lograra observar a un enjambre de actores del tipo *jefe final*, pasando por el exterior. Inicialmente modifiqué el *blueprint* que tenía para sustituirlo por la malla correspondiente, después de eso, modifiqué el acomodo que se tenía para que en lugar de que fuera un rectángulo de "*X*" por "*Y*" elementos, por una forma de toroide para añadir actores a lo largo del eje "*Z*" y dar esa impresión de que estaban volando por un enjambre. Además de esto aproveché para hacer pruebas y añadir un *Occlusion Culling* (que es un sistema de *UE4* donde los objetos más lejanos o no visibles para el jugador no se renderizan), esto se puede observar en la *Figura 26* donde se ve un semicírculo con los modelos instanciados, además de un par de parámetros para renderizar objetos a una distancia de entre 200 a 2000 unidades, con respecto a la cercanía con la cámara, tal y como se puede observar en la *Figura 27* donde al acercar la cámara se ven más modelos generados en total de los que se observan en la figura anterior.

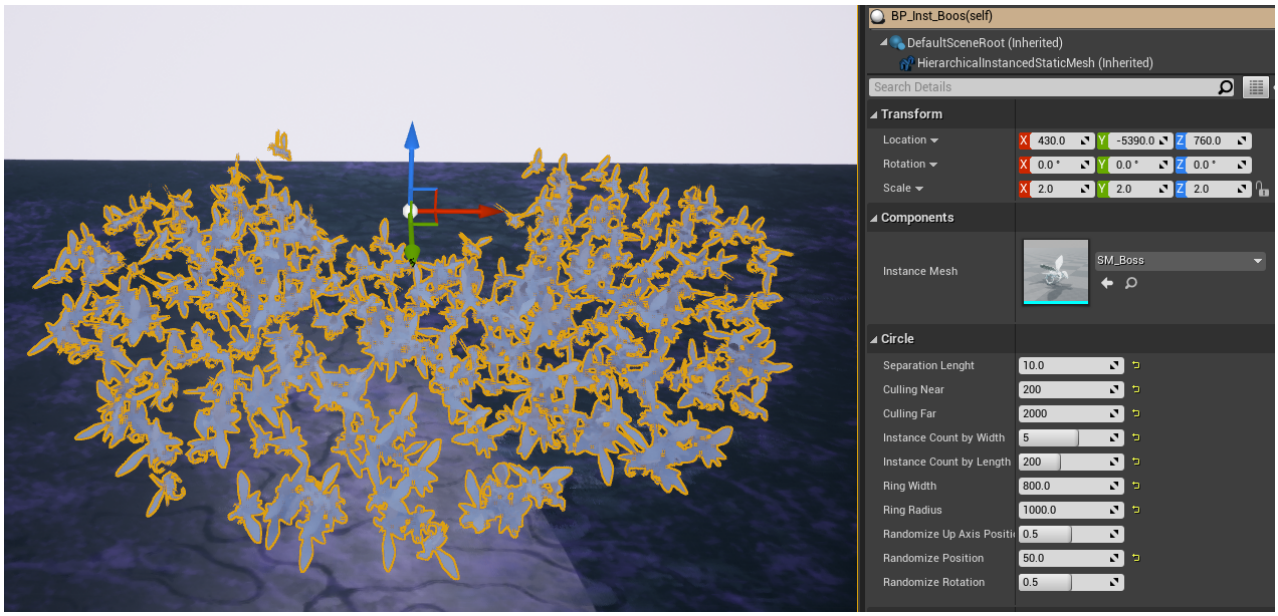


Figura 26. Instancia parcial de actores jefe final.

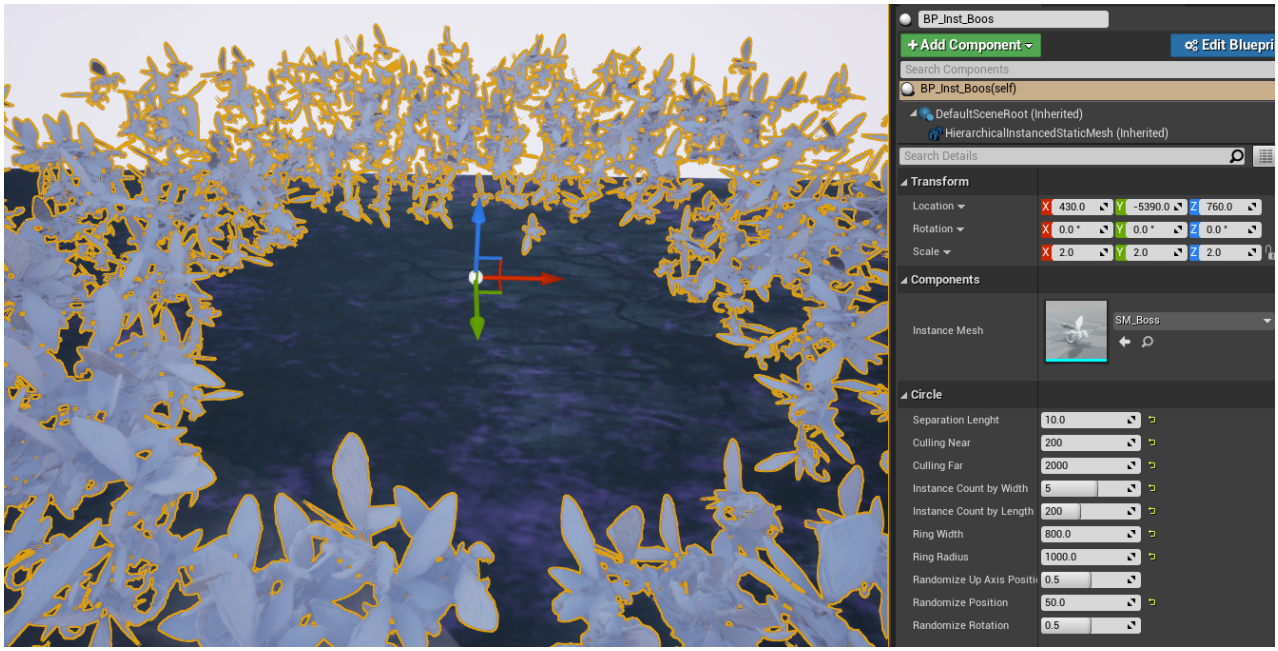


Figura 27. Instancia completa de los actores jefe final.

Mostré esta primera aproximación en una de las juntas semanales pero el resultado no terminó de convencer, así que sólo se quedó como otro experimento para optimizar el renderizado de muchos actores en tiempo real.

Para este momento del proyecto ya se había logrado implementar todos los niveles en un nivel maestro (*Level Master*) para con esto ir creando distintos puntos de carga y reaparición para el jugador y continuar acercándonos a ese primer *alfa* jugable del demo.

Buscando que más se podía añadir al demo se me ocurrió crear una breve cinemática en tiempo de juego, para una sección del demo en que debes ir por un objeto clave (un paquete de pilas) para poder avanzar de ese nivel. Para hacer esto me puse a investigar entre la documentación oficial de *Unreal* y los tutoriales de internet, hasta que encontré que *Unreal* tiene un *asset* llamado *Level Sequence*, con el que se pueden crear cinemáticas en tiempo de juego a través de un editor; te permite grabar y guardar a través de *key frames* (fotogramas clave) la manipulación de objetos, personajes y cámaras. Así que creé una secuencia de animación, en la cual al momento de que el jugador llegara a un punto específico del nivel, se activaría una colisión con una caja invisible y a partir de ahí la cámara partiera de esa posición, hiciera un recorrido sobre el trayecto a seguir, para después señalar el objeto clave (paquete de pilas) y la ubicación de la salida. En la *Figura 28* se puede observar parte del código que lleva a cabo la cinemática, mientras que en la *Figura 29* se ve la ventana asignada al *Level Sequence*

desde la que se puede modificar el comportamiento de los actores en el escenario, así como el de la cámara.

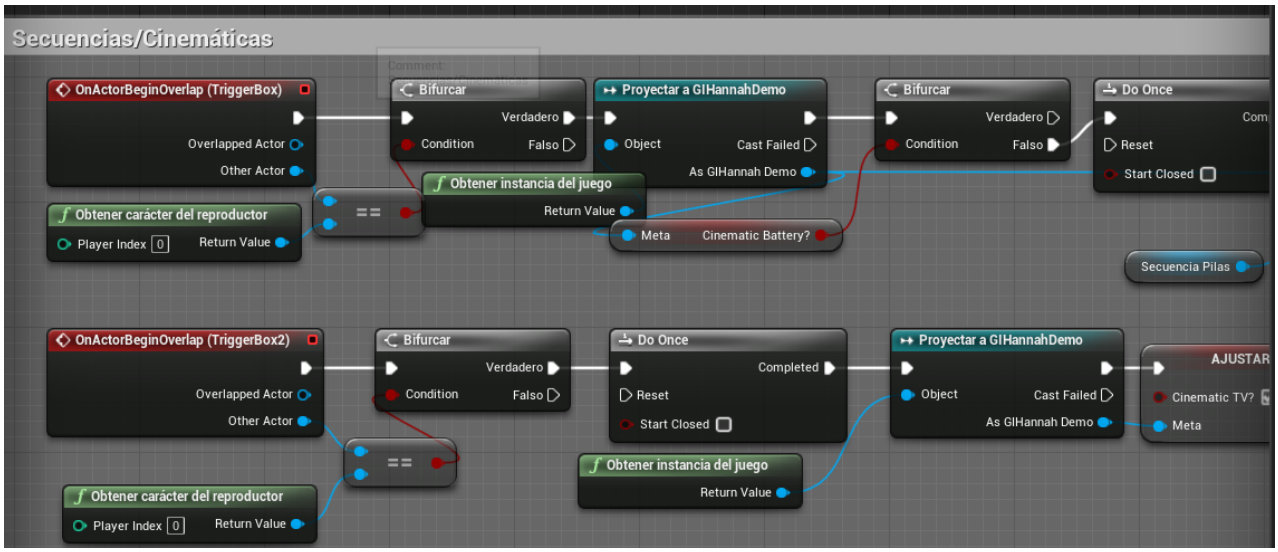


Figura 28. Level blueprint de la animación en el nivel Sala TV.

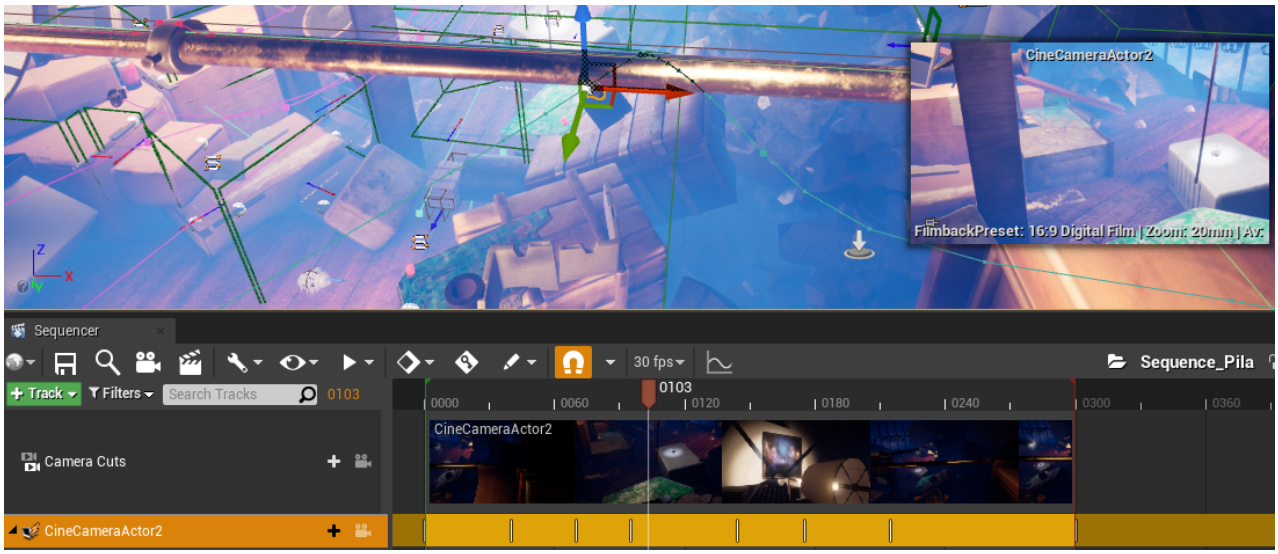


Figura 29. Editor de Level Sequence en el nivel Sala TV.

Para esta secuencia de animación se agregaron varias características como que el jugador no se moviera de posición mientras transcurre la misma, reproducir sólo una vez la cinemática para no generar una experiencia tediosa y repetitiva al usuario, ajustar la velocidad de transición de la cámara y buscar tomas donde resaltara el entorno creado en el nivel.

Al tiempo que yo trabajé en esto uno de los directores encontró otra forma para poder instanciar varios actores del tipo larva al mismo tiempo dentro de la escena, esto a través de la herramienta de *Foliage* que tiene *Unreal*, esta herramienta permite agregar actores de manera sencilla “pintándolos” sobre el terreno en una sola llamada de dibujo. Así que con esta herramienta y el *Vertex Animation* de los actores tipo *larva mar* se logró llegar a una mejor solución que la antes mencionada en este documento. En este momento del desarrollo ya se nos había brindado el *Landscape* (terreno) y el acomodo de objetos final de ese nivel, por lo que empecé a “pintar” sobre el terreno combinando las cinco animaciones diferentes de los actores *larva mar*; como se puede observar en la *Figura 30*, además de que la herramienta de *Foliage* incluye la opción de *Cull Distance* (similar al *Occlusion Culling* mencionado anteriormente) con la cual se pueden establecer parámetros para que a cierta distancia con respecto a la cámara los objetos se dejen de renderizar para liberar carga de memoria.

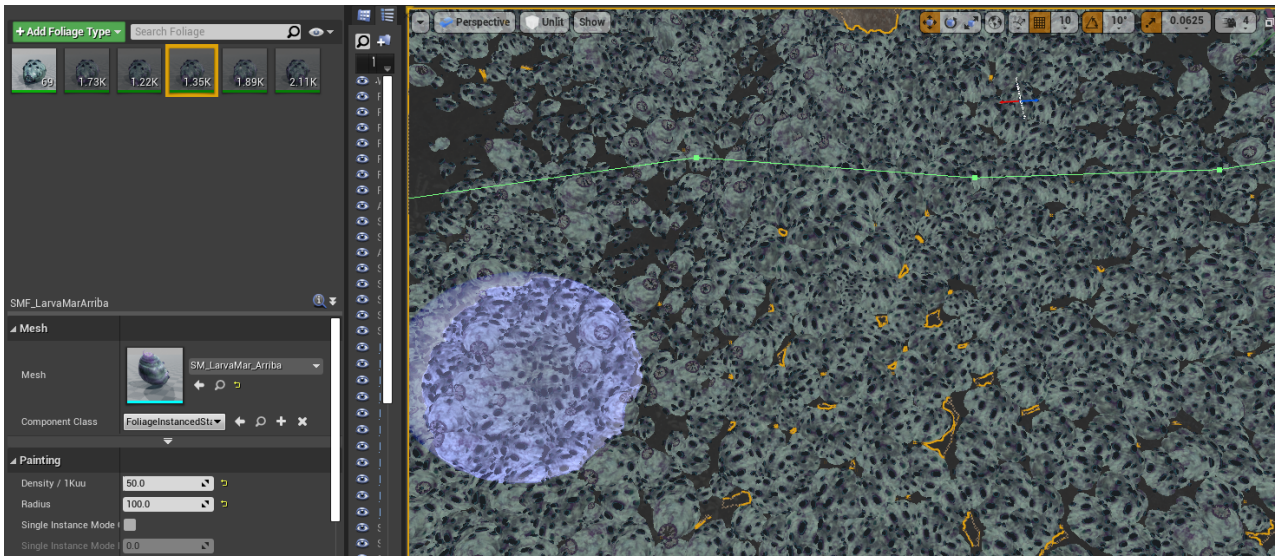


Figura 30. “Pintado” de actores larva mar.

Este nuevo resultado se quedó como la versión final. Después en este mismo nivel, se tenía que implementar otra mecánica de explosión de una onda. Para esto dividí la mecánica en dos, la explosión per se que generaría daño en un radio de distancia y el objeto destruible que se vería afectado por la explosión.

Para la explosión, en el actor de la bomba agregué un componente de *Radial Force*, con el que se puede aplicar una fuerza constante o un impulso dentro de las físicas del motor, para que al momento de detonar la bomba se creara este impulso simulando una explosión. Por otra parte, para el objeto destruible se tenía que crear una variante del *static mesh* original llamada *destructible mesh* la cual incluye componentes para

establecer en cuantos fragmentos se quiere partir el objeto y que al recibir algún impulso o daño físico se rompe. En la *Figura 31* se puede observar como una de las *destructible mesh* sale volando al centro del escenario, esto por lo que ya se explicó previamente de que este tipo de objetos resulta afectado por cualquier impulso físico del escenario.

Figura 31. Explosión en el nivel mar de larvas.

En este punto del desarrollo ya se tenían más mecánicas y actores enemigos desarrollados por lo que se debía agregar más modos de recibir daño de parte del jugador, pero en lugar de agregar más eventos al *blueprint* del jugador opté por generalizar el modo en que el jugador recibe daño, ya que con el evento *AnyDamage*, el jugador recibe daño de cualquier actor que lo esté generando, en este caso los enemigos, que ya tenían su propio modo de generar daño al jugador dentro de su respectivo blueprint, dentro de la *Figura 32* se puede observar el código que terminó englobando todos los tipos de daño posibles que el jugador podría recibir.

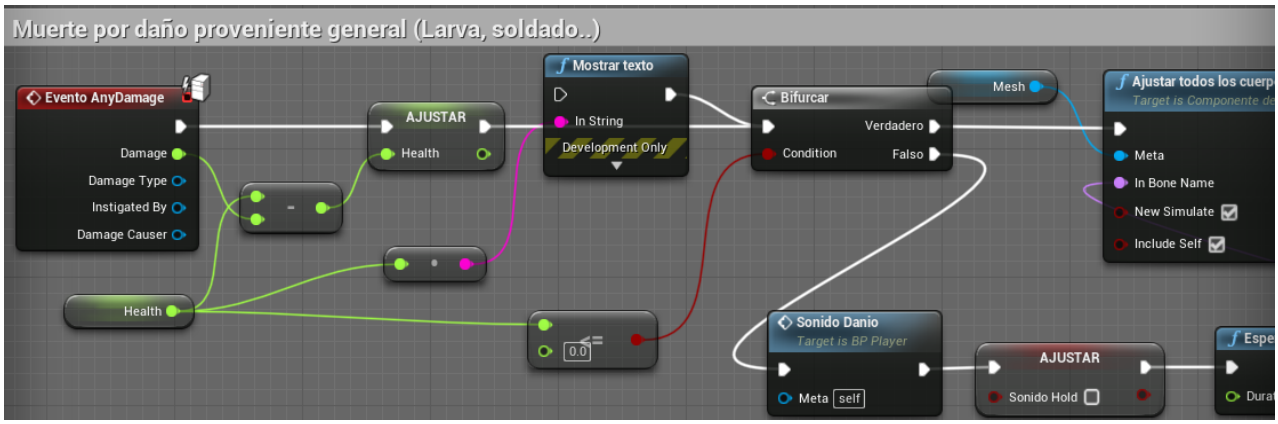


Figura 32. Actualización del método en que el jugador recibe daño.

Posteriormente a esta actualización, implementé un sistema de “cambio de nivel con una llave en específico”, porque en el nivel *Sala TV*, se necesitaba que el jugador se lleve cargando de ese nivel el objeto de paquete de pilas (la llave específica) debido a que es un objeto clave para el siguiente nivel. Para esto establecí las variables destinadas a la “puerta” y a la “llave”, la puerta siendo unas tablas que agregué al tubo por el cual se cambia el nivel para impedir el paso al jugador a no ser que tenga la llave, estas tablas eran objetos del tipo *destructible mesh* para que sean rompibles, como se puede observar en la *Figura 33*; y la llave siendo el paquete de pilas que el jugador debe cargar, como se ve en la *Figura 34*, dándole al objeto propiedad de *RadialForce* para que este aplique un peso sobre las tablas haciendo que se rompan y logrando así que el jugador progrese de nivel con el objeto clave, lo cual se puede ver en la *Figura 35*, donde el jugador cargando el paquete de pilas consigue romper las tablas que estaban en el suelo logrando acceder al siguiente nivel y activando la cinemática correspondiente.

Figura 33. Jugador en la “puerta” sin poder avanzar.

Figura 34. Jugador recogiendo la “llave”.

Figura 35. Jugador abriendo la “puerta” con la “llave”.

Con esto implementado y funcional se me pidió que buscara información sobre un sistema de *ragdoll* (muñeca de trapo) para la muerte del jugador, ese nombre se le da porque las físicas normales del actor se ven reemplazadas por una física de muñeca de trapo que son normalmente usadas en varios videojuegos, como Halo 2 (2004), Happy Wheels (2010), Gang Beasts (2014). Para implementar esto dentro de *Unreal*, es necesario que el personaje posea físicas para poder generar la simulación correspondiente en el evento de morir, en la *Figura 36* se puede observar las cápsulas de colisión internas del personaje.

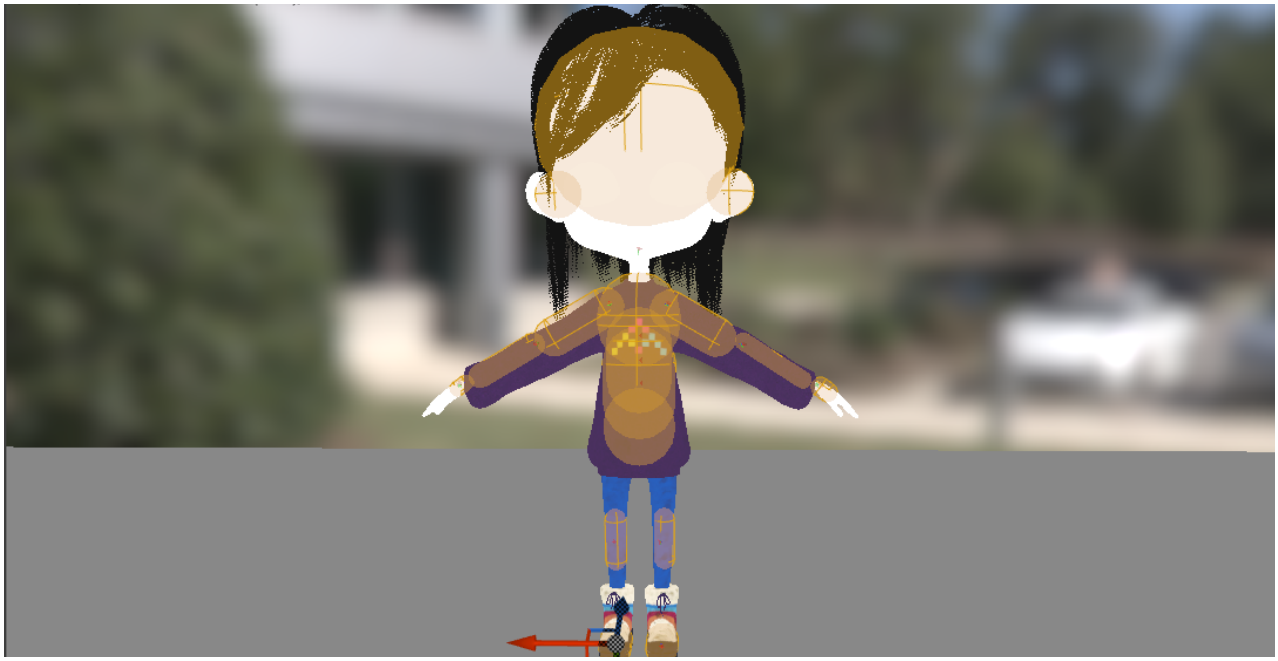


Figura 36. Físicas del personaje.

Como se puede observar en la figura anterior el sistema de físicas del personaje no abarcaba toda la malla del personaje, lo cual provocó que al momento de implementar el sistema de *ragdoll* se deformara la malla, como se ve en la *Figura 37*, ya que sólo tomaba en cuenta los puntos donde existían las cápsulas de colisión.

Figura 37. Simulación de ragdoll del personaje.

La solución para que la malla del personaje no se deforme como en la imagen anterior es ajustar las físicas del mismo, pero eso se debe hacer desde el programa de origen

donde se modeló y/o animó el modelo; por lo que se logró implementar el sistema de *ragdoll* pero el resultado no fue el óptimo.

Para este punto del desarrollo nos enfocamos en corregir y ajustar detalles de la experiencia de juego, como ajustar colisiones, *Blocking Volume, kill ZVolume,* puntos de carga, entre otros; esto debido a que por fechas de fin de año íbamos a tener un descanso, y el objetivo era generar un *alfa* del proyecto jugable para compartirlo con los demás miembros de la empresa y recibir su retroalimentación; así sucedió y el primer *build* del proyecto terminó pesando 12.4 GB.

Regresando del periodo decembrino, con base a la retroalimentación recibida por la empresa empecé a corregir errores, ajustar volúmenes de colisión, volúmenes de muerte, pulir mecánicas, pulir comportamientos de enemigos, y buscar en la documentación por métodos para optimizar tiempos de carga, tiempos de renderizado y el peso del proyecto. Ejemplo de esto es el ajuste que se le hizo a las colisiones del nivel "Descenso" en la *Figura 38*, el cual se puede comparar con la *Figura 25* de este mismo documento.

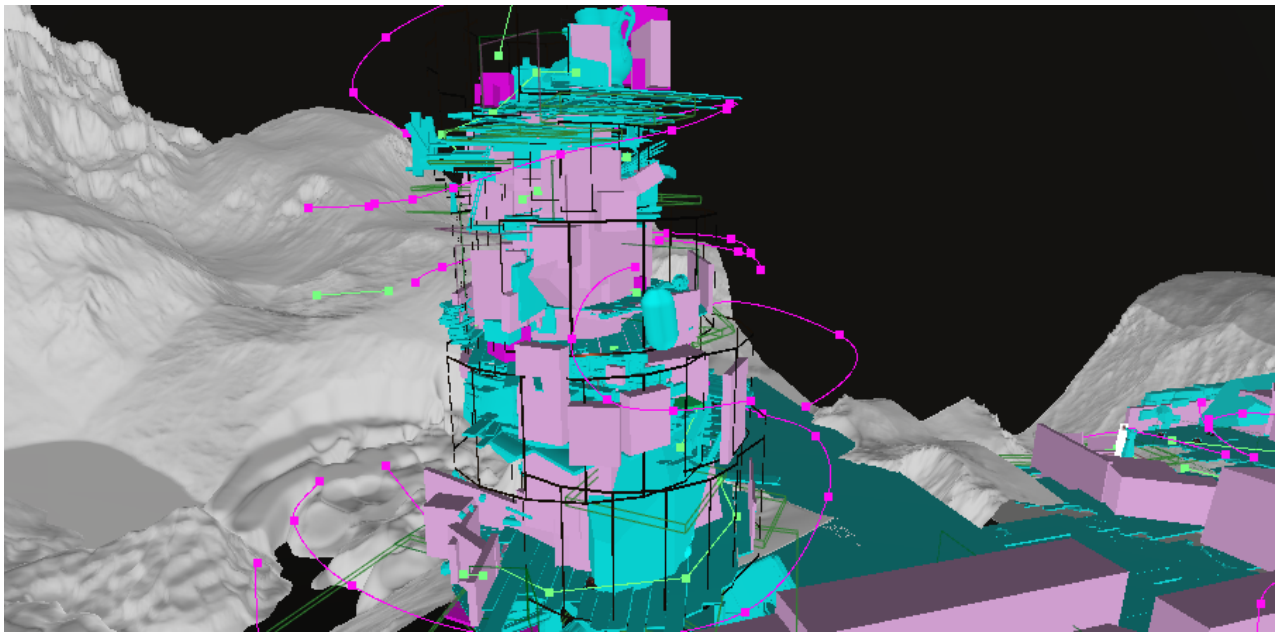


Figura 38. Ajustes de colisión al nivel del "Descenso".

Además de los ajustes entre niveles, en la interfaz de usuario del menú añadí una pantalla con los controles de juego para teclado y control de *Xbox*, como se ve en la *Figura 39*, esto con un *Widget blueprint* (que es con el que se programa la interfaz de usuario dentro de *Unreal*).

Figura 39. Pantalla con los controles de juego para teclado.

Continuando con estos ajustes, agregué una pantalla de apertura que me brindaron para que se proyectara al abrir el ejecutable del juego, donde se muestra el logo del motor gráfico, el logo de la empresa y un fondo de pantalla del juego. Junto con esto también investigué el modo de añadir un ícono personalizado al ejecutable del proyecto, con las dimensiones necesarias, las cuales solicité y agregué al proyecto, como se ve en la *Figura 40*.

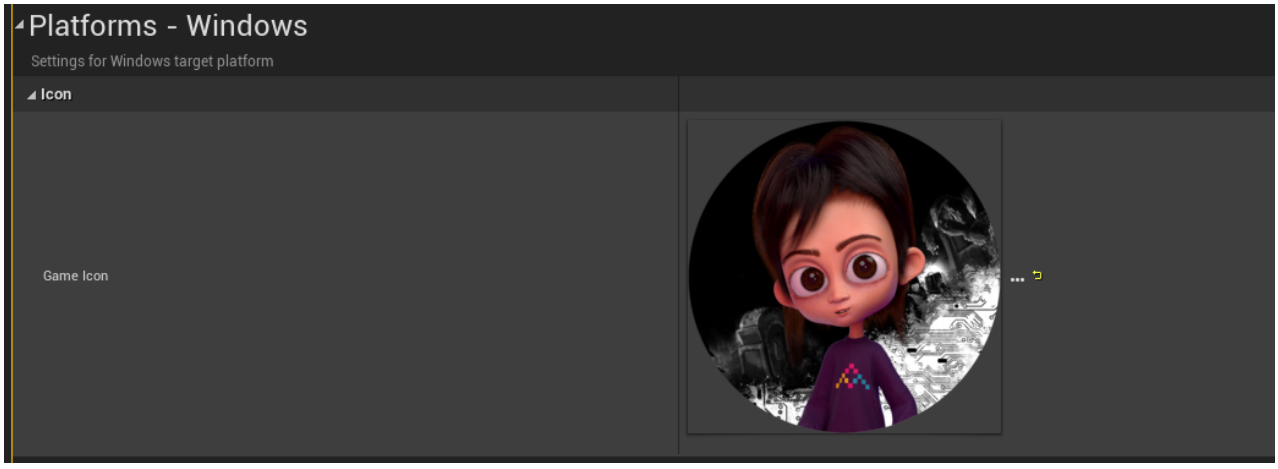


Figura 40. Ícono para el ejecutable en Windows del proyecto.

En el lapso de realizar todos estos ajustes, se realizó otro *build* del proyecto que sería una versión *Beta* de este, y del mismo modo que la versión *alfa*, se distribuyó entre los compañeros de la empresa para que lo probaran y nos dieran su retroalimentación al respecto. Se logró reducir el peso de estos ejecutables a alrededor de 4.2 GB.

Después de varios ajustes generé otra versión ejecutable y como de tiempo atrás venía investigando la documentación del proceso para subir el ejecutable a *Steam*; además de esto como el juego ya empezaba a verse cercano a su versión final la empresa decidió una fecha de salida del demo para publicarlo en la plataforma de *Steam*. Con la fecha decidida me coordiné con uno de los directores para realizar el proceso de subir el ejecutable a *Steam*. Para esto con anterioridad la empresa ya había creado una página de la tienda del juego, llenado todos los datos solicitados, registrado la propiedad intelectual de la idea y generando el aviso de privacidad.

Para subir un juego a *steam*, se necesita descargar el *SDK* (kit de desarrollo de *software*) de *Steamworks* (set de herramientas de *steam*), y *SteamCMD* (intérprete de comandos del cliente de *steam*), se debe extraer la carpeta del SDK de Steamworks en una ruta específica dentro del motor gráfico del juego, esta ruta se puede ver en la *Figura 41*.

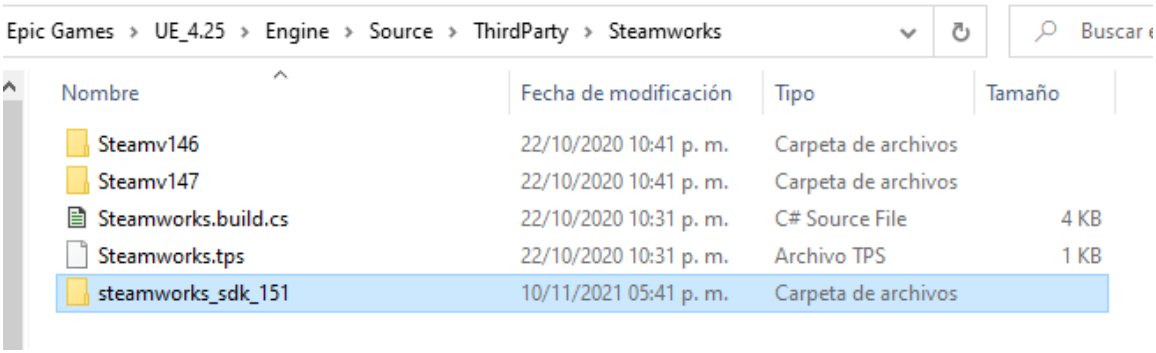


Figura 41. Ruta donde extraer el sdk.

Posteriormente se deben buscar y modificar un par de scripts "*.vdf*" (Valve Data File) con el *ID* que te genere la página de la tienda de *Steamworks*, así como especificar la ruta donde se encuentra la carpeta del ejecutable del proyecto. Después de eso se debe iniciar *SteamCMD*, donde a través de líneas de comando accedemos a la cuenta y para subir el proyecto a su plataforma se da el comando "*run_app_build<ruta donde se encuentran los scripts .vdf>*" (este mismo comando funciona para subir una actualización del juego). Para verificar que el proyecto se haya subido con éxito se revisa desde la página de *Steamworks*,en la *Figura 42* se puede ver una imagen alusiva a como se muestra en la página de *Steamworks*.

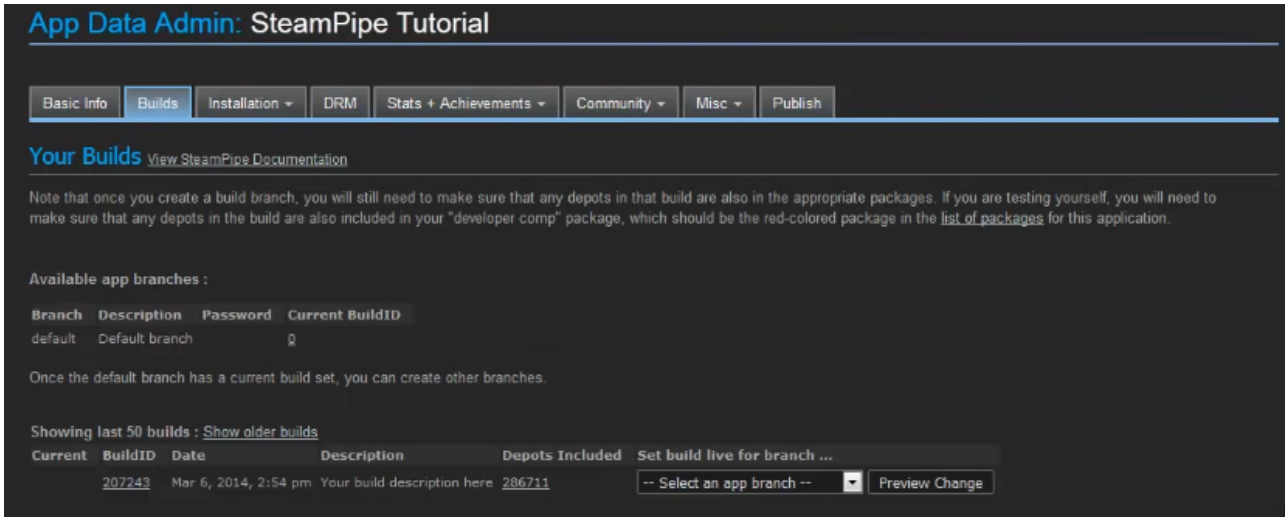


Figura 42. Imagen alusiva a como se muestra el ejecutable del juego dentro de Steamworks.

Habiendo realizado el proceso anteriormente mencionado y los últimos ajustes pertinentes al proyecto el miércoles 3 de marzo del 2021 se lanzó de manera oficial el demo del videojuego *Hannah The Game*.

Posteriormente al lanzamiento sugerí la opción de realizar una localización del demo a otros idiomas porque el mismo en su lanzamiento salió en el idioma inglés. A la empresa le pareció una buena idea por lo que me di a la tarea de investigar cómo hacerlo e implementarlo al español en un inicio.

Encontré que *Unreal* tiene la ventana de *Localization Dashboard*, donde viene una sección de culturas en la cual puede importar automáticamente el texto del proyecto, esto para agregar una nueva cultura (lengua) y hacer la traducción, en la *Figura 43* se puede observar el tablero de culturas para localización de *Unreal*.

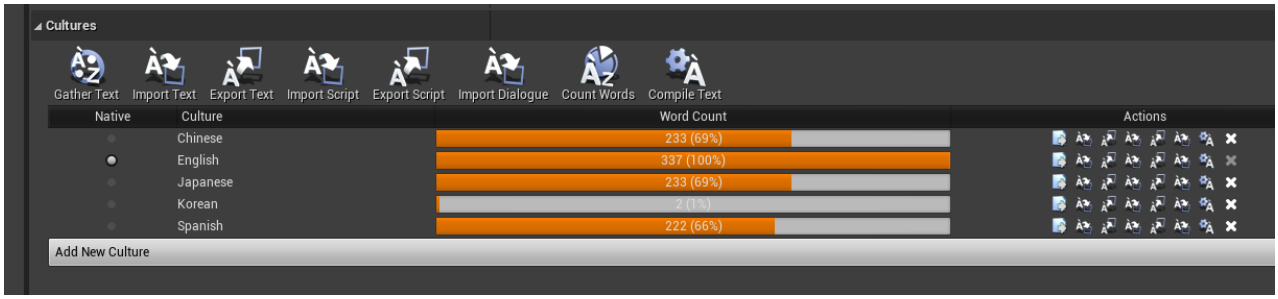


Figura 43. Tablero de culturas de localización.

En un inicio agregué el idioma de español y empecé a editar las traducciones para ese idioma, esto me desplegó una nueva ventana en la cual manualmente empecé a traducir los textos del proyecto, esto se puede observar en la *Figura 44*.

Figura 44. Localización al español.

Ahora que ya tenía el texto traducido, a través del *Widget blueprint* que se tenía para las opciones de desempeño, agregué otra opción para modificar el idioma en la que en un arreglo de elementos llené los campos con los idiomas que se tenían traducidos y a los que se tenía pensado traducir, como se puede observar en la *Figura 45*.

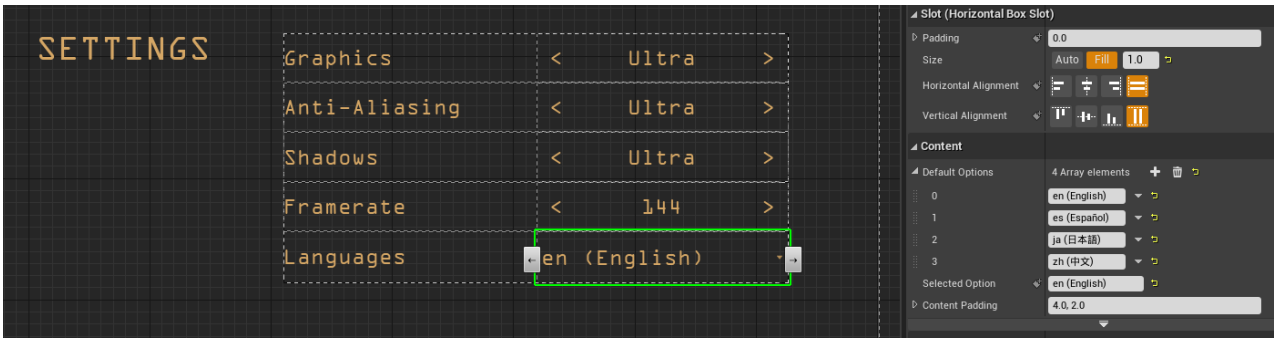


Figura 45. Opción de cambio de idioma.

Con la función de *Set Current Language* se cambiaba el idioma del juego, sin embargo, esto también modificaba el idioma del motor, por lo que agregué una función de *Get Substring* para que sólo tomara los valores de una cadena de caracteres, especificando

que tome los dos primeros caracteres de la cadena para después darle estos valores a la función de *Set Current Language*, para que con esto modificara el idioma en tiempo de juego y no dentro del motor gráfico.

Y posteriormente tuve que crear una nueve fuente (*Font*) en la cual añadiera todos los caracteres que no tenía la fuente original como los acentos o la “ñ”, para el caso del español, o los caracteres de *Hiragana* y *Katakana* en el caso de la traducción al idioma chino y japonés, en la *Figura 46* se muestran las fuentes que se agregaron y lo que engloba.

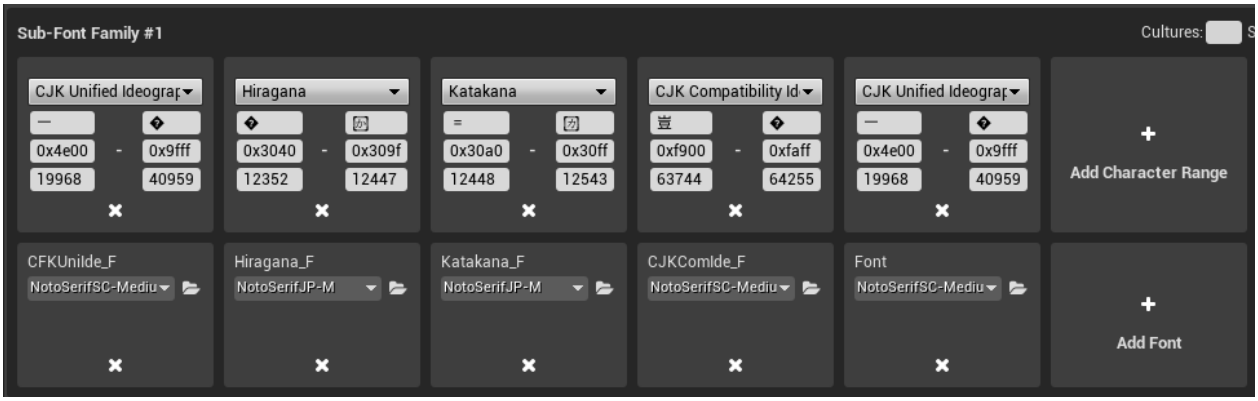


Figura 46. Creación de nueva fuente de caracteres.

Esta misma fuente la sustituí en un material que desplegaba texto en ciertas secciones del juego para que al cambiar de idioma en el texto se viera reflejado también en este texto flotante y con esto poder traducir los textos flotantes a lo largo del recorrido. Este material se puede observar en la *Figura 47* con los caracteres en blanco dentro de la previsualización.

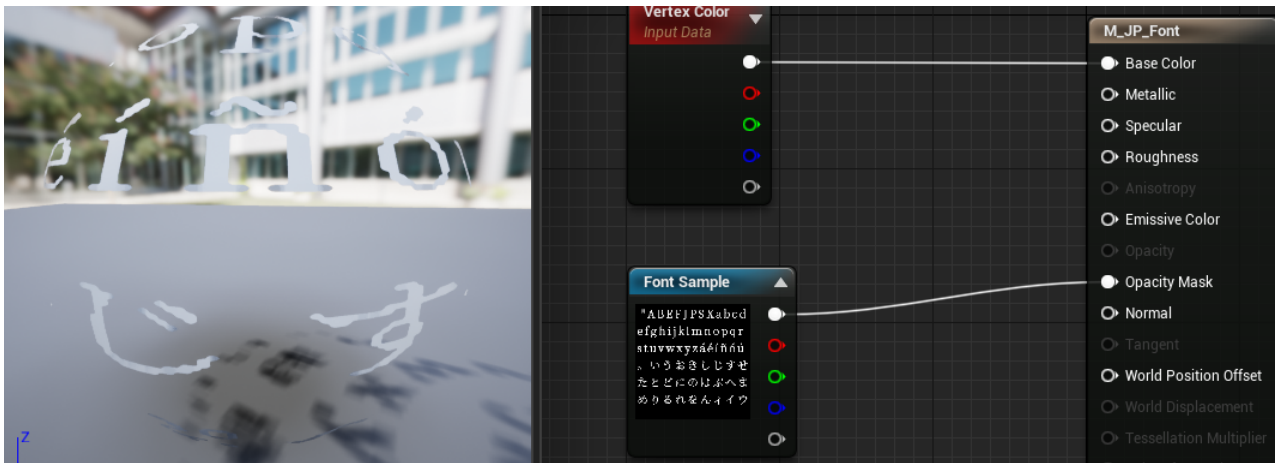


Figura 47. Modificación del material de fuentes con los nuevos caracteres.

9. Resultados obtenidos.

En un inicio se logró el objetivo principal que es subir la demo del videojuego a la tienda en línea de steam, https://store.steampowered.com/app/1466870/Hannah/, el jueves 4 de marzo de 2021.

Una de las razones de porque se creó en un inicio un producto demo de un videojuego fue para buscar distribuidoras o inversores que crean en el proyecto y apoyen a su desarrollo, para esto fue que sugerí semanas después de la salida oficial del demo de realizar las localizaciones a idiomas como el chino y japonés, porque dentro de las analíticas que da *steam* a los desarrolladores, la empresa notó que existe interés en el mercado asiático por el producto.

Para la empresa que con anterioridad había plasmado su forma de contar historias al igual que mostrar las capacidades y calidad de trabajo, esto a través del cortometraje, *iO Inner Self* (2018) y *Kizuato* (2019), el adentrarse en el mundo de los videojuegos presentaba un nuevo reto de contar una historia de otro modo, así como un nuevo entorno de trabajo para continuar creando contenido de calidad.

Además de que les agradó nuestro desempeño con el proyecto por lo cual nos contrataron y abrieron el área de innovación dentro de la empresa, la cual busca crear distintos tipos de proyectos estando a la vanguardia de los avances tecnológicos; desarrollando actualmente proyectos en Realidad Aumentada (AR), un juego basado en *iO*, el primer cortometraje hecho en Realidad Virtual (VR), espacios virtuales en 3D con el uso de Mozilla Hubs, arte digital, entre otros.

10. Conclusiones.

Como ya mencioné en el punto anterior se logró el objetivo de elaborar el producto demo de un videojuego y subirlo a una plataforma de distribución digital de videojuegos, esto en un periodo de aproximadamente 8 meses desde que se hizo el primer contacto con la empresa hasta que se subió a la plataforma de Steam. Personalmente este proyecto fue una consolidación de los conocimientos que adquirí en los años cursé la carrera de ingeniería en computación en complemento con lo aprendido en SODVI, y poder experimentar en el mundo laboral el estar trabajando en un proyecto de estas dimensiones a nivel profesional.

El colaborar en el desarrollo de un videojuego mexicano me presentó otro panorama sobre cómo es la industria de desarrollo desde dentro, las horas de arduo trabajo que conlleva para llegar a un resultado final. Además, el trabajar a la par que me encontraba cursando los últimos semestres de la carrera en medio de una pandemia global me sirvió

para empezar a distribuir mis horarios de un modo óptimo para cumplir con ambas responsabilidades. Aprendí a ser más autodidacta para aprender y desarrollar en periodos “cortos”, aprendí a proponer ideas y soluciones que considero sean de ayuda para el desarrollo de un proyecto, a tener una mente abierta y no cerrarme a que algo sólo se puede resolver de un modo, a no tener miedo de no hacer un proyecto por no conocer al 100% la herramienta de trabajo (a pesar de conocer las bases teóricas) y a estar a disposición de lo que se necesite realizar por un bien mayor o beneficio común.

Si bien, a nivel profesional apenas voy empezando y queda un largo camino lleno de aprendizajes, las horas de investigación, de prueba, de parcheo, de risas, de desvelos, dieron como resultado un proyecto del cual estoy orgulloso y agradecido de haber formado parte.

11. Bibliografía y Referencias.

- “Unreal Engine”, Recuperado de: https://www.unrealengine.com/en-US/
- UnrealLou, “Little Nightmares Camera Research”, *Unreal Engine Forums*, Jul. 2017. [En línea]. Recuperado de: https://forums.unrealengine.com/t/little-nightmares-camera-research/96448 (Consultado en: 16-07-2020)
- “Camera”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Framework/Camera/ (Consultado en: 16-07-2020)
- “Using Cameras”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/UsingCameras/ (Consultado en: 16-07-2020)
- KITATUS, “[TUTORIAL] Unreal Engine 4| Devil May Cry/ Resident Evil Camera System”, *KITATUSxGAMEDEV*, Feb. 2019. [En línea]. Recuperado de: https://medium.com/kitatus/tut-unreal-engine-4-devil-may-cry-resident-evil-camera-system-de0a43507b0 (Consultado en: 16-07-2020)
- “UE4 camera system análisis”, *ProgrammerSought*, [En línea]. Recuperado de: https://www.programmersought.com/article/20885504473/ (Consultado en: 16-07-2020)
- “Mapping the inputs”, *Packtpub*, [En línea]. Recuperado de: https://subscription.packtpub.com/boo-k/web-development/9781785883569/1/ch01lvl1sec12/mapping-the-inputs (Consultado en: 16-07-2020)
- “Visibility and Occlusion Culling”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/RenderigAndGraphics/VisibilityCulling/ (Consultado en: 16-07-2020)
- Unreal Engine, “Understanding Culling Methods | Live Training | Inside Unreal”, Ene. 2019. [En línea]. Recuperado de: https://youtu.be/6WtE3CoFMXU (Consultado en: 16-07-2020)
- Mathew Wadstein, “WTF Is? Volume - Cull Distance in Unreal Engine 4”, Dic. 2015. [En línea]. Recuperado de: https://youtu.be/g0ML7oJll3w (Consultado en: 16-07-2020)
- Timothy Hobson, “PBF: Brief Overview of Occlusion Culling”, Feb. 2016. [En línea]. Recuperado de: https://youtu.be/6MhsdwTW3SQ (Consultado en: 16-07-2020)
- Jayanam, “Unreal Engine 4 Animation Blueprints Tutorial”, Jul. 2016. [En línea]. Recuperado de: https://youtu.be/6ow796lR8vo (Consultado en: 16-07-2020)
- DevSquad, “Animation Blueprints - #9 Unreal Engine 4 Animation Essentials Tutorial Series”, Abr. 2018. [En línea]. Recuperado de: https://youtu.be/GiCuPdgQN9Y (Consultado en: 16-07-2020)
- Ryan Laley, “Unreal Engine 4 Tutorial - Animation Pt.1 Character Setup”, Feb. 2019. [En línea]. Recuperado de: https://youtu.be/9Hr4ZzevBcA (Consultado en: 17-07-2020)

- Ryan Laley, “Unreal Engine 4 Tutorial - Animation Pt.2 Animation Blueprint”, Mar. 2019. [En línea]. Recuperado de: https://youtu.be/4uomQr2giS0 (Consultado en: 17-07-2020)
- AstrumSensei – Game Dev, “Adding Enemy Character Model & Animations - Unreal Engine 4 Hack & Slash RPG #17”, Mar. 2020. [En línea]. Recuperado de: https://youtu.be/Lau4aLFHqI4 (Consultado en: 20-07-2020)
- Mathew Wadstein, “WTF Is? AI: Play Animation Task Node in Unreal Engine 4 (UE4)”, Mar. 2016, [En línea]. Recuperado de: https://youtu.be/94WLK3AGZ7Y (Consultado en: 21-07-2020)
- Brad Code Tips, “UE4 - NPC Random Animation Behavior Tree Task”, Nov. 2019. [En línea]. Recuperado de: https://youtu.be/byW3833iG0E (Consultado en: 21-07-2020)
- Ben Ornmstad, “UE4 Tutorial | Spawn Waves of Enemies in Increasing Numbers”, Mar. 2017. [En línea]. Recuperado de: https://youtu.be/6FmUGF1trlY (Consultado en: 27-07-2020)
- UE4Docs, “Gameplay How-to: Spawn / Destroy Actors with Blueprints”, Jun. 2017. [En línea]. Recuperado de: https://youtu.be/uEzFpMt6mxo (Consultado en: 27-07-2020)
- Markom3D, “UE4 - Spawn an Object in a Random Location - Unreal Engine 4 Blueprints Tutorial” Oct. 2018. [En línea]. Recuperado de: https://youtu.be/IjFUqverTio (Consultado en: 27-07-2020)
- Code cavern, “Advanced Spawn System Tutorial - Unreal Engine 4 Blueprint”, Jun. 2019. [En línea]. Recuperado de: https://youtu.be/hj6usdPqRVA (Consultado en: 27-07-2020)
- MBC, “12 Spawning enemy waves”, Nov. 2017. [En línea]. Recuperado de: https://youtu.be/d5RdSv6l84c (Consultado en: 27-07-2020)
- GomVo Tutoriales, “Unreal Engine 4: Tutorial oleadas RESPAWN enemigos (español)”, Mar. 2017. [En línea]. Recuperado de: https://youtu.be/9DBcDN28szk (Consultado en: 27-07-2020)
- Sheenweedy, “UE4 UV Animated Rats”, *imgur.* Ago. 2020. [En línea]. Recuperado de: https://imgur.com/gallery/Bzyi18s (Consultado en: 10-08-2020)
- Veloz Jorge, “Vertex animation Tool”. (Consultado en: 18-08-2020)
- Josh_Bogart, “Vertex Animation Script for Blender 3D Users”, *Unreal Engine Forums,* Feb. 2017. [En línea]. Recuperado de: https://forums.unrealengine.com/t/vertex-animation-script-for-blender-3d-users/85580 (Consultado en: 18-08-2020)
- “Vertex Animation Tool - Timeline Meshes”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/AnimatingObjects/SkeletalMeshAnimation/Tools/VertexAnimationTool/VAT_TL_Meshes/ (Consultado en: 18-08-2020)
- Dean Ashford, “UE4 - Tutorial - Vertex Animation (3DS Max to Unreal Material)”, Feb. 2020. [En línea]. Recuperado de: https://youtu.be/6Cain-u9_g4 (Consultado en: 18-08-2020)
- Unreal Engine, “Into to Cascade: Particle Terminology | 01 | v4.2 Tutorial Series | Unreal Engine”, Jul. 2014. [En línea]. Recuperado de: https://youtu.be/OXK2Xbd7D9w (Consultado en: 20-08-2020)

- Unreal Engine, “Intro to Cascade: Starter Content Particle Systems | 10 | v4.2 Tutorial Series | Unreal Engine”, Jul. 2014. [En línea]. Recuperado de: https://youtu.be/mDHZcecbsLg (Consultado en: 20-08-2020)
- “Particle System User Guide”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/ParticleSystems/UserGuide/ (Consultado en: 20-08-2020)
- “Depth Expressions”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/ExpressionReference/Depth/ (Consultado en: 24-08-2020)
- Ben Ormstad, “UE4 Tutorial | Spawn Waves of Enemies in Increasing Numbers”, Mar. 2017. [En línea]. Recuperado de: https://youtu.be/6FmUGF1trlY (Consultado en: 31-08-2020)
- CodeLikeMe, “Safely Destroy Actors When Out Of Range and Not On Screen - UE4 Tutorial #385”, Mar. 2020. [En línea]. Recuperado de: https://youtu.be/Wdx83Ds1OhU (Consultado en: 04-09-2020)
- Tech Art Aid, “UE4 Optimization: Instancing”, Nov. 2016. [En línea]. Recuperado de: https://youtu.be/oMIbV2rQO4k (02-10-2020)
- Świerad Oskar, “Tech Art Aid”, *Tech Art Aid*. [En línea]. Recuperado de: https://techartaid.com/ (Consultado en: 02-10-2020)
- Simon, “Render Hell – Book III”, *simonschreibt*. [En línea]. Recuperado de: https://simonschreibt.de/gat/renderhell-book3/ (Consultado en: 02-10-2020)
- Nizio Cole, “Unreal engine - Change level with a blueprint”, Feb. 2017. [En línea]. Recuperado de: https://youtu.be/2ZakwxEWobg (Consultado en: 09-10-2020)
- Mathew Wadstein, “WTF Is? Apply and Receive Point Damage in Unreal Engine 4”, Feb. 2016. [En línea]. Recuperado de: https://youtu.be/B-2NvegyrcQ (Consultado en: 14-10-2020)
- Mathew Wadstein, “WTF Is? Apply and Receive Radial Damage in Unreal Engine 4”, Feb. 2016. [En línea]. Recuperado de: https://youtu.be/asZZ6awygtc (Consultado en: 14-10-2020)
- Titanic Games, “Unreal Engine 4 - Line Trace Applying Damage”, Oct. 2016. [En línea]. Recuperado de: https://youtu.be/d3XioV6WKqs (Consultado en: 16-10-2020)
- “Spawning and Destroying an Actor”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/SpawnAndDestroyActors/ (Consultado en: 16-10-2020)
- “Sequencer Overview” *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/AnimatingObjects/Sequencer/Overview/ (Consultado en: 18-11-2020)
- Ryan Laley, “Unreal Engine 4 Tutorial - Cutscenes - Level Sequencer”, Oct. 2018. [En línea]. Recuperado de: https://youtu.be/dmqC7s91RBU (Consultado en: 18-11-2020)

- “Foliage Tool”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Foliage/ (Consultado en: 25-11-2020)
- “1.3 - Radial Force / Impulse”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/Resources/ContentExamples/Physics/1_3/ (Consultado en: 26-11-2020)
- “Destructible”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Components/Destructible/ (Consultado en: 26-11-2020)
- Markom3D, “How to Enable Destructible Mesh ue4 4.21”, Ene. 2016. [En línea]. Recuperado de: https://youtu.be/dE_7u7GDtdE (Consultado en: 26-11-2020)
- Shaun Foster, “Unreal 4 Destructible Mesh Tutorial (with resource links below)”, Sep. 2016. [En línea]. Recuperado de: https://youtu.be/sKJlB_ep3lU (Consultado en: 26-11-2020)
- Matt Aspland, “Destructible Mesh (Any Mesh) - Unreal Engine 4 Tutorial”, May. 2020. [En línea]. Recuperado de: https://youtu.be/4MaIhIBZq_A (Consultado en: 26-11-2020)
- Titanic Games, “Unreal Engine 4 - Open Door/Level with Keys (Specific Keys)”, Sep. 2016. [En línea]. Recuperado de: https://youtu.be/LggvN6GZ3G4 (Consultado en: 02-12-2020)
- Reids Channel, “Unreal Engine 4 - Door and Key Tutorial”, Feb. 2020. [En línea]. Recuperado de: https://youtu.be/Nz3DRG9AgMg (Consultado en: 02-12-2020)
- Steamworks Development, “Steamworks Tutorial #1 - Building Your Content in Steampipe”, Oct. 2016. [En línea]. Recuperado de: https://youtu.be/SoNH-v6aU9Q (Consultado en: 04-12-2020)
- “Documentación de Steamworks”, *STEAMWORKS*, [En línea]. Recuperado de: https://partner.steamgames.com/doc/home (Consultado en: 04-12-2020)
- Matt Aspland, “How To Create Unique Ragdoll Physics (Ragdoll Death) - Unreal Engine 4 Tutorial”, Jul. 2020. [En línea]. Recuperado de: https://youtu.be/8d_x8M9DDV8 (Consultado en: 11-12-2020)
- Horacio Meza, “Tutorial Unreal Engine 4 Español | Activar Ragdoll y Bomba con Explosión”, Jun. 2017. [En línea]. Recuperado de: https://youtu.be/aD-Lzp8KwbI (Consultado en: 11-12-2020)
- underscore, “UE4 Tutorial: Ragdoll Physics”, Ene. 2019. [En línea]. Recuperado de: https://youtu.be/1OcGAGT2opU (Consultado en: 11-12-2020)
- Nawaf_Al-Rawachy, “How to change the icon of your game?”, *Unreal Engine Forums* Abr. 2016. [En línea]. Recuperado de: https://forums.unrealengine.com/t/how-to-change-the-icon-of-your-game/352581 (Consultado en: 27-01-2021)
- Matt Aspland, “Splash Screen/Intro Credits - Unreal Engine 4 Tutorial”, May. 2020. [En línea]. Recuperado de: https://youtu.be/WQKLZYtYxrA (Consultado en: 04-02-2021)
- Studio Don Quixote, “how to upload game to steam”, Mar. 2018. [En línea]. Recuperado de: https://youtu.be/Rfvl7awkzl0 (Consultado en: 06-02-2021)

- “Online Subsystem Steam”, *Unreal Engine Documentation,* [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Online/Steam/ (Consultado: 06-02-2021)
- “SteamCMD”, *Valve Developer Community,* [En línea]. Recuperado de: https://developer.valvesoftware.com/wiki/SteamCMD (Consultado: 06-02-2021)
- “Localization Overview”, *Unreal Engine Documentation*, [En línea]. Recuperado de: https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/Localization/Overview/ (Consultado: 09-04-2021)
- Mamoiem, “Unreal Engine, Localization and Culture packages - UE4U.XYZ”, Feb. 2016. [En línea]. Recuperado de: https://youtu.be/LNrjpWas8Bg (Consultado: 09-04-2021)
- Slayemin, “Unreal Engine 4 Localization Guide”, Ago. 2018. [En línea]. Recuperado de: https://youtu.be/0Pi4_ceURHY (Consultado: 09-04-2021)
- Unreal Engine, “Localizing Action RPG Game | Inside Unreal”, Mar. 2020. [En línea]. Recuperado de: https://youtu.be/UD2_TEgxkqs (Consultado: 16-04-2021)
- Mathew Wadstein, “Unreal Engine 4.18 - In Editor Localization Preview Highlight”, Nov. 2017. [En línea]. Recuperado de: https://youtu.be/Ou048Ao19WA (Consultado: 16-04-2021)
- Der Sky, “Spiel übersetzen / Localization (Blueprint Version) ► Unreal Engine Tutorial (German)”, Ene. 2020. [En línea]. Recuperado de: https://youtu.be/4TBdblFnf-M (Consultado en: 23-04-2021)
- Mather Wadstein, “Unreal Engine 4.18 - In Editor Localization Preview Quickie”, Nov. 2017. [En línea]. Recuperado de: https://youtu.be/ZOSOFd67WC8 (Consultado en: 23-04-2021)
- BRAINSHACK, “4.15: Translating your Game with the Localization Dashboard”, May. 2017. [En línea]. Recuperado de: https://youtu.be/GiviTjyG0zc (Consultado en 23-04-2021)
- “Hannah The Game”, *Spaceboy*, [En línea]. Recuperado de: https://hannahthegame.com/
- Spaceboy, “Hannah”, Steam, Mar. 2021. [En línea]. Recuperado de: https://store.steampowered.com/app/1466870/Hannah/
- “Spaceboy”, *Spaceboy*, [En línea]. Recuperado de: https://spaceboy.mx/