INTERACCIÓN CON PERSONAJES BASADOS EN VÓXELES

JAMES MIGUEL ROMERO QUINTERO 1201786
DANIEL SANTIAGO SANDOVAL SALINAS 1201770

DIRIGIDO POR
CARLOS ALBERTO ROCHA CASTAÑO

UNIVERSIDAD MILITAR NUEVA GRANADA
FACULTAD DE INGENIERÍA
INGENIERÍA EN MULTIMEDIA
BOGOTÁ D.C
2021

TABLA DE CONTENIDOS

RESUMEN

1. DEFINICIÓN DEL PROBLEMA Y OBJETIVOS 09
- 1.1. Antecedentes 09
- 1.2. Planteamiento del problema 10
  - 1.2.1. Identificación 10
  - 1.2.2. Descripción 11
- 1.3. Objetivos 12
  - 1.3.1. Objetivo General 12
  - 1.3.2. Objetivos Específicos 12
- 1.4. Justificación 12
- 1.5. Estado del arte 13
- 1.6. Marco Teórico 14
- 1.7. Metodología 17

2. DESARROLLO DEL PROTOTIPO 18

- 2.1. Análisis e Implementación de vóxeles 18
- 2.2. Documento de diseño 26
  - 2.2.1. Concepto 26
  - 2.2.2. Objetivos del jugador 26
  - 2.2.3. Jugabilidad 26
  - 2.2.4. Mecánicas del juego 27
    - 2.2.4.1 Acciones del usuario 27
    - 2.2.4.2 Acciones del personaje principal 28
    - 2.2.4.3 Acciones del personaje enemigo 28
  - 2.2.5. Casos de uso 29
  - 2.2.6. Tipo de escenario 33
  - 2.2.7. Estilo gráfico 33
    - 2.2.7.1 Referencia de arte 33
    - 2.2.7.2 Arte conceptual 35
    - 2.2.7.3 Modelos 3D 36
  - 2.2.8. Interfaz gráfica 37
    - 2.2.8.1 HUD 37
    - 2.2.8.2 Mockup 38
- 2.3. Implementación y prototipado 39

3. PRUEBAS **44**

3.1. Pruebas de rendimiento **44**
3.2. Pruebas de usuario **46**
3.3. Resultado y análisis **50**

4. CONCLUSIÓN Y TRABAJO FUTURO **52**
5. BIBLIOGRAFÍA **54**
6. ANEXOS **60**

Lista de tablas

Tabla 01: Ficha técnica equipo **18**
Tabla 02: Cuadro comparativo de los 4 métodos **25**
Tabla 03: Comparativa entre versiones (vóxel vs malla) **45**

Lista de figuras

Figura 01: Gráfico metodología **17**
Figura 02: Esquema implementación aplicación 1 **20**
Figura 03: Esquema implementación aplicación 2 **21**
Figura 04: Esquema implementación aplicación 3 **23**
Figura 05: Esquema implementación aplicación 4 **24**
Figura 06: Mapa de navegación **29**
Figura 07: Caso de uso de interacción **30**
Figura 08: Caso de uso de durante el juego (versión malla) **31**
Figura 09: Caso de uso de durante el juego (versión vóxel) **32**
Figura 10: Collage referencias de arte **33**
Figura 11: Referencias de personaje principal **34**
Figura 12: Referencias de personaje enemigo **34**
Figura 13: Model sheet personaje enemigo **35**
Figura 14: Bocetos personaje principal **35**
Figura 15: Modelo 3D personaje enemigo **36**
Figura 16: Modelo 3D personaje principal **36**
Figura 17: HUD vidas del personaje principal **37**
Figura 18: HUD Barra de vida enemiga, carga láser y vidas del personaje principal **37**
Figura 19: Vista menú principal **38**
Figura 20: Vistas pausa y victoria **38**
Figura 21: Vistas pérdida de juego y pantalla de carga **38**
Figura 22: Rig personaje principal **39**
Figura 23: Versión malla y vóxel del personaje enemigo **40**
Figura 24: Modelo 3D del personaje enemigo **40**
Figura 25: Cuerpo externo versión malla y versión vóxel (SDF) **41**
Figura 26: Colisionadores personaje enemigo **42**
Figura 27: Deformación con colisionador personaje enemigo **43**
Figura 28: Deformación con colisionador personaje enemigo 2 **43**
Figura 29: Resultado primera pregunta encuesta (gráfico torta) **47**

Figura 30: Resultado segunda pregunta encuesta (gráfico barras) **47**
Figura 31: Resultado tercera pregunta encuesta (gráfico barras) **48**
Figura 32: Resultado cuarta pregunta encuesta (gráfico barras) **48**
Figura 33: Resultado quinta pregunta encuesta (gráfico torta) **49**
Figura 34: Resultado sexta pregunta encuesta (gráfico torta) **49**

Glosario

1. Compute Buffer: Buffer de datos usado para operaciones varias en la GPU, particularmente en Compute Shaders.
2. Compute Shader: Es una etapa programable en el pipeline de renderizado de un procesador gráfico que se usa completamente para calcular información arbitraria. Si bien puede renderizar, generalmente se usa para tareas que no están directamente relacionadas con el dibujo de triángulos y píxeles.
3. DrawProceduralIndirect: Es una función de Unity utilizada para dibujar geometría procedural en la GPU. Realiza una llamada de dibujo en la GPU, sin ningún vértice o buffer de índice. La cantidad de geometría a dibujar se lee de un Compute Buffer. El caso de uso típico es generar una cantidad arbitraria de datos desde un ComputreShader y luego renderizarlos, sin requerir una lectura de la CPU.
4. Native Container: Es un tipo de valor administrado que proporciona un wrapper de C# relativamente seguro para la memoria nativa. Contiene un puntero a una asignación no administrada. Cuando este se combina con el C# Job System de Unity, permite que un job acceda a datos compartidos con el thread (hilo) principal en lugar de trabajar con una copia.
5. Shader: Es un programa definido por el usuario diseñado para ejecutarse en alguna etapa de un procesador gráfico. Proporcionan el código para ciertas etapas programables del pipeline de renderizado. También se pueden usar en una forma un poco más limitada para el cálculo general en la GPU.
6. Signed Distance Functions (SDF, funciones de distancia con signo): Son funciones matemáticas que describen una forma primitiva; toma una posición como dato inicial y genera la distancia desde esa posición a la parte más cercana de una forma.
7. Surface Shader: En Unity, es un enfoque de generación de código que hace que sea mucho más fácil escribir Shaders que usar Vertex / Pixel Shaders de bajo nivel. Estos simplemente generan todo el código repetitivo que tendría que escribirse a mano, pero de igual manera se programan en HLSL.

RESUMEN

El uso de vóxeles en aplicaciones interactivas se aplica desde hace décadas, con múltiples maneras de implementación de acuerdo al hardware para el que se desarrollaba y los objetivos del desarrollador. En los videojuegos concretamente, las representaciones basadas en vóxeles generalmente están supeditadas a efectos de iluminación, postprocesado, simulaciones, estilo visual y generación y deformación de terreno procedural, siendo muy raro encontrar personajes completos construidos con vóxeles. En este trabajo, se explora una implementación de un personaje enemigo deformable basado en vóxeles desarrollado a partir del algoritmo de Marching Cubes y Signed Distance Fields. Con el fin de analizar las diferencias de rendimiento de máquina y de percepción de usuario, se elabora un prototipo de videojuego que incorpora al personaje enemigo en dos versiones: una que utiliza la implementación basada en vóxeles y otra un modelo en malla poligonal convencional.

PALABRAS CLAVE: vóxel, videojuego, deformación en tiempo real, interacción, Signed Distance Field.

1. DEFINICIÓN DEL PROBLEMA Y OBJETIVOS

1.1 ANTECEDENTES

Las representaciones basadas en vóxeles en la computación gráfica tienen una larga historia, desde comunes cuadrículas de vóxeles (Voxel Grids) [1] hasta estructuras Octree (árbol octal) [2] usadas para acelerar los procesos de computación gráfica por décadas.

Si bien los gráficos tridimensionales basados en mallas poligonales son el estándar actualmente e incluso son reforzados por aceleración de hardware en las GPUs (unidades de procesamiento de gráficos), los vóxeles aún son usados para mejores resultados en tareas como el ray tracing [3][4][5], análisis topológico [6], estimación de volumen [7], detección de colisión [8], renderizado de sombras [9], y otros efectos complejos de iluminación (tales como iluminación volumétrica) [10].

Además de la iluminación, los vóxeles son usados también en el área de las aplicaciones interactivas en tiempo real, tales como visualizaciones y simulaciones médicas [11], simulaciones de procedimientos industriales [12], aplicaciones de realidad extendida [13] y videojuegos, en estos últimos se han empleado para la generación de terreno procedural [14][15] y modificación dinámica del mismo [15][16], simulación física más realista y en tiempo real [17], estética [18] y sistemas de destrucción de elementos en tiempo real [15][16][17].

El uso de vóxeles en aplicaciones interactivas, y especialmente en videojuegos ha sido empleado mayormente para dar dinamismo o formas de interacción a elementos que por lo general son de carácter estático, como el terreno de un escenario y/o algunos de sus elementos, sin embargo, a excepción de motores gráficos que funcionan con representaciones basadas en vóxeles [19][20][21], las posibilidades que ofrece el dinamismo de estos no es tan explorada en lo que a personajes refiere. Ha habido investigaciones sobre cómo animar eficientemente SVOs (árbol octal de vóxeles dispersos) [22][23] puesto que son una de las técnicas de representación de gráficos basados en vóxeles más usada, sin embargo, estos métodos se quedan cortos puesto que los SVOs son estáticos (aunque no inmutables), representar objetos de geometría estática (como rocas, montañas, terreno en general) es relativamente sencillo pero objetos animados o dinámicos como personajes se vuelve un poco más complejo [22], por ello también existen motores gráficos que se benefician del uso de vóxeles en aspectos del entorno, la física y la iluminación de sus escenas pero objetos dinámicos como personajes y superficies planas (como piezas arquitectónicas y amoblado) se implementan a través de mallas poligonales convencionales [24][25], llegando a soluciones mixtas que aprovechan las bondades de ambos métodos.

En 2015, Alex Evans, co-fundador y director técnico de la empresa Media Molecule hizo una charla para Siggraph en donde explicaba el funcionamiento a grandes rasgos de su juego Dreams para PlayStation 4, el cual es una aplicación de creación de contenido audiovisual y videojuegos 3D que destaca por un motor de renderizado que no utiliza triángulos, sino que se basa en el uso de múltiples operaciones de CSG (Geometría constructiva de sólidos) que son evaluadas en tiempo real a SDFs (Funciones de Densidad con Signo) y a partir de las cuales generan nubes de puntos de resolución adaptativa [21]. Este método permite que un usuario cree modelos tridimensionales mucho más rápido y con mucha más libertad artística que los métodos convencionales con polígonos, sin embargo, parte del proceso establecido en Dreams para la creación de objetos y personajes está limitada a que una vez se esté satisfecho con el trabajo realizado, el objeto o personaje se convierte en un modelo estático, sólo puede ser deformado al añadirle un rig y skinning, pero en general, para hacer un cambio al modelo habría que ingresar al menú de edición, realizar las modificaciones pertinentes y luego guardarlo otra vez como objeto estático, limitando el hacer modificaciones en tiempo real como sistemas de destrucción o deformación procedural de objetos y personajes. [26]

Hay casos en los que los personajes son creados a partir de mallas poligonales convencionales pero utilizan vóxeles para ciertos procesos como la creación de sistemas de skinning automático [27] o interacciones físicas como colisiones [28][29]. Se han visto también sistemas de destrucción de personajes en tiempo real que se enfrascan en el uso de renderizado de vóxeles como cubos poligonales que se separan del personaje según las acciones del jugador con el entorno u otros personajes no jugables, obteniendo resultados poco realistas pero que dejan ver interacciones interesantes con los modelos [30][31].

1.2 PLANTEAMIENTO DEL PROBLEMA

1.2.1 IDENTIFICACIÓN

El uso de vóxeles en aplicaciones interactivas como los videojuegos se da generalmente para efectos de iluminación [18], generación y deformación de terreno y elementos estáticos que hagan parte del escenario [15][16], simulaciones físicas más realistas y eficientes [17], sin embargo, a la hora de representar geometrías dinámicas como lo son los personajes su uso es menos común, principalmente por la complejidad técnica que esto conlleva [22][23], y es por esto que los desarrolladores generalmente optan por implementaciones híbridas que utilizan representaciones basadas en vóxeles para los entornos y mallas poligonales convencionales para los personajes [24][25], o bien implementan también personajes basados en vóxeles pero sacrificando

ciertas interacciones como la deformación en tiempo real de los mismos [20][21] y la resolución de sus representaciones [31].

1.2.2 DESCRIPCIÓN

Los personajes en aplicaciones interactivas, sean controlados o no por un usuario, son elementos dinámicos que cuentan con animaciones preestablecidas y movimientos varios para dar la sensación de vitalidad y personalidad, independientemente de su morfología (antropomorfos, zoomorfos, amorfos...), es por esto que su representación generalmente está basada en el uso de mallas poligonales convencionales que nos permiten obtener unos rendimientos gráficos óptimos. [22]

El uso de vóxeles es más adecuado que las mallas poligonales para describir algunos elementos estáticos, efectos de iluminación o simulaciones físicas basándose en estructuras de datos que se utilizan en pro de optimizar su procesamiento, sin embargo, estas estructuras consumen mucho más tiempo/memoria a la hora de manipular elementos dinámicos según el acercamiento que tome el desarrollador, generando posibles problemas de rendimiento en la implementación de esta técnica en personajes en tiempo real. [22]

Existen motores gráficos que hacen uso de métodos de renderizado de vóxeles para todos los elementos de una escena, tanto estáticos como dinámicos, incluyendo personajes, pero, independientemente del método de renderizado que usen, al final los elementos dinámicos carecen de la posibilidad de modificación o deformación en tiempo real, desaprovechando uno de los beneficios que ofrece el dinamismo del uso de vóxeles con el fin de evitar posibles caídas en el rendimiento por la actualización constante de los modelos deformados sumada a la actualización que conlleva la animación de estos mismos. [20][21]

Basado en lo anterior, es importante identificar las diferencias en cuanto a rendimiento del sistema y la percepción que pueda tener un usuario frente a la implementación de vóxeles en la interacción con personajes a comparación de las mallas poligonales convencionales, todo esto con el fin de analizar si las características de los vóxeles se pueden aplicar a un personaje con un rendimiento (computacional/gráfico) estable y una buena percepción de usuario.

1.3 OBJETIVOS

1.3.1 OBJETIVO GENERAL

Desarrollar un prototipo de un videojuego con un personaje deformable basado en vóxeles para analizar su rendimiento e impacto en la percepción del usuario comparado con el uso de mallas poligonales convencionales.

1.3.2 OBJETIVOS ESPECÍFICOS

- Identificar el estado del arte de la implementación de personajes basados en vóxeles en los videojuegos.

- Desarrollar un prototipo funcional de un videojuego que integre un mismo personaje basado en vóxeles (deformable) y en mallas poligonales (no deformable).

- Realizar pruebas de rendimiento del sistema y de la percepción de los usuarios, que permitan evaluar la influencia de la implementación de un personaje deformable basado en vóxeles en comparación del uso de mallas poligonales convencionales.

1.4 JUSTIFICACIÓN

En el desarrollo de personajes para videojuegos, se hace uso de mallas poligonales convencionales, limitando las exploración de mecánicas o acciones menos convencionales como la deformación o destrucción de un personaje en tiempo real puesto que estos métodos son restrictivos a la hora de hacer modificaciones en la geometría ya creada, a comparación de los vóxeles que permiten la implementación de técnicas de simulación que con otros métodos de modelado sería muy complicado lograr [32], agregando más posibilidades a los desarrolladores al momento de crear interacciones con un personaje. Actualmente el uso de vóxeles está en áreas muy específicas del desarrollo debido a que no existen muchos hardware que optimicen el renderizado de estos en alta resolución y de forma eficiente debido a que la mayoría están enfocados en mallas poligonales [22][32].

Es por esto que se propone desarrollar un prototipo de una aplicación 3D interactiva en la cual se evaluará el rendimiento de un personaje elaborado a partir de métodos basados en vóxeles en comparación con un personaje realizado en mallas poligonales convencionales, con el objetivo de analizar otras posibilidades de visualización que se le ofrece a un usuario al momento de interactuar con un personaje en un videojuego.

1.5 ESTADO DEL ARTE

Los personajes basados en vóxeles son un tema complejo puesto que dependen de varios factores, entre esos su implementación y los comportamientos que se quieran lograr. Con respecto a esto, trabajos como “Animated Sparse Voxel Octrees” de Bautembach [22] y “Efficient Animation of Sparse Voxel Octrees for Real-Time Ray Tracing” de Engmark et al. [23] se enfocan en explorar formas de animación en tiempo real de árboles octales de vóxeles puesto que son una de las técnicas de representación más usada, logrando muy buenos resultados pero careciendo de la posibilidad de deformación en tiempo real por la naturaleza de la estructura de datos.

Otros trabajos han utilizado vóxeles para implementar características sobre personajes modelados con mallas poligonales convencionales, tales como “Volumetric heat diffusion skinning” del estudio Wolfire [27] que es un sistema de skinning automático a partir de la voxelización del modelo de un personaje, y “FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation” de Rivers & James [28] que es un sistema de deformación en tiempo real de modelos en mallas poligonales basado en la generación de latices mediante voxelización.

Existen juegos comerciales que crean a sus personajes basados en vóxeles y los renderizan por medio de cubos poligonales, haciendo una combinación de series de cubos pequeños con el objetivo de poder cambiar dinámicamente los cubos que los componen, variando o quitando algunos de su posición para exagerar cortaduras y destruir partes del modelo, un ejemplo de este tipo de implementación es el juego “Paint the Town Red Dev Log 2” de South East Games [30] donde su objetivo principal es el desmembramiento de sus personajes, y “Voxatron” de White [31] es una recopilación de minijuegos basados con estética voxel art, los cuales tanto como los personajes y escenarios son deformables.

Recientemente se han visto desarrollos que exploran distintas formas de renderizado y generación de vóxeles en videojuegos, a tal punto que los modelos resultantes son muy similares (y en unos casos muy superiores) visualmente a los modelos de personajes en mallas poligonales tradicionales, como es el caso de “Dreams” del estudio Media Molecule [21] desarrollado como una herramienta de creación de contenido audiovisual (3D) e interactivo para PlayStation 4 y 5; Dreams utiliza SDFs para generar los vóxeles y los renderiza como una isosuperficie a partir de splatting, realizando todo el proceso directamente en la GPU; al ser una herramienta de creación de contenido, todos los modelos elaborados en Dreams son estáticos, es decir, una vez modelados deben “bloquearse” para dejar de recibir más cambios y poder añadirles un rig si el objetivo es animarlo para un videojuego o corto animado, dejando de lado la posibilidad de deformación en tiempo real de un personaje.

“Clayxels” de Interguglielmi [33] y “MudBun” de Chou [34] se basan en las técnicas de Dreams para generación de vóxeles adaptando el método de renderizado para hacerlo funcionar con el algoritmo de Marching Cubes en Unity, enfocando su desarrollo al esculpido digital y a la creación de esculturas y efectos visuales volumétricos respectivamente; debido a que tanto Clayxels como MudBun se encuentran en la tienda de assets de Unity, cualquier usuario de dicho motor puede comprar alguno de los dos paquetes y empezar a experimentar con sus herramientas, dando como resultado prototipos con interacciones interesantes que incluyen deformación de personajes en tiempo real [35], simulaciones físicas y efectos visuales estilizados [34].

Otro desarrollo que utiliza el algoritmo de Marching Cubes para renderizar vóxeles y se enfoca en la creación de contenido volumétrico es el “Voxel Plugin” para Unreal Engine 4 de Careil [36]; tal como sucede con Clayxels y MudBun, puesto que Voxel Plugin es un paquete para el motor Unreal Engine 4, cualquier usuario de dicho software puede comprarlo o usar la versión de prueba y experimentar con sus funciones, entre las cuales destacan la generación procedural de mundos infinitos, simulación física, deformación del entorno [36] y de personajes [37].

1.6 MARCO TEÓRICO

Los datos volumétricos son entidades 3D que pueden tener información dentro de ellas, pueden no estar formadas por superficies y bordes, o pueden ser demasiado voluminosos para ser representados geométricamente. La visualización de volumen es un método para extraer información significativa de datos volumétricos mediante gráficos e imágenes interactivos, y está relacionada con la representación, modelado, manipulación y renderizado de datos de volumen. [38]

Por lo general, un conjunto de datos volumétrico se representa como una cuadrícula 3D de elementos de volumen (vóxeles) y comúnmente se almacena en un búfer de volumen, que es un gran arreglo 3D de vóxeles. Alternativamente, se han empleado otras estructuras y formatos de datos para almacenar y manipular el conjunto de datos, como la descomposición de celdas en octrees [2], matrices de vóxel dispersos, cuadrículas irregulares y superficies de objetos. [39]

Un vóxel es la unidad cúbica de volumen centrada en un punto de una cuadrícula 3D. Representando una unidad de volumen, el vóxel es la contraparte 3D del pixel 2D que representa una unidad de área. Cada vóxel tiene valores numéricos asociados que representan algunas propiedades medibles o variables independientes (como el color, opacidad, densidad, material, índice de refracción, velocidad, fuerza, distancia a la superficie, etc.) del fenómeno real u objeto que reside en la unidad de volumen representada por ese vóxel. [39]

Los vóxeles tienen múltiples formas de representación, entre la más comunes destacan los Sparse Voxel Octrees (SVO, árbol octal de vóxeles dispersos), los cuales pueden encarnar la geometría de superficie y la geometría de volumen en un octree [2]. En este formato, el nodo raíz ocupa todo el espacio, cada nodo (vóxel) subdivide el espacio que representa en ocho subespacios y representa una aproximación de la geometría que se encuentra dentro del nodo. En cada paso de subdivisión, el error de aproximación disminuye y, en consecuencia, se obtiene una mejor representación de la geometría real. Generalmente, el renderizado de los SVO se da por medio de ray tracing, una técnica que genera la imagen resultante simulando trayectorias de luz, las cuales pueden comenzar desde la cámara o desde la fuente de luz en concreto. En el caso de los SVO, para cada pixel de la pantalla se traza un rayo desde la cámara hasta el octree en un proceso iterativo hasta que el rayo alcanza un nodo (vóxel) según la profundidad deseada, y, una vez alcanzado el nodo se toma sus propiedades (color, iluminación, sombreado...) y se dibuja un color en el pixel correspondiente para ese rayo, así hasta que se tiene una imagen completa de la escena en pantalla [40].

Otra forma bastante común es la generación de isosuperficies, las cuales consisten en superficies que representan puntos de un valor constante (por ejemplo presión, temperatura, velocidad, densidad...) dentro de un volumen de espacio (vóxel), y el método más conocido para generar isosuperficies a partir de arreglos de vóxeles es el algoritmo de Marching Cubes (cubos de marcha), el cual crea la superficie cruzando los bordes del arreglo con el contorno del volumen. Donde la superficie se cruza con el borde, el algoritmo crea un vértice y utiliza una tabla (LookUp Table) de diversas triangulaciones dependiendo de los diferentes patrones de intersecciones de los bordes obteniendo como resultado una superficie a partir de dichos triángulos [41].

Las mallas poligonales son una forma de representación geométrica para aplicaciones de gráficos 3D interactivos. Son lo suficientemente flexibles como para aproximar formas arbitrarias y pueden ser procesadas de manera eficiente por el hardware de gráficos actual que está disponible incluso en las máquinas de bajo costo de hoy, siendo más eficientes para modelar superficies planas (como elementos arquitectónicos y amoblado) [22]. Las mallas poligonales constan de un conjunto de vértices, bordes, caras (compuestas por triángulos, cuadriláteros y polígonos convexos simples generalmente) y relaciones topológicas entre ellos, a partir de las cuales una estructura de datos define cómo se almacena cada elemento y qué referencias necesita a su vecindario. Entre las estructuras de datos más comunes se encuentran las basadas en caras (face-based), que almacenan para cada cara punteros a sus vértices y sus caras vecinas, y las estructuras de datos basadas en bordes (edge-based), que almacenan para cada borde punteros a sus vértices y a los bordes vecinos. [42]

Los videojuegos son aplicaciones interactivas voluntarias en la que uno o más jugadores siguen reglas que restringen su comportamiento, promulgando un conflicto artificial que puede terminar o no en un resultado cuantificable y que pueden basarse o no en una historia. Los usuarios interactúan con videojuegos gracias a dispositivos de entrada y aparatos audiovisuales. [43]

Los personajes en los videojuegos son esenciales. Una persona se conecta necesariamente a un juego a través de su personaje principal, identificándose con este como en ningún otro medio es posible, pues es dueño de sus propias acciones y no toma una posición de observador pasivo. El vínculo y la lealtad a estos personajes puede ser prácticamente instantánea.[44]

Además de los personajes, otro punto esencial para la experiencia de un usuario en un videojuego es el feedback (retroalimentación). El feedback se refiere al bucle clásico entre presionar un botón y que el juego responda de una manera que simplemente se sienta bien. Cualquier acción que realice un usuario en un videojuego debe tener una reacción que comunique de forma evidente que dicha acción se realizó exitosamente. Un juego con un feedback claro puede hacer que cualquier personaje o acción se sienta más creíble o interesante para el usuario, sin importar qué tan sencilla sea.[45]

1.7 METODOLOGÍA

Para el desarrollo del proyecto se manejó la metodología cascada, dividiendo el desarrollo en 5 etapas cuyo inicio dependía de la finalización de la etapa que le precede pero así mismo era permitido regresar a cualquier etapa y

corregirla según se necesitara por medio de iteraciones en las revisiones parciales del prototipo, esto basado en [46]. Las etapas se definieron como Análisis, Diseño, Desarrollo visual, Implementación y Prototipado, y finalmente Pruebas.

Análisis: En esta etapa se realizó una investigación de distintos métodos de renderizado de vóxeles y se escogió uno que podía cumplir con los objetivos del proyecto, así que a partir de ese método se llevaron a cabo múltiples pruebas de distintas implementaciones de ese mismo método hasta dar con una que fuese idónea para el desarrollo del prototipo.

Diseño: En esta etapa se realizó la estructuración del proyecto, específicamente el funcionamiento central del juego (mecánicas, objetivo, mapa de navegación, casos de uso), la línea gráfica, el paradigma de programación y pequeños prototipos funcionales de mecánicas para descartar ideas que surgían en el proceso.

Desarrollo visual: En esta etapa se creó el contenido visual del juego (escenario, UI, personajes).

Implementación y prototipado: En esta etapa se desarrolló la lógica del juego y ésta se unificó con la implementación del método seleccionado de representación de vóxeles y con los assets gráficos (modelos 3D, UI, VFX) y sonoros, además de micro-optimizaciones y limpieza del código fuente.

Pruebas: En esta etapa final se realizaron pruebas de rendimiento del prototipo finalizado y la evaluación con usuarios y un experto en el sector de videojuegos respecto a la percepción del mismo en relación al método de vóxel y de malla poligonal.

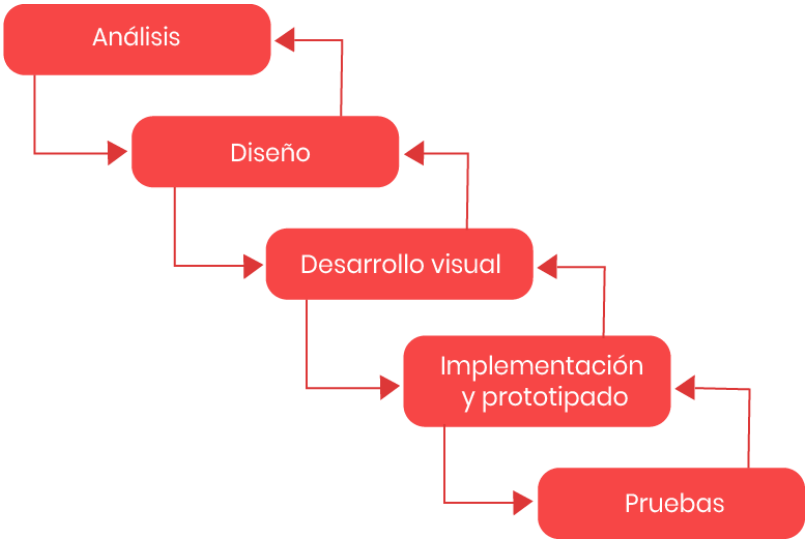

Figura 01: Gráfico metodología

2. DESARROLLO DEL PROTOTIPO

2.1 Análisis e implementación de vóxeles

El método de Marching Cubes fue escogido para la representación gráfica de los vóxeles puesto que es un algoritmo sencillo de implementar y como se basa en la extracción de mallas poligonales cumplía con el objetivo de obtener un resultado suavizado en lugar de cúbico, además de ser un método relativamente simple y rápido en ejecución debido a estar basado en su mayoría en el uso de lookup tables.

Para la implementación se utilizaron los programas Autodesk Maya [47] para la creación de los modelos en malla poligonal de los personajes, Blender [48] para el rigging y weight painting del personaje principal y el motor de videojuegos Unity [49] para la elaboración de toda la parte técnica puesto que la cantidad de herramientas y facilidades que otorga a los desarrolladores permitió llevar un flujo de trabajo mucho más eficiente y enfocarnos en la investigación y puesta en marcha de los métodos de generación, representación gráfica, deformación en tiempo real y estructuras de datos relacionados con los vóxeles.

Teniendo en cuenta que la finalidad del proyecto era el desarrollo de un prototipo de videojuego, la implementación del método de Marching Cubes debía ejecutarse en tiempo real (30 cuadros por segundo mínimo) y por ende se investigaron diversas formas que pudieran brindar los resultados más afines a lo requerido. La investigación se puso en práctica realizando pruebas de rendimiento progresivo, derivando en cuatro aplicaciones distintas teniendo los siguientes resultados en un computador con estas características:

Equipo	**Procesador**	**Ram**	**GPU**
Laptop Dell Inspiron 7577	Intel Core i7-7700HQ	16GB	NVIDIA GeForce GTX 1060 Max-Q 6 GB VRAM

Tabla 01: Ficha técnica equipo

Aplicación 1 (SDF + Compute Shaders + generación de malla por CPU):

Para esta primera implementación se utilizaron dos Compute Shader y dos Compute Buffer. Uno de los Compute Buffer contenía los datos de la posición (X, Y, Z) de cada vóxel, y un dato extra (W) para el valor de densidad. Este valor de densidad se calculaba en uno de los Compute Shader a partir de una forma primitiva generada por un SDF basado en [50]. El segundo Compute Shader recibía el Compute Buffer de vóxeles y sobre él se ejecutaba el algoritmo de Marching Cubes basado en [51], el cual, una vez calculada la isosuperficie de acuerdo a los datos de los vóxeles, se almacenaban los triángulos de dicha superficie en el segundo Compute Buffer que posteriormente era leído por la CPU, obteniendo así los triángulos y vértices de una isosuperficie que era finalmente generada proceduralmente en la CPU utilizando la clase MeshRenderer de Unity.

Este procedimiento tenía la ventaja de que los cálculos de mayor costo (como lo son los SDFs y el algoritmo de Marching Cubes) se delegaba a la GPU, la cual está diseñada para ejecutar ese tipo de tareas muy rápidamente [52], además que al generar la malla por medio de la CPU, se pudo aprovechar las funciones propias de Unity para el correcto cálculo de normales de la superficie, efectos de iluminación y cálculo de colisiones. Sin embargo, a la hora de deformar en tiempo real solo 1 figura primitiva se evidenció que el paso de información entre la GPU y la CPU era un proceso bastante costoso debido a que la CPU debe esperar un tiempo a que la GPU deje de trabajar y luego sí pueda mandar la información computada a la CPU, y dicho tiempo es muy variable puesto que depende de factores como el hardware en que se ejecute y la cantidad de cálculos realizados por frame [53], lo cual se tradujo en una baja de rendimiento considerable que además era indeterminable.

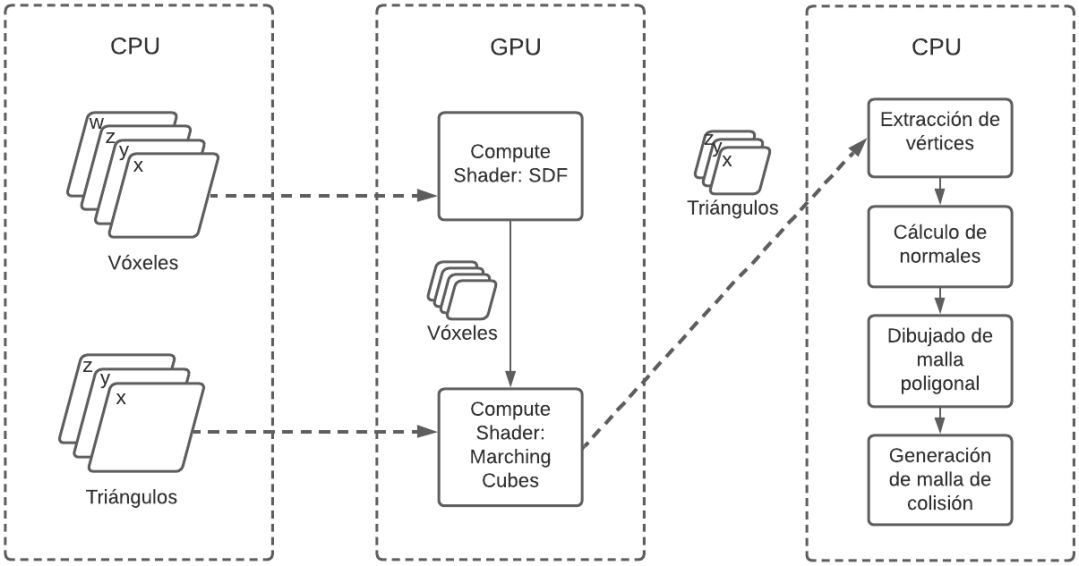

Figura 02: Esquema implementación aplicación 1

Aplicación 2 (SDF + C# Job System + Burst Compiler + Generación de malla por CPU):

Tomando como base la desventaja del paso de información de GPU a CPU en la anterior implementación, se intentó trabajar todo el proceso en solo una unidad, en este caso la CPU gracias a las ventajas observadas anteriormente. Para esta prueba se utilizó el C# Job System de Unity, que permite escribir de una manera simple código multithreaded (multihilo, para ser ejecutado en paralelo por los núcleos de la CPU), y el compilador Burst que optimiza mucho más el código escrito en C# y específicamente a través del Job System [54]. La implementación en este caso es muy similar a la anterior, la diferencia es que se cambiaron los Compute Buffer por Native Containers y los Compute Shaders por Jobs. Un Job generaba las figuras primitivas a partir de SDFs y guardaba la información en un Native Container con las posiciones (X, Y, Z) de cada vóxel y el valor de densidad W. Los vóxeles se enviaban a un 2do Job en donde se ejecutaba el algoritmo de Marching Cubes y los triángulos generados se guardaban en otro Native Container que era enviado a un 3er Job donde se extraían los vértices de los triángulos generados; estos vértices se almacenaban en un Native Container que finalmente era leído en el thread principal de Unity en donde se creaba una malla poligonal a partir de estos. La principal ventaja de este método es que, al generar la malla en la CPU se pudo hacer uso de la iluminación nativa de Unity, el cálculo de normales y

colisiones, pero con la diferencia de que no había ningún traspaso costoso de datos de GPU a CPU. Todo esto se tradujo a una mejora en el rendimiento, inclusive con múltiples primitivas y al realizar deformaciones en tiempo real.

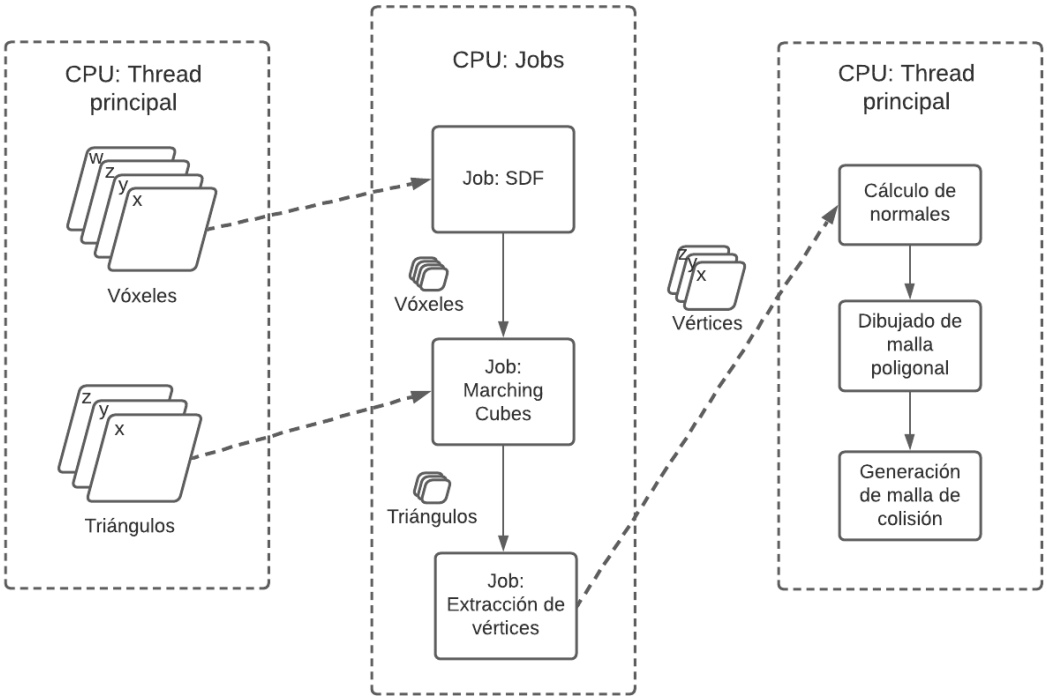

Figura 03: Esquema implementación aplicación 2

Aplicación 3 (Voxelizador + C# Job System + Burst Compiler + Generación de malla por CPU):

La única diferencia entre esta implementación y la anterior fue el prescindir de generar SDFs en tiempo real, sino que en lugar de eso se utilizó un ejemplo de una implementación de un algoritmo de voxelización básico que tomaba como entrada una malla poligonal que luego es trazada con rayos para encontrar las posiciones donde cada rayo se cruza con un triángulo, las cuales finalmente se utilizan para crear un arreglo de vóxeles [55]. Esto se hizo con el fin de alivianar la carga de la CPU, ya que el algoritmo de voxelización solo se ejecuta 1 vez y los datos luego son guardados en un arreglo con la posición (X, Y, Z) en donde un rayo hizo colisión con un triángulo, y un valor W de 0 o 1 determinando si en esa posición hay un vóxel vacío (no hubo colisión) o lleno (hubo colisión) respectivamente; luego, el arreglo resultante de vóxeles se envía a un Job que pasa los datos de un arreglo estándar de vectores a un Native Container del Job System. Posteriormente, los datos son procesados por un 2do Job que ejecuta el algoritmo de Marching Cubes y el resto de la

implementación es exactamente igual a la aplicación anterior. Si bien hubo una mejora en el rendimiento, esta no fue tan alta como se esperaba, además que introducía dificultades estéticas como que la malla resultante de la voxelización era muy cuadrada y aún aplicando interpolación a los valores de los vóxeles el resultado seguía siendo muy rígido, además presentaba dificultades de interacción como que la deformación en tiempo real (basada también en raycasting de la cámara a la malla generada e ir eliminando vóxeles en los puntos de colisión) era dependiente del ángulo de visión pues en casos en que el rayo hacía colisión formando un ángulo ortogonal se ignoraba dicha colisión y el vóxel no desaparecía. Finalmente esta aplicación hizo evidente que, si bien la generación de una malla procedural mediante la CPU traía beneficios como iluminación nativa de Unity, cálculo de normales y colisiones, la naturaleza de dicha malla procedural obligaba a que las colisiones fuesen calculadas por triángulo, lo cual es un proceso costoso si se quiere una colisión medianamente precisa [56], y de lo contrario podría crearse un modelo de colisión convexo que, de acuerdo a la forma de la malla, podría ser aceptablemente preciso, o muy poco preciso. Se estableció entonces que los colisionadores de la geometría procedural tendrían que ser básicos (en este caso de esfera) para que el rendimiento no se viera estancado por estos, lo cual llevó a reconsiderar el uso de la GPU, puesto que de cierto modo la ventaja más grande de la CPU podría terminar siendo un problema más adelante.

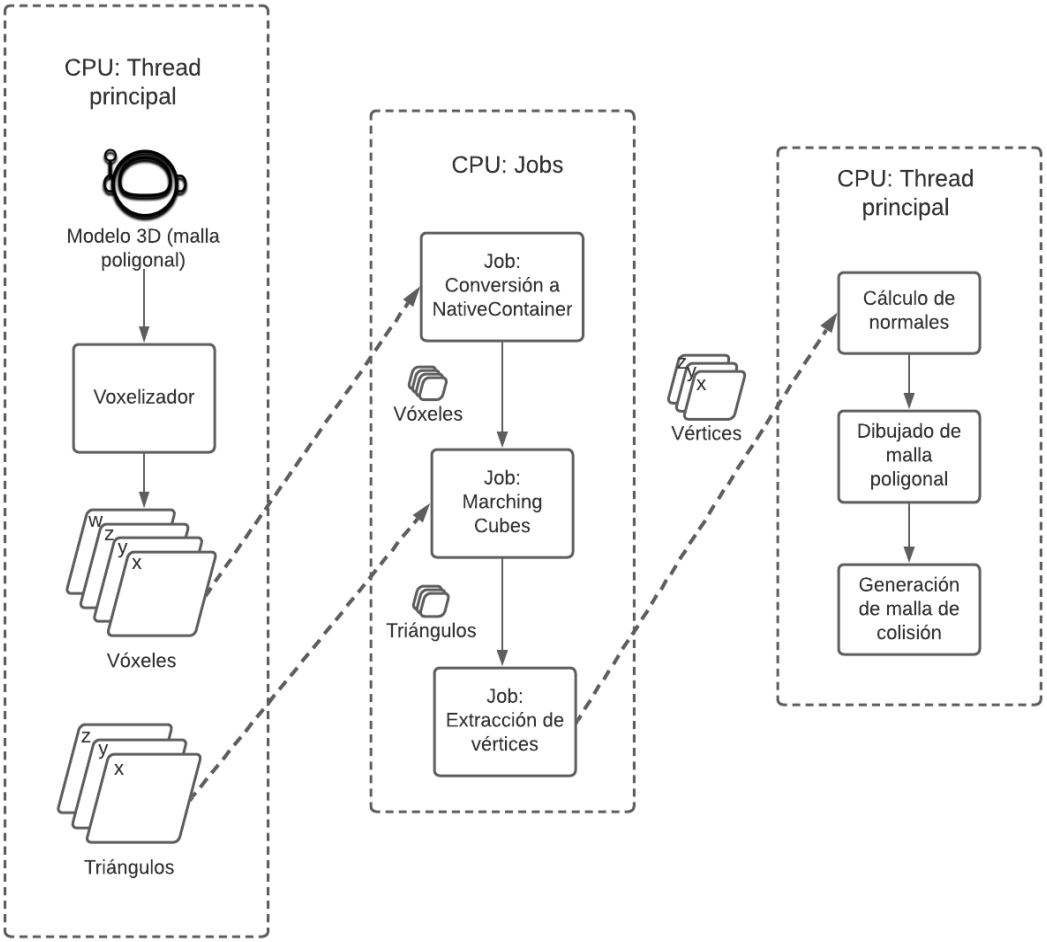

Figura 04: Esquema implementación aplicación 3

Aplicación 4 (SDF + Compute Shader + Generación de malla por GPU):

El problema más grande que se obtuvo con la primera implementación de Compute Shaders fue crear la malla mediante la CPU ya que esto causaba un costoso proceso de paso de datos de GPU a CPU, por lo que en esta aplicación se buscó prescindir de la CPU y delegar todos los cálculos (SDFs y el algoritmo de Marching Cubes) y el renderizado de la geometría procedural a la GPU utilizando la función DrawProceduralIndirect de Unity. Todo esto permitió liberar una carga enorme de la CPU y enfocarla netamente a cálculos propios del juego y las interacciones del usuario con el mismo, lo cual se tradujo en una alza en el rendimiento considerable, incluso con múltiples geometrías y deformación en tiempo real.

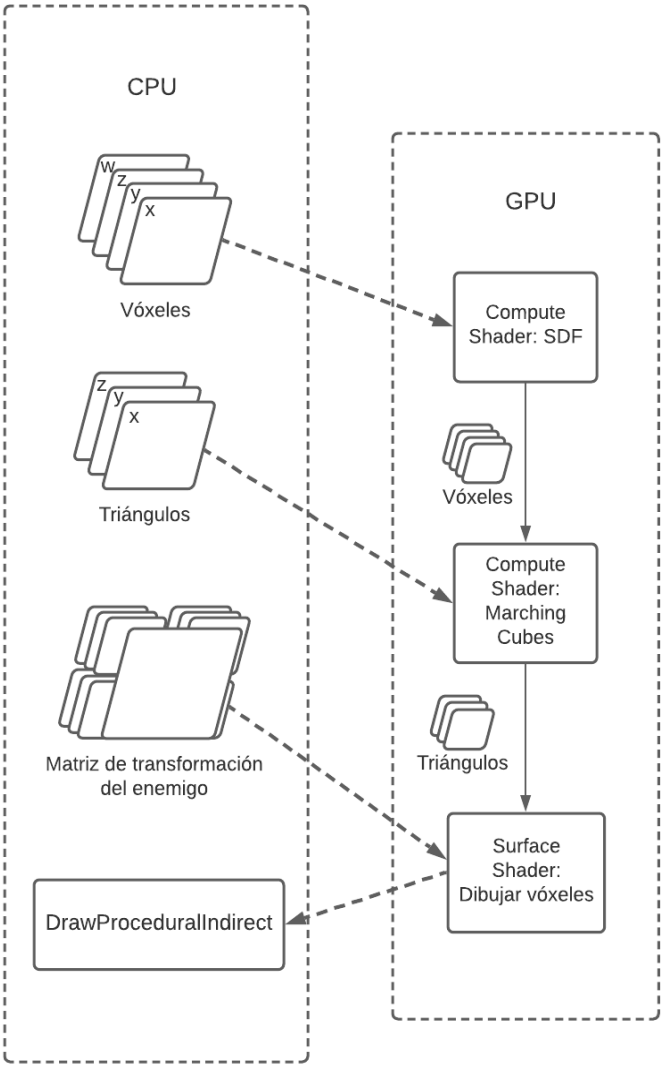

Figura 05: Esquema implementación aplicación 4

En base a los resultados de cada una de las aplicaciones mencionadas, se realizó el siguiente cuadro comparativo teniendo en cuenta 5 métricas base para determinar el consumo de recursos:

CD: Con deformación, SD: Sin deformación

	Aplicación 1		Aplicación 2		Aplicación 3		Aplicación 4	
Métricas	CD	SD	CD	SD	CD	SD	CD	SD
Frame rate (FPS)	~115	~750	~191	~618	~248.9	~587.9	~732.5	~812.8
Procesamiento CPU (ms)	~36.09	~9.00	~24.15	~6.10	~19.45	~6.47	~22.57	~5.25
Procesamiento GPU (ms)	~35.89	~5.72	~16.57	~3.86	~14.73	~4.45	~18.96	~2.95
Triángulos procesados	2000	2000	19.5k	20.5k	21.5k	22.5k	1000	1000
Memoria RAM (GB)	~1.20		~1.54		~1.29		~1.18	
Cantidad de geometría (número de objetos)	1		5		1		6	

*Las pruebas se realizaron dentro de Unity utilizando la herramienta profiler, lo que generó una carga extra en el rendimiento, la cual era significativa pero constante entre aplicaciones; los resultados no restan las cargas mencionadas

Tabla 02: Cuadro comparativo de los 4 métodos

Como se puede observar, la aplicación 4 fue la que mejores resultados generó con gran diferencia; el delegar todo los cálculos costosos a la GPU (incluyendo el dibujado de la malla resultante) hizo que el frame rate fuese muchísimo más estable que cualquiera de las demás aplicaciones, incluso manejando más geometría y realizando múltiples operaciones entre ellas, permitiendo así que la CPU pueda encargarse de toda la parte lógica del juego sin preocuparse por posibles problemas de rendimiento causados por la generación e interacción con vóxeles.

Finalmente la implementación resultante fue la siguiente:

Los vóxeles se representaron como un Compute Buffer escalar de 20^3 vóxeles de tipo float4 (flotantes de 4 dimensiones: X, Y, Z y W) el cual se enviaba a un Compute Shader en donde se determinaba la forma de una

isosuperficie especificada por código utilizando SDFs de múltiples primitivas y operaciones entre ellas; los resultados de estas operaciones se guardaban en el Compute Buffer, se retornaba y luego se enviaba a un segundo Compute Shader en donde el algoritmo de Marching Cubes se encargaba de extraer una malla poligonal a partir de la isosuperficie establecida, malla que finalmente es renderizada gracias a la función DrawProceduralIndirect de Unity y un Surface Shader especial que le otorgaba un color uniforme y permitía que proyectara y recibiera sombras de los objetos en la escena. En la figura 05 se aprecia el esquema de esta implementación.

2.2 DOCUMENTO DE DISEÑO

2.2.1 CONCEPTO

Prototipo de videojuego de acción para plataforma PC enfocado a la comparación del uso de un personaje basado en vóxeles y uno basado en mallas poligonales.

El objetivo principal del videojuego es la interacción entre dos personajes, uno de ellos es el personaje principal controlado por un usuario y el segundo personaje es un enemigo el cual tiene que ser destruido por el jugador.

2.2.2 OBJETIVOS DEL JUGADOR

El jugador debe destruir al enemigo disparándole con su arma láser hasta que su barra de salud quede vacía, intentando huir de los ataques del enemigo y las amenazas del entorno en el proceso.

2.2.3 JUGABILIDAD

La vista es en 3ra persona. El personaje se mueve en 8 direcciones distintas según las flechas direccionales del teclado o las teclas W, A, S y D, y la dirección hacia adelante del movimiento está determinada por la rotación horizontal del ratón. La cámara está ubicada detrás del personaje principal apuntando siempre a un punto en su cabeza sobre el cual rota y esta depende del movimiento del ratón del PC, con la posibilidad de 180 grados de libertad en el eje vertical y 360 grados en el horizontal.

El personaje debe disparar con su arma láser al enemigo con el fin de destruirlo y terminar satisfactoriamente el juego. Los disparos se realizan manteniendo el click izquierdo del ratón y el personaje apunta manteniendo el click derecho del ratón. El personaje puede disparar si y sólo si el usuario está apuntando (manteniendo el click derecho del ratón), de lo contrario es imposible emitir disparos. El arma cuenta con un medidor de calentamiento que va disminuyendo a medida que el jugador dispara y se rellena cuando este deja de hacerlo. Si el jugador sigue disparando y el medidor queda vacío, el arma se sobrecalienta y no puede ser disparada hasta que el medidor esté completamente lleno otra vez, si intenta disparar estando sobrecalentada el medidor no se carga.

2.2.4 MECÁNICAS DE JUEGO

2.2.4.1 ACCIONES DEL JUGADOR

-Modo de juego: El jugador puede escoger qué modo de juego quiere probar primero (modo vóxel o modo malla).

- Salir: El jugador puede salir del juego.

Durante la batalla:

- Pausar: El jugador puede pausar el juego.

- Reanudar: El jugador puede continuar con el juego.

- Volver al menú: El jugador puede salir a la vista principal del juego.

- Salir: El jugador puede salir del juego.

- Manipular la Cámara: La vista del jugador es en tercera persona con 360° de libertad en el eje horizontal y 180° de libertad en el eje vertical, manipulada con el ratón del PC.

2.2.4.2 ACCIONES DEL PERSONAJE PRINCIPAL

- Desplazamiento: El personaje puede caminar en dirección adelante, atrás, izquierda y derecha con velocidad constante utilizando las teclas W, A, S y D. La dirección frontal de desplazamiento está determinada por la rotación horizontal del ratón.

- Apuntar: El personaje puede activar un acercamiento de cámara al mantener oprimido el click derecho del ratón teniendo 180 grados de libertad de apuntado en el eje vertical y 360 grados en el horizontal determinados por el movimiento del ratón.

- Disparar: El personaje tiene un arma láser que puede ser disparada manteniendo oprimido el click izquierdo del ratón (siempre y cuando esté apuntando al mismo tiempo). El arma tiene un medidor de calentamiento que va disminuyendo cada vez que se dispara y se regenera lentamente cuando deja de hacerlo. Si el personaje sigue disparando hasta que el medidor se vacíe, el arma se sobre calienta y el usuario no puede disparar hasta que este se enfríe nuevamente, si intenta seguir disparando estando sobrecalentada el arma no se enfría hasta que deje de disparar.

- Morir: El personaje puede morir durante el juego al hacerse daño por disparos del enemigo o por entrar en contacto con amenazas del entorno, agotando así su número de vidas de 3 hasta que llegue a 0.

2.2.4.3 ACCIONES DEL PERSONAJE ENEMIGO

- Rotación sobre el eje: El enemigo puede rotar sobre sí mismo para mantener al personaje principal en su campo de visión.

- Disparar: El enemigo puede disparar al personaje principal 7 proyectiles cada determinado tiempo.

-Armadura (vóxeles): El enemigo cuenta con una armadura que recubre todo su esqueleto y es deformada cada vez que el personaje principal le dispara además de recibir daño a su barra de vida.

-Armadura (malla poligonal): El enemigo cuenta con una armadura que recubre todo su esqueleto y recibe daño a su barra de vida cada que el personaje principal le dispara.

- Morir: El enemigo cuenta con una barra de salud que disminuye cada vez que el personaje principal dispara a su armadura. Cuando esta llegue a 0 el enemigo muere.

2.2.5 CASOS DE USO

Los casos de uso están desarrollados para explicar el comportamiento del personaje jugable, enemigo y el videojuego como tal dependiendo de las acciones que realice el usuario durante su experiencia en ambas versiones del prototipo.

Mapa de navegación

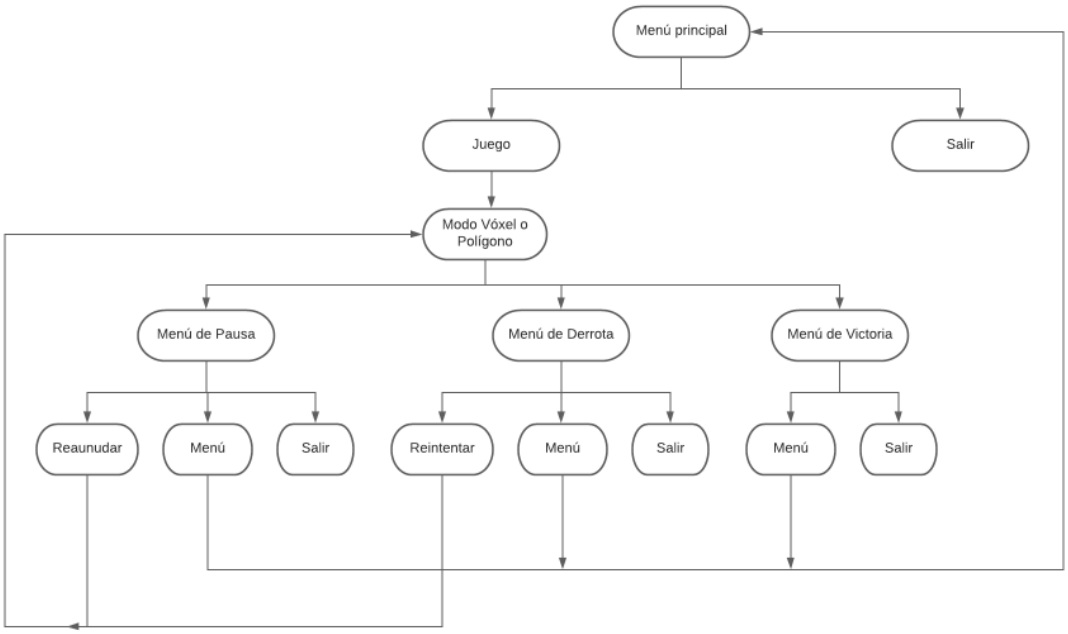

Figura 06: Mapa de navegación

Casos de uso de interacción

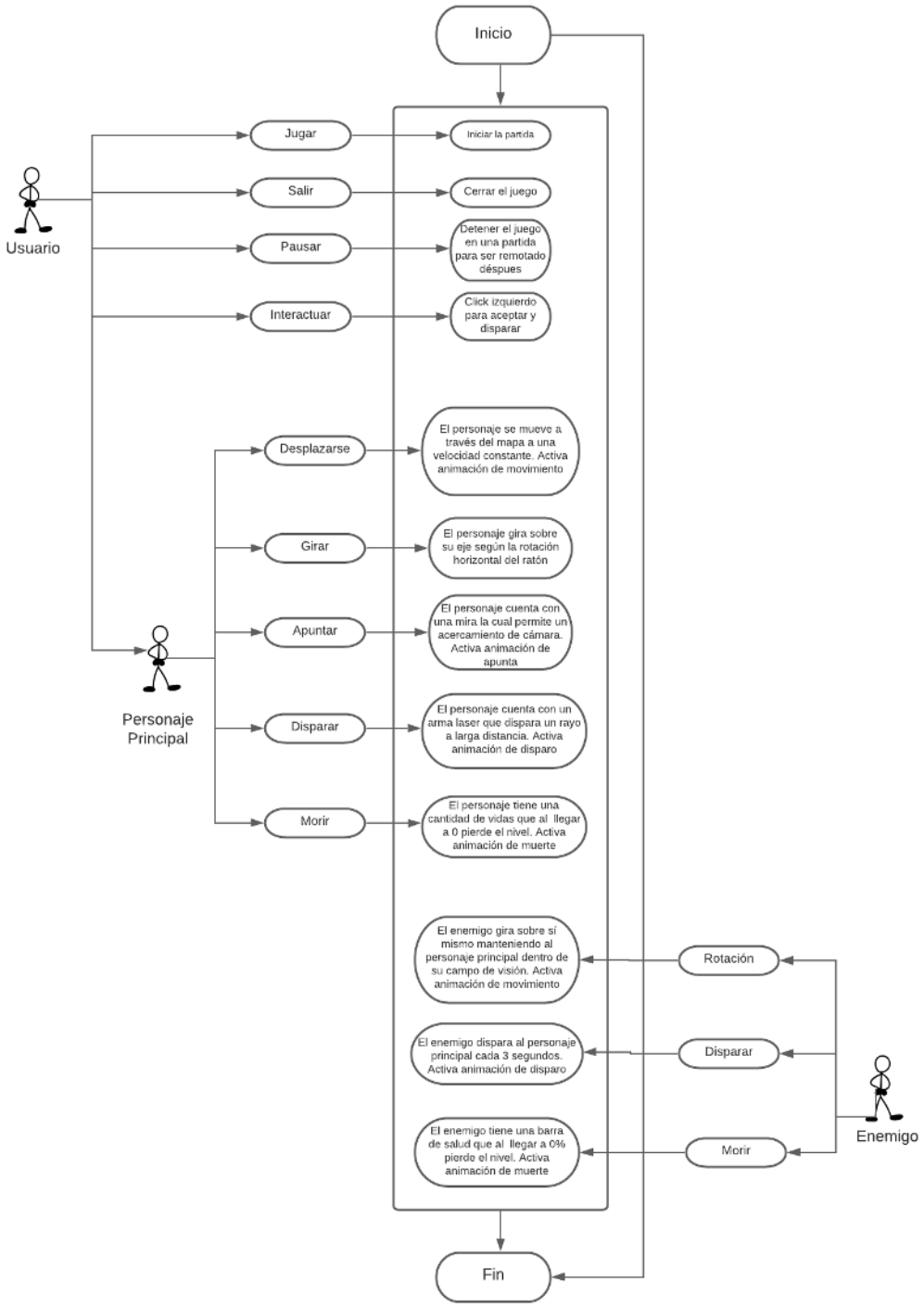

Figura 07: Caso de uso de interacción

Casos de uso durante el juego (versión malla)

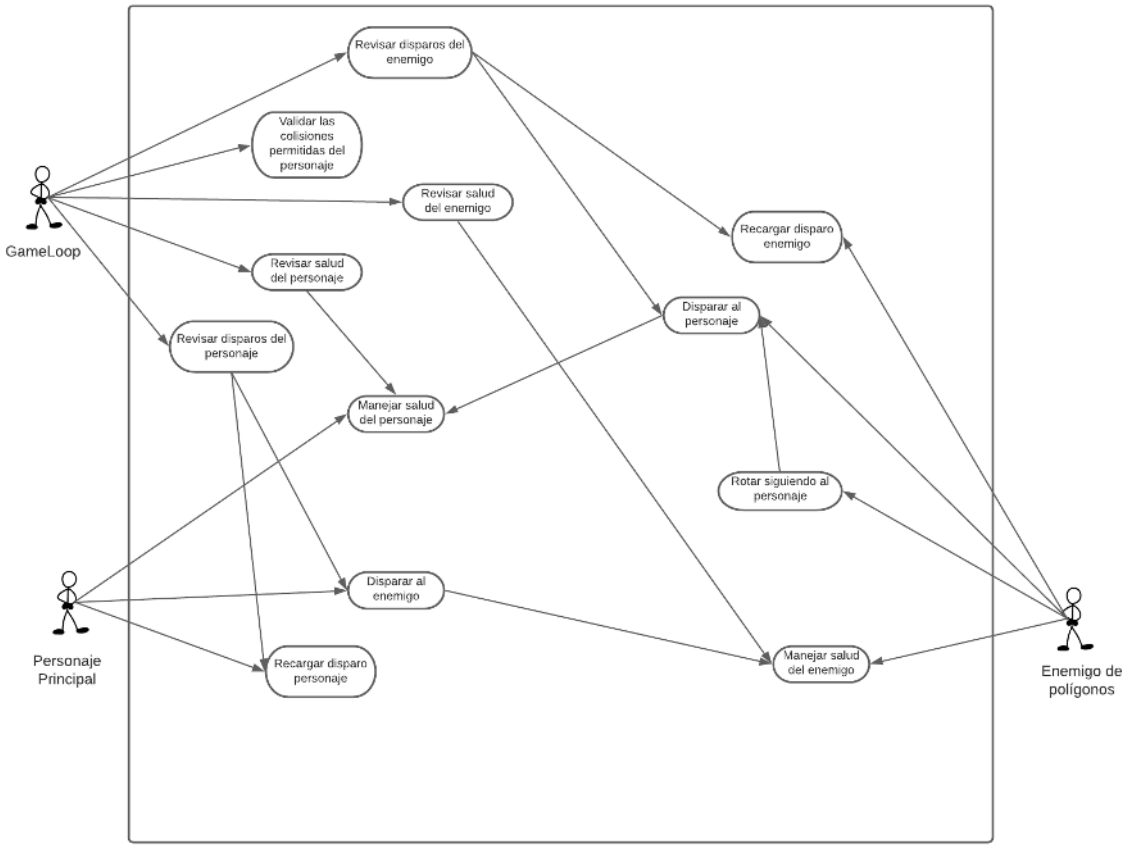

Figura 08: Caso de uso durante el juego (versión malla)

Casos de uso durante el juego (versión vóxel)

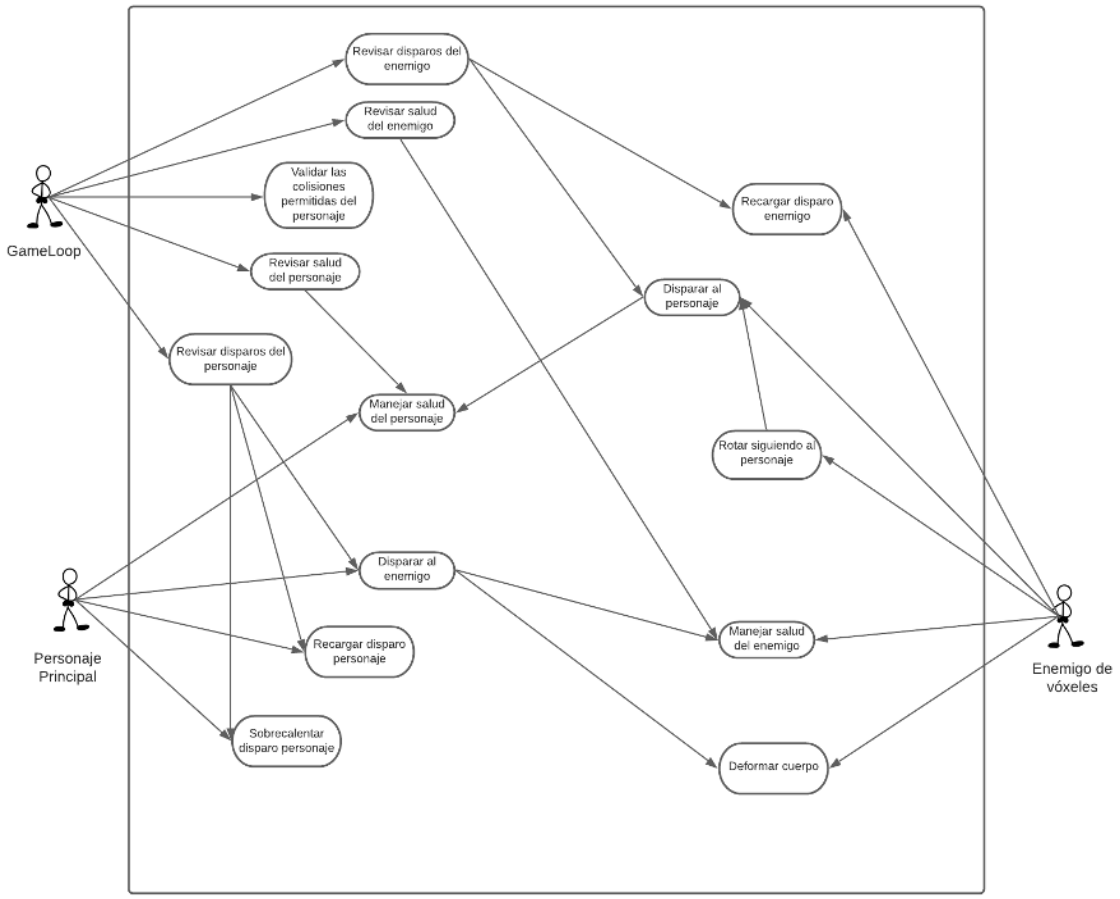

Figura 09: Caso de uso de interacción (Versión vóxel)

2.2.6 TIPO DE ESCENARIO

La batalla se desarrolla en un único escenario, sobre la superficie de un planeta desconocido donde se estrelló la nave en la que se encontraba el personaje principal (astronauta), un lugar oscuro y desolado, este tiene cráteres llenos de lava, custodiado por el personaje enemigo quien busca destruir a cualquiera que invada su morada.

2.2.7 ESTILO GRÁFICO

2.2.7.1 Referencias de arte

Figura 10: Collage referencias de arte

Referencias del personaje principal

Figura 11: Referencias de personaje principal

Referencias del enemigo

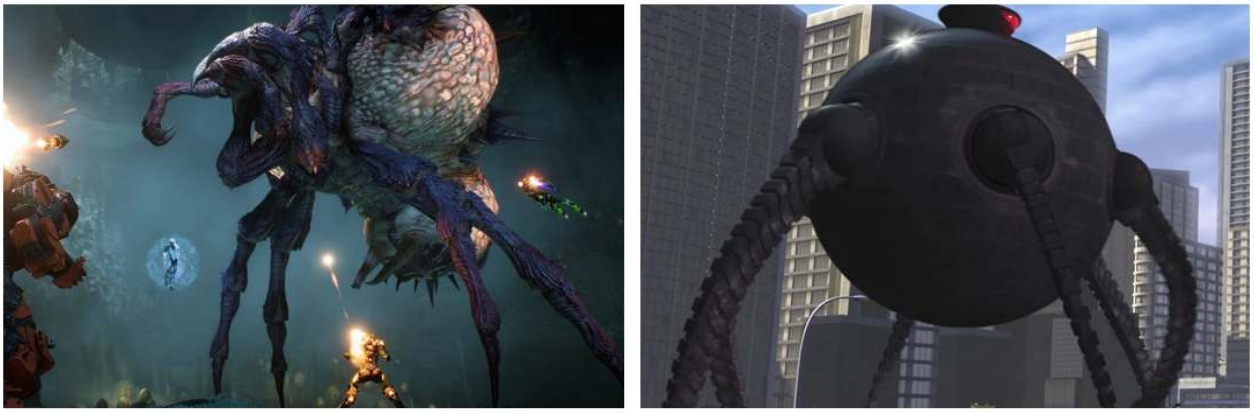

Figura 12: Referencias de personaje enemigo

2.2.7.2 Arte conceptual

Model Sheet enemigo:

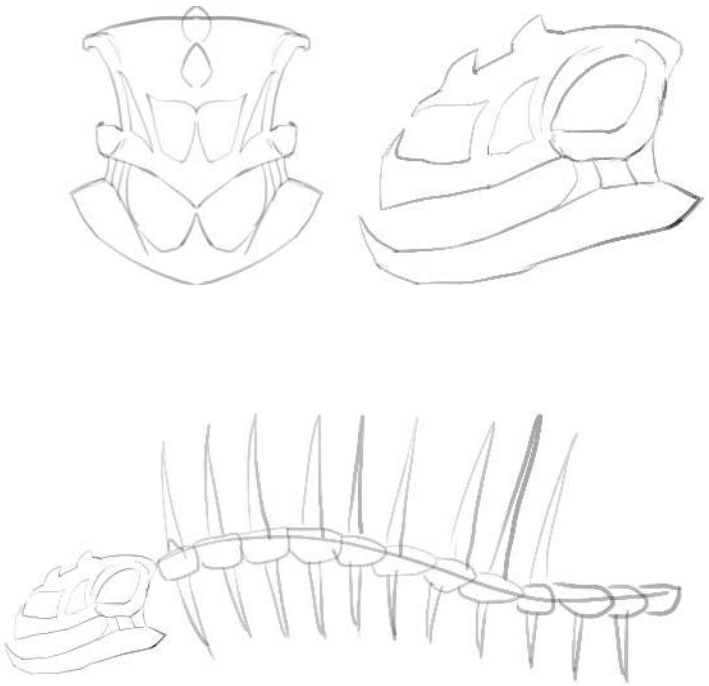

Figura 13: Model Sheet personaje enemigo

Bocetos Del personaje principal:

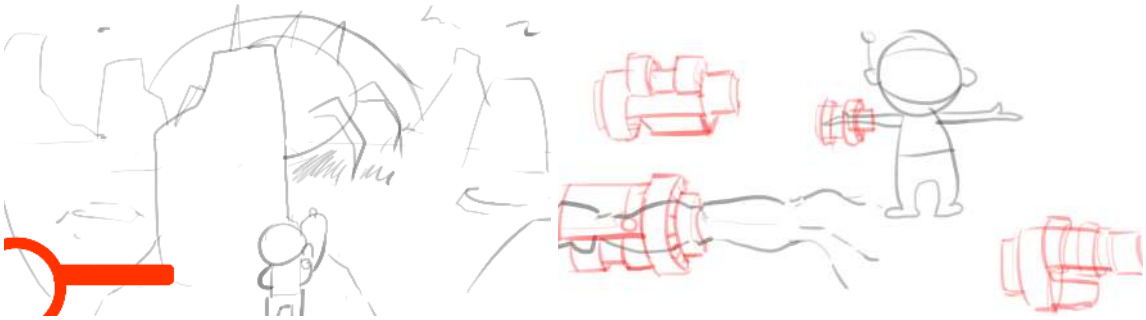

Figura 14: Bocetos personaje principal

2.2.7.3 Modelos 3D

Esqueleto personaje enemigo / Cuerpo personaje enemigo

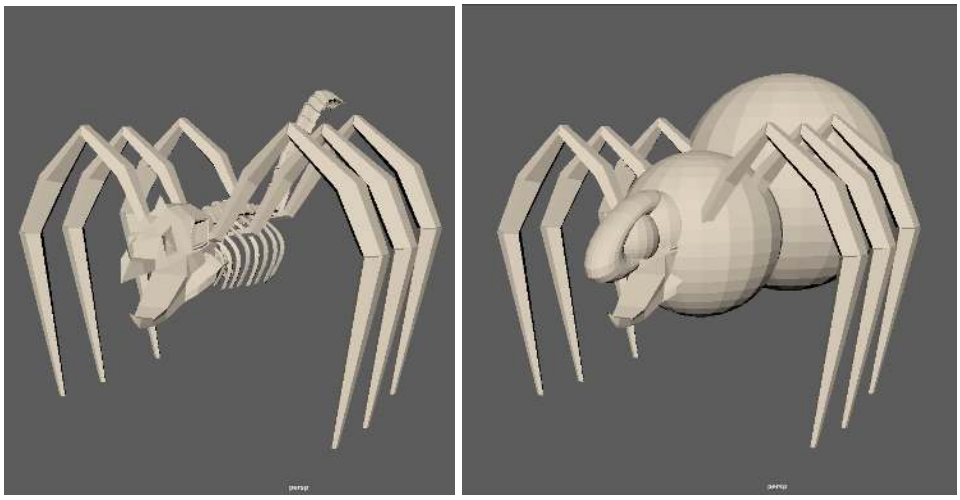

Figura 15: Modelo 3D personaje enemigo

Personaje jugable

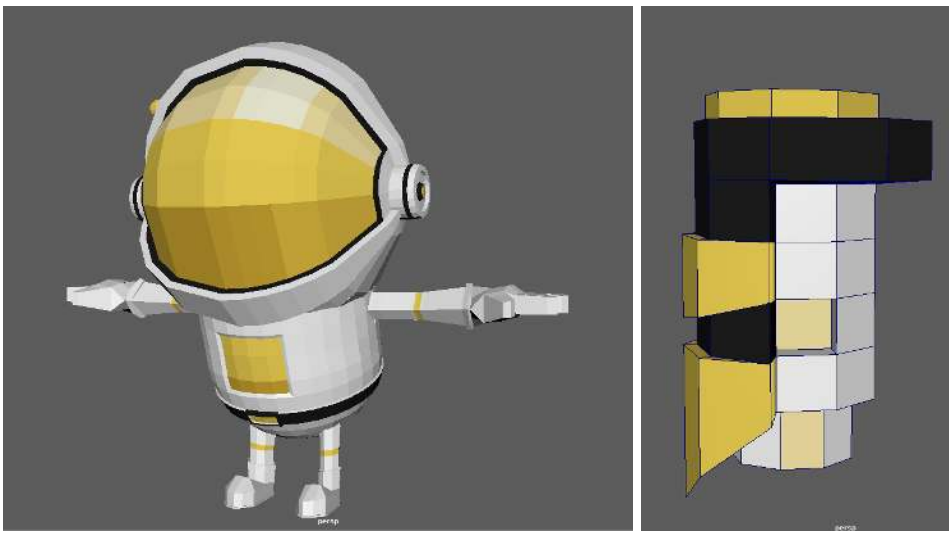

Figura 16: Modelo 3D personaje principal

2.2.8 INTERFAZ GRÁFICA

2.2.8.1 HUD

Vida vacía

Vida completa

Figura 17: HUD vidas del personaje principal

Barra de vida enemiga, carga láser, vidas de personaje principal

Figura 18: HUD Barra de vida enemiga, carga láser y vidas del personaje principal

2.2.8.2 MOCKUP

Menú

Figura 19: Vista menú principal

Vista de pausa / Vista de victoria

Figura 20: Vistas Pausa y victoria

Vista de pérdida de juego / Pantalla de carga

Figura 21: Vistas pérdida de juego y pantalla de carga

2.3 IMPLEMENTACIÓN Y PROTOTIPADO

Cuando se finalizó el documento de diseño y se tuvo claro el método de renderizado de vóxeles a utilizar, se procedió a elaborar el prototipo de videojuego planteado en los objetivos. Para el desarrollo se utilizaron los paquetes Cinemachine [57], Postprocessing stack [58] y FastIK [59] de Unity que permitieron más facilidades al implementar ciertas mecánicas y efectos visuales.

Una vez completado el modelo del personaje jugable, se le elaboró un rig automático en la herramienta Mixamo con ajustes manuales y pintado de pesos (skinning) en Blender 3D; en Mixamo también se descargaron diversas animaciones de acuerdo a las acciones establecidas en el documento de diseño. Con todo listo se importó el modelo y sus assets en Unity para programar funciones que le permitieran moverse por el entorno y colisionar, gestionar sus animaciones, apuntar, disparar rayos, recibir daño y morir.

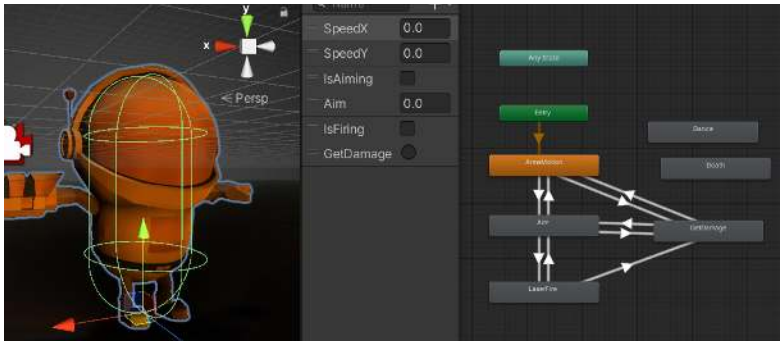

Figura 22: Rig personaje principal

Para facilitar la implementación del cuerpo del enemigo en mallas poligonales y vóxeles se decidió que el enemigo iba a estar compuesto de un esqueleto base (modelado con mallas poligonales), y el cuerpo “externo” fuese en malla o en vóxeles iba a emparentarse a dicho objeto base, obteniendo como resultado que el cuerpo heredaba las transformaciones del esqueleto, con lo cual para realizar ambas versiones del prototipo únicamente debía duplicarse toda la escena del juego y reemplazar el cuerpo externo por su versión en malla o vóxel respectivamente.

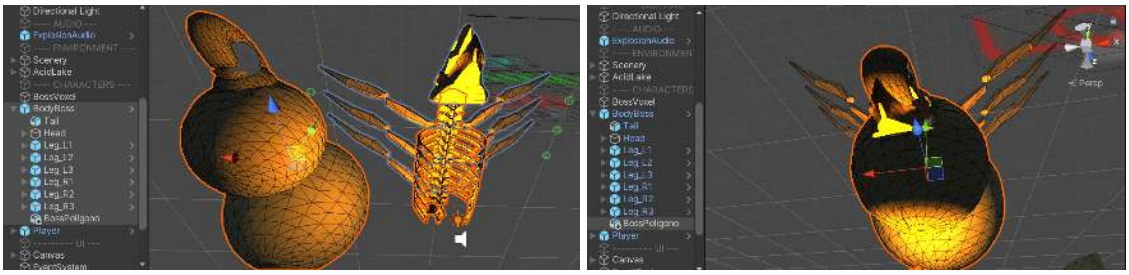

Figura 23: Versión malla y vóxel del personaje enemigo

Las animaciones y locomoción en general del personaje enemigo se realizaron de forma procedural (completamente por código), esto debido a que de cierto modo fue más sencillo implementarlo de esta manera por la naturaleza del personaje y la complejidad de su movimiento a la hora de animarlo manualmente. Para lograr la locomoción se utilizó el paquete FastIk, que es una implementación del algoritmo FABRIK [60] para cinemática inversa (IK, Inverse Kinematics en inglés). La IK es una técnica que calcula el movimiento requerido u óptimo de un sistema conectado de objetos para llegar a un destino determinado [61], en este caso dichos objetos son las partes del esqueleto del modelo enemigo (torso, cola, cabeza, patas).

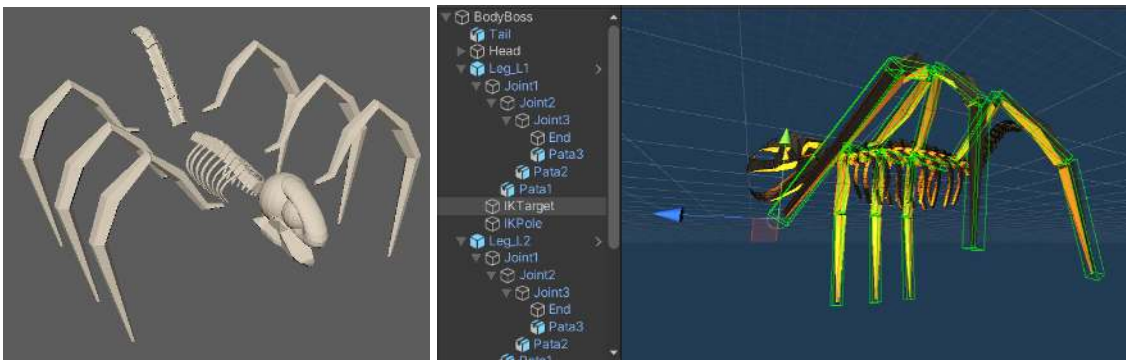

Figura 24: Modelo 3D del personaje enemigo

Los comportamientos del enemigo se administraron a partir de dos códigos importantes: uno encargado de actualizar los movimientos de cada pata y el otro encargado de actualizar los movimientos del cuerpo y de provisionar de nuevos datos a los parámetros necesarios para producir los movimientos de cada pata. A partir de estos dos códigos se crearon las funciones de persecución, ataque y muerte que se establecieron en el documento de diseño como acciones del enemigo, siempre teniendo presente que este debía estar en el centro del escenario y sólo podía girar sobre sí mismo.

La versión del cuerpo del enemigo en malla poligonal convencional se modeló en Autodesk Maya y se elaboró de forma tal que su geometría fuese extremadamente sencilla para que al modelarlo con vóxeles el resultado quedase lo más parecido posible. El modelado de los vóxeles se llevó a cabo utilizando SDFs y operaciones entre estos con base en [50]; las figuras primitivas utilizadas fueron las siguientes: 3 esferas (cráneo, cola y torso), 1link (hocico), 1 elipsoide (nariz) y la operación de Smooth Union (unión suave) para comprimir todas las geometrías en una de acuerdo a la distancia entre ellas y un factor arbitrario de unión.

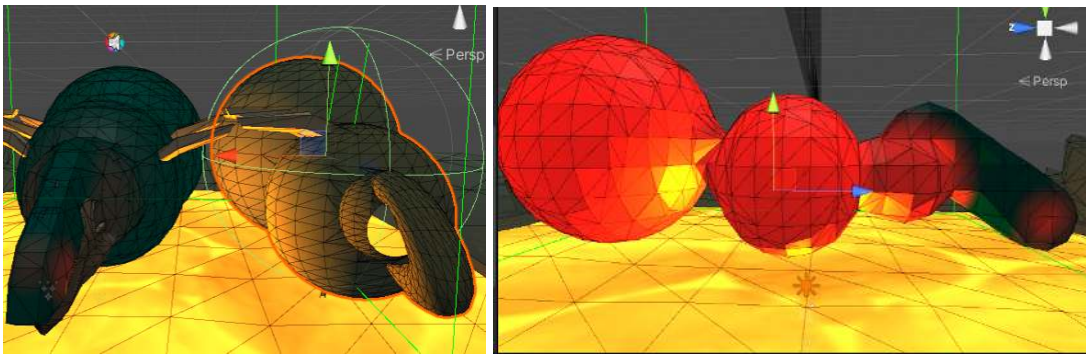

Figura 25: Cuerpo externo versión malla y versión vóxel (SDF)

Para generar el cuerpo de vóxeles se utilizó enteramente la GPU, todo mediante dos Compute Shaders y un Surface Shader. El 1er Compute Shader recibía un Compute Buffer de 20^3 vóxeles y una posición en el mundo; el Shader se encargaba de calcular el modelo del cuerpo a partir de las operaciones entre SDFs; guardar los resultados en el Buffer y finalmente retornarlo; este proceso solo se llevaba a cabo una vez (en cuanto iniciaba el juego), el resto del tiempo retornaba el Buffer sin modificarlo, a menos que hubiese ocurrido algún cambio (deformación).

El 2do Compute Shader recibía el Buffer con los resultados de los SDFs y un ComputeBuffer de triángulos; este Shader se encargaba de ejecutar el algoritmo de Marching Cubes implementado con base en [51] sobre los vóxeles; el algoritmo calculaba las posiciones y las normales de los vértices que conformaban la superficie del cuerpo del enemigo y luego los guardaba en el Buffer de triángulos que finalmente se retorna; los triángulos se definen como un struct de 3 vértices, y los vértices se definen como un struct de posiciones y normales.

El Surface Shader recibía el Buffer de triángulos generados por Marching Cubes y una matriz de transformación del enemigo; este Shader se encargaba de dibujar los vóxeles en la escena, mapeando los vértices de la superficie generada en coordenadas de la escena (multiplicando la matriz de transformación del enemigo por la posición de los vértices) y otorgándole un color uniforme al cuerpo externo; junto a esto se tuvo que indicar manualmente que el Shader debía renderizar sombras puesto que por defecto no lo hacía. (En la figura 05 se aprecia el esquema de esta implementación)

La malla al ser generada completamente en la GPU imposibilitó el usar directamente el motor de física de Unity para detectar colisiones en ella, así que se crearon 4 colisionadores de esfera y se emparentaron a las partes en donde el jugador podría disparar (la cola, el torso, el “cuello” y el hocico). Estos colisionadores sólo eran afectados por el láser del jugador, además eran de tamaño dinámico (entre más chocaba el láser, más disminuía su radio) para que a medida que se fuera deformando el cuerpo del enemigo, su área de disparo reducía su tamaño también.

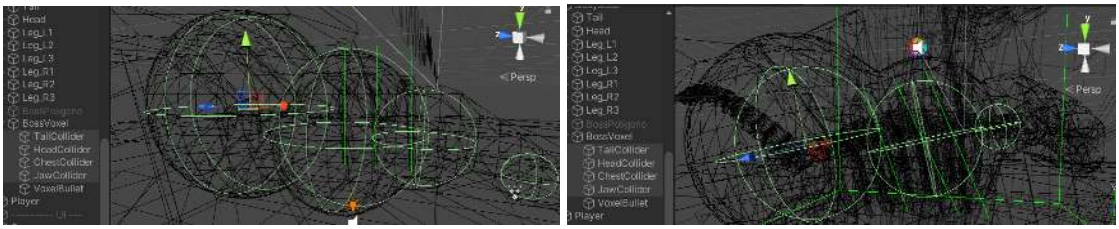

Figura 26: Colisionadores personaje enemigo

Para la deformación se utilizó una operación de Smooth Subtraction (sustracción suave) de SDFs de una esfera con radio arbitrario sobre la malla de vóxeles; la posición de dicha esfera era determinada por raycasting de acuerdo al punto de colisión del láser del jugador hacia alguno de los 4 colisionadores de esfera en el cuerpo del enemigo.

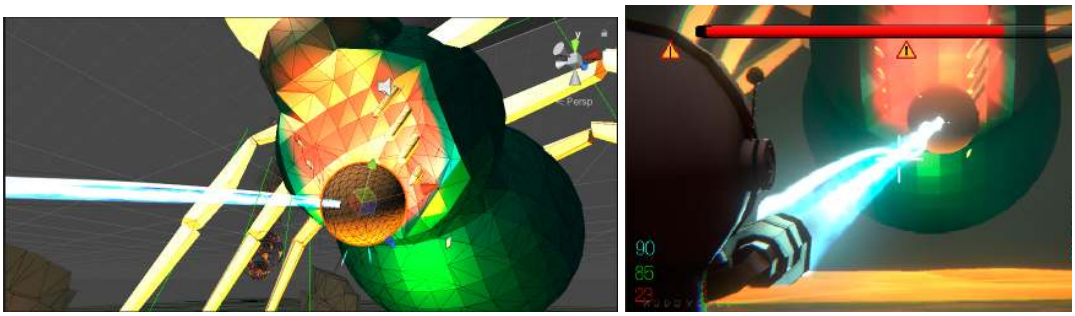

Figura 27: Deformación con colisionador personaje enemigo

El cambio de color entre la parte externa del cuerpo y la interna se obtuvo a partir de una interpolación lineal de 2 colores arbitrarios de acuerdo a la distancia entre los vértices de la malla del cuerpo externo y el centro de las esferas de colisión; entre más alejados (más afuera) estuviesen los vértices, más se inclinaría al 1er color esa parte de la malla y viceversa.

Figura 28: Deformación con colisionador personaje enemigo 2

3. PRUEBAS

Teniendo como base el enfoque del proyecto, se tomaron dos tipos de pruebas; una de ellas es del rendimiento que tiene cada una de las dos versiones del prototipo en un equipo con características específicas; y el otro tipo de prueba se realizó con un grupo de usuarios con el fin de analizar su percepción respecto a las dos variantes del videojuego.

3.1 PRUEBAS DE RENDIMIENTO

Las pruebas de rendimiento del proyecto se realizaron sobre el archivo ejecutable final, utilizando el profiler de Unity. Los parámetros que se tuvieron en cuenta para las pruebas fueron los siguientes:

- tiempo más alto y más bajo de ejecución de la CPU en un frame.
- tiempo más alto y más bajo de ejecución de la GPU en un frame.
- número más alto y más bajo de triángulos renderizados por la CPU en un frame.
- cantidad de memoria RAM consumida por la aplicación.

Se despreció la cantidad de cuadros por segundo pues en ambos modos el prototipo iba a 60 estables, si llegaba a bajar no pasaba de 57 y cuando subía no pasaba de 65. Las pruebas se realizaron en el siguiente equipo:

Equipo	**Procesador**	**Ram**	**GPU**
Laptop Dell Inspiron 7577	Intel Core i7-7700HQ	16GB	NVIDIA GeForce GTX 1060 Max-Q 6 GB VRAM

Tabla 01: Ficha técnica equipo

La forma en que se llevaron a cabo fue ejecutando el build de desarrollo del proyecto al mismo tiempo que el Profiler en Unity utilizando ambos modos (vóxel y malla) uno tras otro obteniendo los siguientes resultados:

	Versión Vóxel		Versión Malla Convencional	
Métricas	Pico	Valle	Pico	Valle
Procesamiento CPU (ms)	~18.08	~15.22	~19.16	~14.73
Procesamiento GPU (ms)	~14.88	~12.91	~13.86	~12.20
Triángulos procesados	115.7k	68.6k	144.4k	82.9k
Memoria RAM (MB)	301		295	

Tabla 03: Comparativa entre versiones (Vóxel vs Malla)

De acuerdo a los resultados presentados se puede observar que la diferencia de rendimiento en ambas versiones del prototipo es prácticamente despreciable, teniendo que la cantidad de memoria RAM utilizada en la versión de vóxeles es 2% mayor con respecto a la de mallas poligonales, además que el tiempo de procesamiento de la CPU y GPU solo dista en prácticamente 1ms, tanto en momentos de mayor necesidad de computo como en los de menor. La mayor discrepancia en ambas versiones es el hecho de que al renderizar los triángulos generados por el algoritmo de Marching Cubes directamente en la GPU, causa que la CPU llegue a procesar hasta casi un 20% menos triángulos, con lo cual puede enfocarse en ejecutar la lógica como tal del juego sin necesidad de preocuparse por el renderizado del personaje enemigo y sus deformaciones.

3.2 PRUEBAS DE USUARIO

Con el objetivo de analizar la percepción que tenían lo usuarios con respecto a las dos versiones del prototipo, se realizó un test con un grupo de personas que estuvieran dentro del target (conocedor de videojuegos, jugador frecuente y desarrollador de videojuegos) además de contar con una tarjeta gráfica dedicada debido a que el proyecto en su versión de vóxeles específicamente hacía un basto uso de Compute Shaders y múltiples cálculos en la GPU en general. Debido a la emergencia sanitaria mundial, las pruebas se realizaron de forma virtual por lo cual fue más complicado conseguir personas ya que fue casi imposible tener el control sobre el hardware en el que se ejecutaría el proyecto, así que en la medida de lo posible se preguntó previamente a los diferentes usuarios con qué tarjeta gráfica contaban para corroborar si podrían ejecutar la aplicación sin inconvenientes.

De 25 usuarios que accedieron a realizar la prueba, se descartaron 8 de ellos debido a que los vóxeles no se renderizaban correctamente en tarjetas gráficas AMD/ATI, y mucho menos en tarjetas integradas (la familia Intel HD Graphics), solo en tarjetas NVIDIA. De las referencias de GPU que se utilizaron para las pruebas, la de menor capacidad fue una GeForce 840M, y la de mayor una GeForce RTX 2080 SUPER.

A los usuarios confirmados (17) se les envió un archivo ejecutable del prototipo y una encuesta para diligenciar una vez terminada la experiencia con el videojuego; la instrucción que se les comunicó fue que tuvieran una victoria en ambos modos de juego (sin mencionarles ninguna clase de detalle acerca del prototipo que iban a experimentar) y posteriormente realizaran la encuesta. Ver anexo #2 Formulario entrevista con usuarios

Los resultados de la encuesta fueron los siguientes:

Pregunta: Experiencia con videojuegos:
Respuestas: Conocedor de videojuegos.(2) Jugador frecuente.(14) Desarrollador de videojuegos.(1)

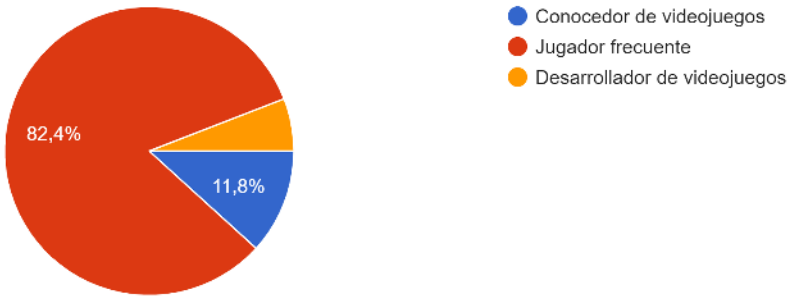

Figura 29: Resultado primera pregunta encuesta gráfico torta

Pregunta: Se observan diferencias marcadas en la visualización del personaje enemigo en cada una de las versiones:
Respuestas: 1.(0) 2.(1) 3.(1) 4.(5) 5.(10)

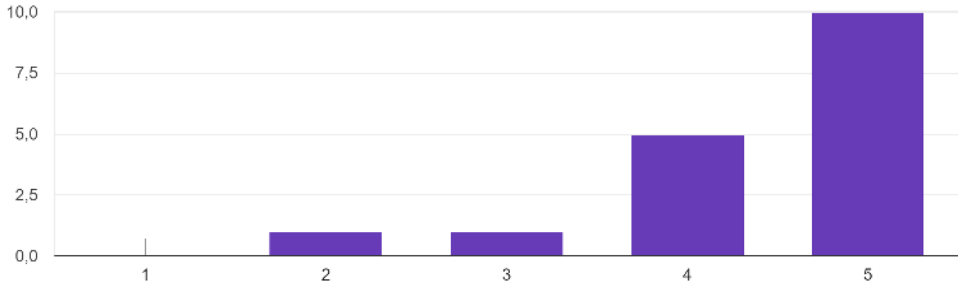

Figura 30: resultado segunda pregunta encuesta (gráfico barras)

Pregunta: Se observan diferencias en la percepción de movimiento del personaje enemigo entre las dos versiones :

Respuestas: 1.(2) 2.(5) 3.(3) 4.(5) 5.(2)

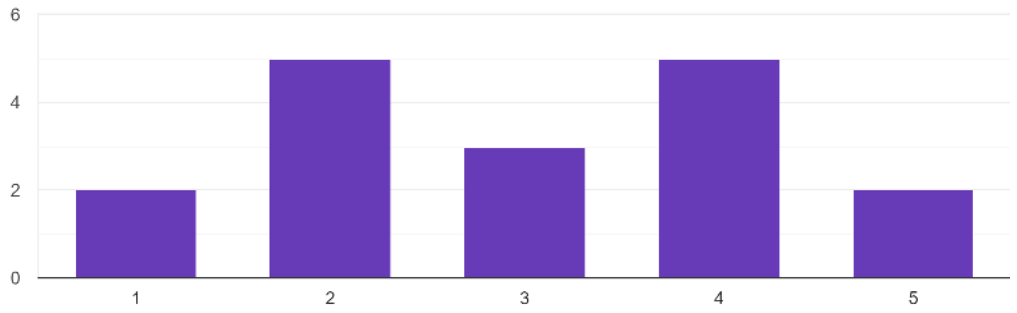

Figura 31: Resultado tercera pregunta encuesta (gráfico barras)

Pregunta:Se observan diferencias en la reacción del personaje enemigo cuando recibe el disparo del personaje jugable:

Respuestas: 1.(1) 2.(4) 3.(1) 4.(5) 5.(6)

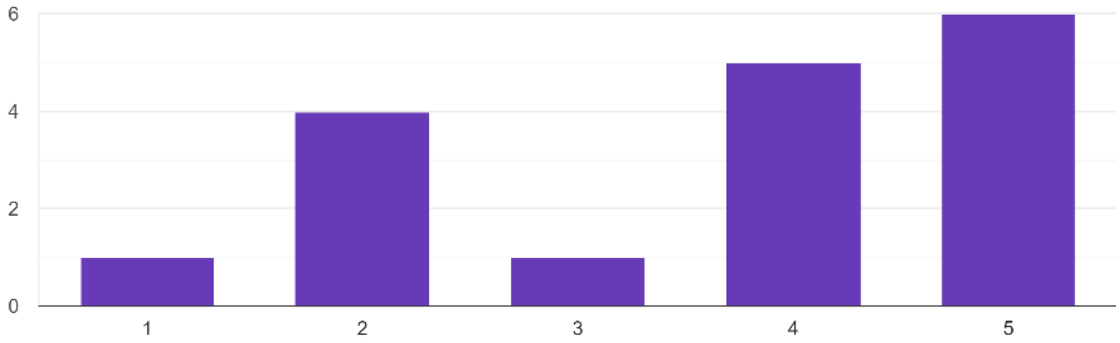

Figura 32: Resultado cuarta pregunta encuesta (gráfico barras)

Pregunta: Considerando su experiencia de juego y de visualización de las dos versiones del prototipo, cuál considera usted más llamativa e interesante:

Respuestas: Versión Vóxel.(12) Versión Malla.(3) Se perciben igual.(2) Ninguna.(0)

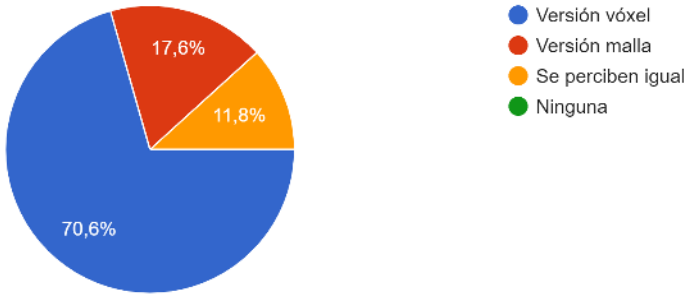

Figura 33: Resultado quinta pregunta encuesta (gráfico barras)

Pregunta: ¿Percibió usted objetos deformables en alguna de las dos versiones?:

Respuestas: Versión Vóxel.(8) Versión Malla.(2) Se perciben igual.(3) Ninguna.(4)

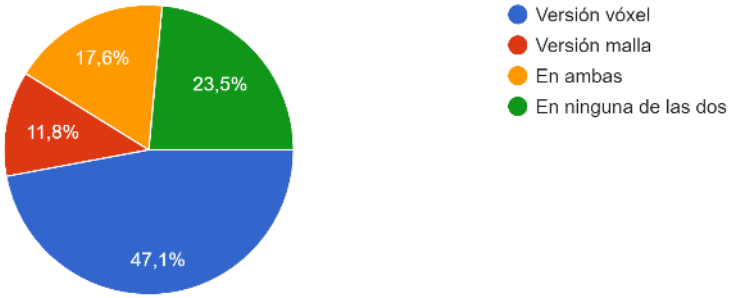

Figura 34: Resultado sexta pregunta encuesta (gráfico barras)

Las respuestas abiertas se pueden encontrar en el anexo #7.

Además de los 17 usuarios que probaron el prototipo y diligenciaron la encuesta, se realizó también una entrevista al desarrollador experto Nicolás Nieto en el sector de los videojuegos, el cual tiene 6 años de experiencia y actualmente labora como lead programmer en la empresa multinacional Teravision Games.

3.3 RESULTADOS Y ANÁLISIS

Con respecto al rendimiento técnico, el resultado encontrado fue de cierto modo el esperado puesto que puede observarse cómo las diferencias más notorias entre ambas versiones del prototipo se encuentran en la cantidad de triángulos que procesa la CPU (~30.000 triángulos menos en vóxeles), el tiempo que tarda la GPU en procesar un frame (~1ms menos en malla) y la cantidad de memoria que consume la aplicación (específicamente la memoria utilizada por el driver gráfico GfxDriver, ~5MB menos en malla, y la memoria RAM, ~6MB menos en malla).

Estas diferencias se dan debido a que la versión de vóxeles delega todo el procesamiento pesado (el cálculo de la geometría procedural y su dibujado en tiempo real) a la GPU, dando como resultado que el uso de la CPU en ambas versiones sea muy similar (siendo menor por ~1ms en vóxeles) puesto que esta se encarga principalmente de gestionar funciones y mecánicas propias del juego, si por el contrario los vóxeles se hubiesen generado y renderizado utilizando la CPU muy seguramente los resultados se irían completamente a favor de la versión de mallas poligonales convencionales y el rendimiento de la versión de vóxeles sería tan bajo que el juego sólo podría llegar a ejecutarse bien en hardware muy potente.

La principal desventaja que tuvieron los vóxeles fue el hecho de que el número de dispositivos que podían soportarlos eran mucho más reducidos ya que dependían mucho de una GPU dedicada y solo estaban ejecutándose correctamente en tarjetas NVIDIA, siendo esto un gran problema sobretodo si se pretendiera llevar una versión finalizada del juego a múltiples plataformas.

Con respecto a la percepción de los usuarios, basándonos en las respuesta obtenidas en la encuesta se puede observar que el 88.4% (15) de los usuarios sí percibieron una diferencia marcada dando a entender que los vóxeles sí generaron un impacto visual en los usuarios, y el 53.3% (8) de esos usuarios sí se percataron de que el enemigo en el modo vóxel se deformaba, además, los comentarios de los usuarios hacían énfasis en que se modifica el modelo del enemigo e incluso le agregaba más realidad e interacción al juego.

Hubo varios usuarios que si bien percibieron una diferencia marcada en ambos modos, no lograban entender completamente de qué se trataba al estar enfocados netamente en acabar satisfactoriamente el reto propuesto por el prototipo; es por esto que por ejemplo se evidencian múltiples respuestas (7 de 17) diciendo que logran percibir diferencias en el movimiento del enemigo en ambos modos cuando en realidad el enemigo estaba programado con exactamente los mismos comportamientos en ambas versiones, el único cambio era el método de renderizado, por otro lado, curiosamente estos usuarios no se percataron enteramente de la deformación en el modo vóxel.

En la entrevista realizada, el desarrollador experto Nicolás Nieto cuenta su punto de vista con respecto al uso de vóxeles en personajes de videojuegos comentando que estos podrían dar un dinamismo interesante al momento de interactuar con personajes enemigos, específicamente en la reacción frente a las acciones del jugador ya que estas se podrían ver mejor reflejadas en el juego y de esta forma enriquecerían la experiencia que tenga el usuario; por otro lado, considera que no es fácil implementar este tipo de métodos por cuestiones principalmente de rendimiento ya que a gran escala pueden llegar a consumir más recursos (hardware y tiempo de optimización e investigación) que con métodos tradicionales de mallas poligonales, y es por esto que en la mayoría de situaciones se opta por tener un buen rendimiento en los aspectos principales del juego por encima de otros factores que a priori puedan parecer más llamativos pero que a largo plazo se conviertan en un problema complejo de resolver.

4. CONCLUSIÓN Y TRABAJO FUTURO

En este trabajo se desarrolló un prototipo de videojuego en dos versiones, una con un personaje enemigo deformable implementado con vóxeles, y otra con el mismo personaje pero implementado con mallas poligonales convencionales. Los resultados muestran que el rendimiento obtenido en ambas versiones es suficiente para soportar una interacción fluida en el videojuego, además, las diferencias entre estas son prácticamente despreciables, siendo la más notoria la dependencia en la versión de vóxeles del uso de una GPU dedicada.

Las pruebas realizadas con usuarios muestran resultados prometedores. Los usuarios, aún sin un conocimiento previo acerca de los vóxeles, en su gran mayoría percibieron la versión de vóxeles como la más llamativa argumentando que el hecho de que el enemigo se deformara cuando recibía los disparos del jugador otorgaba un mayor grado de realismo y mejoraba la experiencia del juego. Estos resultados se complementaron con el punto de vista de un desarrollador de videojuegos con amplia experiencia en el sector, quien resaltó que las interacciones en un videojuego pueden enriquecerse al hacer que cada una de las acciones que se realicen dentro del entorno tengan una reacción tal que evidencie de forma clara que la interacción se llevó a cabo exitosamente, y además de eso mejore lo máximo posible la experiencia del jugador; es por esto que el desarrollador expresó su preferencia por la versión voxel sobre la malla poligonal convencional, ya que para él esta cumple con ese propósito en gran medida.

Sin embargo, aún cuando la interacción con el personaje basado en vóxeles demostró ser más interesante para los usuarios y la diferencia de rendimiento frente a la versión de mallas poligonales resultó mínima, se sabe que la implementación necesita mejorarse para poder ser usada en múltiples plataformas y desarrollos.

Nótese que el renderizado de la superficie de vóxeles tiene una apariencia “low-poly”. Una mejora del Surface Shader y el cálculo de normales por vértice podría mejorar la apariencia de la malla generada y así aumentar significativamente la experiencia del usuario. El método de Marching Cubes genera formas más suaves cuando los vóxeles son más pequeños, sin embargo, los voxels más pequeños exigirían más procesamiento de máquina para la deformación de la superficie en tiempo real.

Junto a esto, es pertinente investigar la razón que haya causado que la generación de vóxeles no se esté ejecutando correctamente en GPUs de marca AMD/ATI; debido a que todo el desarrollo se llevó a cabo utilizando una GPU NVIDIA no hubo oportunidad de probar en otra máquina hasta que se iniciaron las pruebas de usuario. Hay múltiples razones por las que podría estar ocurriendo el problema, entre esas la disposición y accesos de memoria, la precisión de los datos, la longitud de las Look-Up Tables, el número de hilos de la GPU utilizados para los cálculos, falta de actualización de los drivers o errores en la programación de los Shaders per se.

Sería interesante analizar el comportamiento de la implementación utilizando múltiples personajes basados en vóxeles interactuando al mismo tiempo, además de introducir un sistema de nivel de detalle y de generación y modificación a trozos (chunking) para disminuir la carga de procesamiento en el caso de que se quieran implementar geometrías con más de 20^3 vóxeles. Las simulaciones físicas también es un área con la que valdría la pena experimentar puesto que la deformación actual del modelo de vóxeles genera ciertos artefactos visuales, tales como partes de la malla que se desprenden y quedan flotando sin más en el aire por la falta de gravedad y comportamientos físicos en general del objeto, además que si se delegara el cálculo de colisiones con la superficie de vóxeles a la GPU, podría resultar en un método mucho más preciso y sería más fácil el desarrollar un sistema de simulación física básico con lo cual la interacción entre el jugador y el personaje basado en vóxeles podría enriquecerse aún más.

5. BIBLIOGRAFÍA

[1] J. Cleary, G. Wyvill. “Analysis of an algorithm for fast ray tracing using uniform space subdivision”. The Visual Computer Journal, [online], 1988. Available: https://link.springer.com/article/10.1007/BF01905559.

[2] D. Meagher. “Geometric modeling using octree encoding”. Rensselaer Polytechnic Institute, [online], 1982. Available: https://www.sciencedirect.com/science/article/abs/pii/0146664X82901046?via%3Dihub.

[3] J. Mahovsky, B. Wyvill. “Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plucker Coordinates”. University of Calgary, [online], 2004. Available: https://www.tandfonline.com/doi/abs/10.1080/10867651.2004.10487597.

[4] M. Sramek, A. Kaufman. “Fast ray-tracing of rectilinear volume data using distance transforms”. IEEE Transactions on Visualization and Computer Graphics, [online], 2000. Available: https://ieeexplore.ieee.org/document/879785.

[5] D. Cohen, Z. Sheffer. “Proximity clouds — an acceleration technique for 3D grid traversal”. The Visual Computer, [online], 1994. Available: https://link.springer.com/article/10.1007%2FBF01900697.

[6] D. Cohen, A. Kauffman. “Fundamentals of Surface Voxelization”. State University of New York at Stony Brook, [online], 1995. Available: https://www.sciencedirect.com/science/article/abs/pii/S1077316985710398?via%3Dihub.

[7] B. Reitinger, A. Bornik, R. Beichel. “Efficient volume measurement using voxelization”. the 19th Spring Conference on Computer Graphics, [online], 2003. Available: https://dl.acm.org/doi/abs/10.1145/984952.984962.

[8] M. Nießner, C. Siegl, H. Schäfer, C. Loop. “Real-time Collision Detection for Dynamic Hardware Tessellated Objects”. Eurographics 2013, [online], 2013. Available: http://graphics.stanford.edu/~niessner/niessner2013collision.html.

[9] V. Kämpe, E. Sintorn, E. Sintorn. “Fast, memory-efficient construction of voxelized shadows”. 19th Symposium on Interactive 3D Graphics and Games, [online], 2015. Available: https://dl.acm.org/doi/10.1145/2699276.2699284.

[10] C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann. “Interactive Indirect Illumination Using Voxel Cone Tracing”. Computer Graphics Forum,

[online], 2011. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2011.02063.x.

[11] K. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, G. Wiebecke. "Voxel-Based Visualization of Medical Images in Three Dimensions". Medical Images: Formation, Handling and Evaluation, [online], 1992. Available: https://link.springer.com/chapter/10.1007/978-3-642-77888-9_6.

[12] S. Ueng, L. Chen, S. Jen. "Voxel-based virtual manufacturing simulation for three-dimensional printing". National Taiwan Ocean University, [online], 2018. Available: https://journals.sagepub.com/doi/full/10.1177/1687814018781632.

[13] S. Ueng, L. Chen, S. Jen. "Phys-Sketch: Sketching 3D Dynamic Objects in Immersive Virtual Reality". National Taiwan Ocean University, [online], 2018. Available:http://www.inf.ufrgs.br/cg/PAPERS/3dmanipulation/Ticona2019_Chapter_Phys-SketchSketching3DDynamicO.pdf

[14] P. Schuytema. "Hex, Bugs & Rock 'n Rol: The Wizard of Voxels". Computer Gaming World, Issue 120, pp. 140-144, [online], 1994. Available: http://www.cgwmuseum.org/galleries/index.php?year=1994&pub=2&id=120

[15] I. McKendrick, Hello Games. "Continuous World Generation in No Man's Sky". Game Developers Conference, [online], 2017. Available: https://www.gdcvault.com/play/1024265/Continuous-World-Generation-in-No

[16] B. Francis. "What Astroneer's devs learned while leaving Early Access". Gamasutra, [online], 2019. Available: https://www.gamasutra.com/view/news/338417/What_Astroneers_devs_learned_while_leaving_Early_Access.php

[17] J. Henry. "Creative Voxel Game Teardown Gets New Gameplay Video". Gamerant, [online], 2020. Available: https://gamerant.com/teardown-gameplay-video/

[18] J. Yarwood. "How a new wave of developers are using voxels to create jaw-dropping worlds". Gamasutra, [online], 2020. Available: https://www.gamasutra.com/view/news/357225/How_a_new_wave_of_developers_are_using_voxels_to_create_jawdropping_worlds.php

[19] P. Schuytema. "Hex, Bugs & Rock 'n Roll: The Wizard of Voxels". Computer Gaming World Magazine, Issue 120, pp. 140-144. [online], 1994. Available: http://www.cgwmuseum.org/galleries/issues/cgw_120.pdf

[20] K. Silverman. “Voxlap Engine”. Ken Silverman’s Voxlap Page, [online], 2000. Available: http://advsys.net/ken/voxlap.htm

[21] A. Evans. “Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for ‘Dreams PS4’, a Geometrically Dense, Painterly UGC Game’”. Siggraph Advances in Real-Time Rendering in Games Course, [online], 2015. Available: http://advances.realtimerendering.com/s2015/index.html

[22] D. Bautembach. “Animated Sparse Voxel Octrees”. University of Hamburg, [online], 2011. Available: https://bautembach.de/

[23] A. Espe, Ø. Gjermundnes, S. Hendseth. “Efficient Animation of Sparse Voxel Octrees for Real-Time Ray Tracing”. Norwegian University of Science and Technology, [online], 2019. Available: https://arxiv.org/pdf/1911.06001.pdf

[24] R. Shrout. “John Carmack on Id Tech 6, Ray-Tracing, consoles, physics and more”. PCPerspective, [online], 2008. Available: https://pcper.com/2008/03/john-carmack-on-id-tech-6-ray-tracing-consoles-physics-and-more/

[25] C. Yerli. “The Future of Gaming Graphics”. Game Developers Conference Keynote, [online], 2008. Available: https://www.gdcvault.com/play/1012080/The-Future-of-Gaming

[26] H. Dingman. “I wish the PC had game creation tools as intuitive as PlayStation 4's Dreams”. PCWorld, [online], 2020. Available: https://www.pcworld.com/article/3540409/i-wish-the-pc-had-game-creation-tools-as-intuitive-as-playstation-4s-dreams.html

[27] D. Rossen. “Volumetric heat diffusion skinning”. Wolfire Blog, [online], 2009. Available: http://blog.wolfire.com/2009/11/volumetric-heat-diffusion-skinning/

[28] A. Rivers, D. James. FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation”. ACM Transactions on Graphics, [online], 2007. Available: http://www.alecrivers.com/fastlsm/

[29] Cocos Engine. “Building Collision Detection Using Signed Distance Field”. Cocos Creator Engine: Blog, [online], 2020. Available: https://www.cocos.com/en/building-collision-detection-using-signed-distance-field

[30] South East Games. “Paint the Town Red Dev Log 2 - Fun with Voxels and Dismemberment”. South East Games: Online Paint the Town Red Devlog, [online], 2015. Available:

http://www.southeastgames.com/blog/2015/5/17/paint-the-town-red-devblog-2-fun-with-voxels-and-dismemberment

[31] Lexaloffle Games. “Voxatron”. Lexaloffle Games: Voxatron Web Page, [online], 2013. Available: https://www.lexaloffle.com/voxatron.php

[32] D. Systèmes, “The Main Benefits and Disadvantages of Voxel Modeling”. Spatial Corp, [online], 2020.Available: http://blog.spatial.com/the-main-benefits-and-disadvantages-of-voxel-modeling

[33] A. Interguglielmi, “Clayxels”. [online], 2020. Available: https://assetstore.unity.com/packages/tools/game-toolkits/clayxels-165312

[34] M. Chou, “MudBun: Volumetric VFX Mesh Tool”. Long Bunny Labs, [online], 2020. Available: http://longbunnylabs.com/mudbun/

[35] J. Wahlberg, “Clayxels Shooter Test”. [Online], 2020. Available: https://www.youtube.com/watch?v=ZdV_5P6920Q

[36] V. Careil, “Voxel Plugin”. [Online], 2020. Available: https://voxelplugin.com/

[37] M. Larrabee, “VoxelGore”. [Online], 2020. Available: https://www.youtube.com/watch?v=JYabr4uHUGI

[38] A. Kaufman. “Introduction to Volume Graphics”. State University of New York at Stony Brook, [online], 2000. Available: https://www.semanticscholar.org/paper/Introduction-to-Volume-Graphics-Kaufman/a9f48466b9547c583feb454d4dc929518e55064d

[39] A. Kaufman, D. Cohen, B. Gurion. “Volume Graphics”. IEEE Computer, Vol. 26, No. 7, pp. 51-64, [online], 1993. Available: https://www.cs.tau.ac.il/~dcor/articles/older/volume.pdf

[40] M. Arvidsson. “Design and implementation of virtualized sparse voxel octree ray casting”. Lund University, pp. 21, 37. [online], 2010. Available: http://sam.cs.lth.se/ExjobGetFile?id=359

[41] W. Lorensen, H. Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. SIGGRAPH: Proceedings of the 14th annual conference on Computer graphics and interactive techniques. pp. 163–169, [online], 1987. Available: https://dl.acm.org/doi/10.1145/37401.37422

[42] M. Botsch, S. Steinberg, S. Bischoff, L. Kobbelt. “OpenMesh – a generic and efficient polygon mesh data structure”. RWTH Aachen University. pp. 1, [Online], 2002. Available: https://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf

[43] N. Esposito. “A Short and Simple Definition of What a Videogame Is”. University of Technology of Compiègne. pp. 2, [Online], 2005. Available: https://www.utc.fr/~nesposit/publications/esposito2005definition.pdf

[44] Irgamers. “La importancia de un personaje en videojuegos”. Irgamers Web Page, [online], 2018. Available: https://irgamers.cl/index/la-importancia-un-personaje-en-videojuegos-dos-centavos-t1-%E2%80%A2-e5/

[45] L. Perry. “The single most useful advice I can give for making any game better.. feedback”. Gamasutra Web Page, [Online], 2013. Available: https://www.gamasutra.com/blogs/LeePerry/20130506/191739/The_single_most_useful_advice_I_can_give_for_making_any_game_better_feedback.php

[46] N. Ruparelia. “Software development lifecycle models”. ACM SIGSOFT Software Engineering Notes, [Online], 2010. Available: https://www.researchgate.net/publication/220631422_Software_development_lifecycle_models

[47] Autodesk. “Maya ver. 2021” [Online], 2021. Available: https://www.autodesk.com/

[48] Blender Foundation. “Blender 3D”. [Online], 2021. Available: https://www.blender.org/

[49] Unity Technologies.”Unity”. [Online], 2021. Available: https://unity.com/es

[50] I. Quilez. “Distance Functions”. [Online], 2013. Available: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

[51] P. Borke. “Polygonising a scalar field”. [Online], 1994. Available: http://paulbourke.net/geometry/polygonise/

[52] Olena. “GPU vs CPU Computing: What to choose?”. Medium, [Online], 2018. Available: https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4

[53] J. Boni. “Gfx.WaitForPresent”. Unity Forum, [Online], 2013. Available: https://forum.unity.com/threads/gfx-waitforpresent.211166/

[54] Unity Technologies. “C# Job System”. [Online], 2018. Available: https://docs.unity3d.com/Manual/JobSystem.html

[55] J. Scrawk. “Mesh Voxelization”. Github Repository, [Online], 2020. Available: https://github.com/Scrawk/Mesh-Voxelization

[56] M. Pigram. “When would a mesh collider be better than primitive colliders”. GameDev StackExchange, [Online], 2015. Available: https://gamedev.stackexchange.com/questions/86579/when-would-a-mesh-collider-be-better-than-primitive-colliders

[57] Unity Technologies. “Cinemachine: Powering cameras for films and games”. [Online], 2020. Available: https://unity.com/unity/features/editor/art-and-design/cinemachine

[58] Unity Technologies. “Post-processing Stack”. [Online], 2020. Available: https://docs.unity3d.com/2018.3/Documentation/Manual/PostProcessing-Stack.html

[59] D. Erdmann. “Fast IK”. [Online], 2019. Available: https://assetstore.unity.com/packages/tools/animation/fast-ik-139972

[60] A. Aristidou. “FABRIK: a fast, iterative solver for the inverse kinematics problem”. University of Cambridge [Online], 2011. Available: http://andreasaristidou.com/publications/papers/FABRIK.pdf

[61] Computer Hope. “Inverse kinematics”. [Online], 2019. Available: https://www.computerhope.com/jargon/i/inverse-kinematics.htm

6. ANEXOS

Anexo 01. Video Gameplay:
https://drive.google.com/drive/folders/1X1Gsquy71fddOSjpL2Bo2b2sIt8kIzH3?usp=sharing

Anexo 02. Ejecutable:
https://jamesr48.itch.io/spoxel

Anexo 03. Código Fuente:
https://github.com/JamesR48/Deformable-Boss/tree/components-code/Clean/DeformableBoss

Anexo 04. Formulario entrevista con usuarios:
https://forms.gle/TugPxhrfkJ6haLp47

Anexo 05. Gráficos de rendimiento de aplicaciones
https://docs.google.com/document/d/1rjruhmqrX9GW-igaBfid5iTPU5pKFYHXWAfv5WBgc2g/edit?usp=sharing

Anexo 06. Gráficos de rendimiento de ejecutable final
https://docs.google.com/document/d/1achIiT6HzVytXd3Z89ywxpGl-brnds_S1oA4EgB6jSQ/edit?usp=sharing

Anexo 07. Entrevista con Nicolas Nieto:
https://drive.google.com/drive/folders/1X1Gsquy71fddOSjpL2Bo2b2sIt8kIzH3?usp=sharing

Anexo 08. Respuestas abiertas:
https://docs.google.com/spreadsheets/d/1ESI7CfALXZpSZYlamkgq6vcKVMcGc_kvWpXaIRfdPmI/edit?usp=sharing