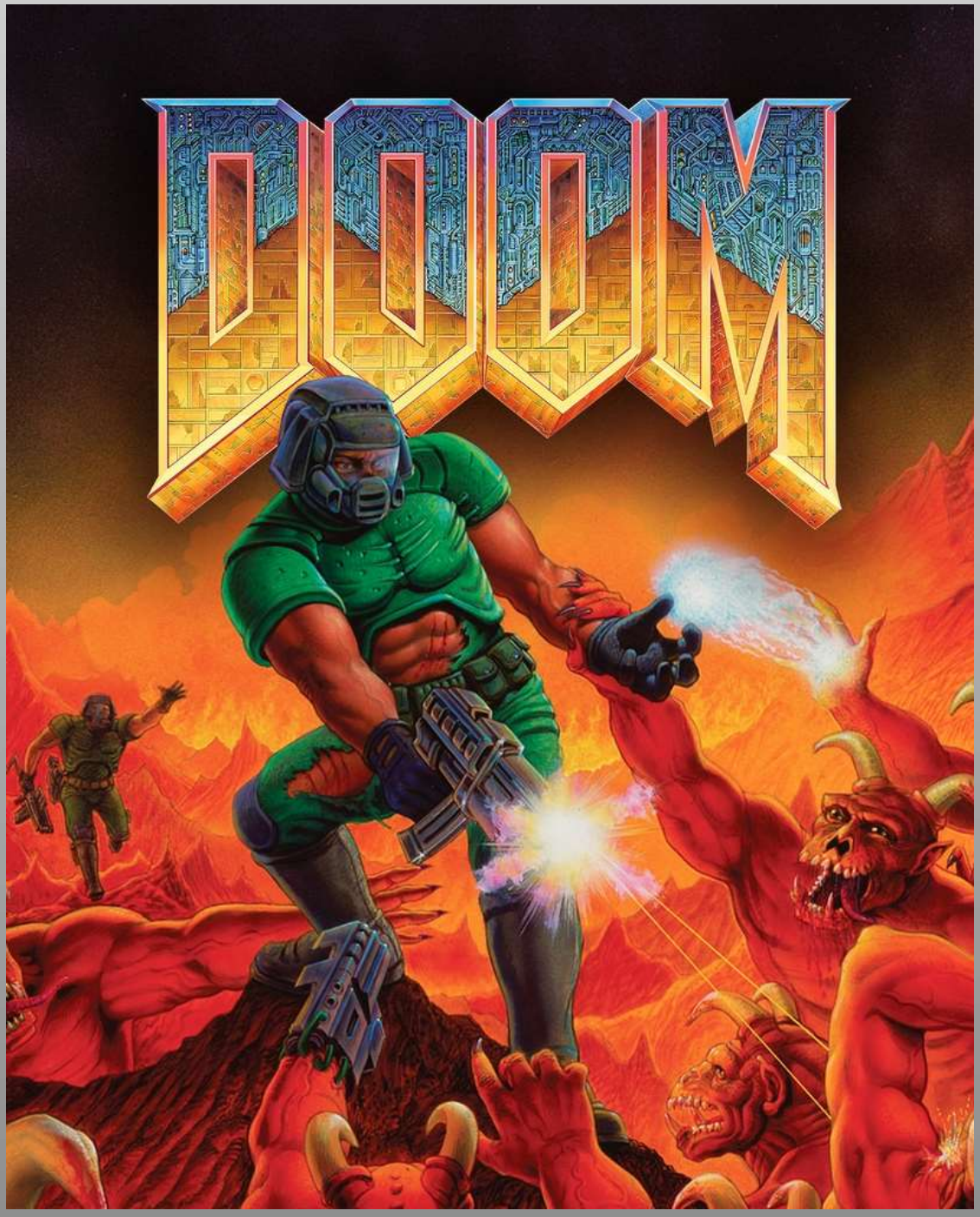


Grupo 4

Víctor Martínez Lascorz - 738845
Pedro José Pérez García – 75664

Zaragoza, 25 de mayo de 2021

Contenidos

1. Introducción 3
2. Detalles y retos de la Implementación 3
2.1. WADs, motor y editar los niveles 4
3. Gráficos y renderizado 5
3.1 Estructura de un mapa 5
3.2 Árboles BSP 5
3.3 Renderizado de paredes 6
3.3.1 Paredes normales 6
3.3.2 Portals: Ventanas y escaleras 8
3.4 Renderizado de suelos y techos 8
3.4.1 Glitches menores en nuestro clon 9
3.5 Mapeado de texturas 9
3.5.1 Paletas de colores 10
3.5.2 Mapeado de texturas en paredes al estilo doom 10
3.5.3 Texturas en suelos y techos 11
3.6 Things y aspectos descartados 12
3.7 Renderizado de enemigos 13
3.7.1 Transición entre sprites 13
3.7.2 RenderEnemy 13
3.8 Resolución y framerate 15
3.8.1 Resolución interna 15
3.8.2 Framerate interno y bucle principal 16
4. Sonido 16
5. Jugabilidad 17
5.1 Lógica de juego 17
5.2 Dificultad 18
5.3 Diagramas de navegación 18
6. Jugador 19
6.1 Movimiento del jugador 19
6.1.1 Desplazamiento frontal 20
6.1.2 Rotación a izquierda y derecha 20
6.1.3 Actualización de la posición vertical del jugador 21
6.2 Movimiento de la cabeza y el arma 21
6.3 Colisiones 22
6.3.3 Limitaciones de nuestras colisiones y puertas 23
6.4 Disparo del Jugador 24
7. Enemigos: Inteligencia Artificial 25
7.1 Salud de los enemigos 25
7.2 Tipos de enemigos: IA 26
8. Conclusiones 27
9. Referencias y bibliografía 29

Anexo I: Doom es 3D y no 2.5D 30
Anexo II: Controles del juego 30
Anexo III: GDD entregado en marzo 31

Nota: Los títulos de las secciones de la memoria no llevan tilde más adelante por problemas con la fuente utilizada



1. Introduccion

DOOM es un videojuego desarrollado originalmente en 1993 por id Software para PC, en concreto para el sistema operativo DOS. El juego se enmarca en el género *first person shooter*, juego de disparos en primera persona. En él, se introduce al jugador como un marine que ha llegado a la base científica de una luna de marte, Phobos, a reconocer el terreno después de que se perdiera contacto con el personal allí presente durante la realización de un experimento. Todos los compañeros del jugador han sido emboscados y devorados, o peor aún, convertidos en las mismas cosas que los han matado, y es que la base está llena de todo tipo de demonios y criaturas infernales y no hay otra salida que atravesarla.

El jugador controla al único superviviente de toda esa masacre y tiene que abrirse paso entre hordas de enemigos infernales a través de numerosos niveles distribuidos a lo largo de tres actos: *Knee Deep In The Dead*, *Shores Of Hell* e *Inferno*. Durante estos niveles el jugador puede recoger armas de todo tipo que le permitirán tener un arsenal de muerte y destrucción a su disposición, así como botiquines, packs de armadura y otros objetos que le podrán ser útiles.

La estructura de los niveles suele ser laberíntica, con emboscadas inesperadas e injustas con el jugador, y una estructura parecida a la de un puzle, basada en buscar las llaves de colores que permiten acceder a las demás zonas del nivel y finalmente, encontrar la salida.

El juego se distribuyó en formato físico, y también en su versión *shareware*, donde los 9 niveles del primer acto son completamente gratuitos. Esto le permitió alcanzar una gran popularidad y revolucionar el género, convirtiéndolo al instante en un juego de culto, y con el tiempo, en un clásico. Después de todo, ¿a quién no le gusta bajar al mismísimo infierno con una escopeta de doble barril y nada que perder?

2. Detalles y retos de la Implementacion

Para esta asignatura de Videojuegos se ha elegido realizar un clon de DOOM, utilizando el lenguaje C++ junto con la librería multimedia SFML, que facilita la gestión de ventanas, audio y dibujado de primitivas, así como de texturas o sprites en la pantalla.

El mayor reto de realizar este clon de DOOM que lo diferencia del resto de juegos que se presentan es el renderizado de los escenarios, algo que el equipo ya se temía desde el principio ya que el juego presenta una apariencia tridimensional, y aunque se especule mucho sobre si DOOM es un raycaster o no, o si es 2.5D o 3D, podemos aclarar que se trata de un videojuego en 3D (ver Anexo I) que gestiona la altura por separado a las otras dos dimensiones y no es un raycaster, sino que se podría decir que es un BSP *engine*. El funcionamiento detallado de estos algoritmos se explicará más adelante.

2.1. WADs, motor y editar los niveles

Ante el reto de recrear un videojuego 3D sin ningún tipo de motor gráfico se tuvo que decidir rápidamente para no perder tiempo cómo se iba a hacer, y tras consultarlo con los profesores se determinó que la mejor forma era intentar recrear el funcionamiento del id Tech 1, el motor gráfico que movía el juego original.

Tras esto, se trató de reunir información acerca de su funcionamiento, y se descubrió que en el juego original solamente había dos archivos, DOOM.exe y DOOM.WAD. El primero es el ejecutable del juego y el segundo es un archivo con formato propio, que aglomeraba todos los datos relevantes de niveles, enemigos, gráficos y sonido. Se basa en una serie de "directorios", que son el nombre del contenido, su tamaño en bytes y un offset a los datos que están en otra dirección dentro del propio archivo.

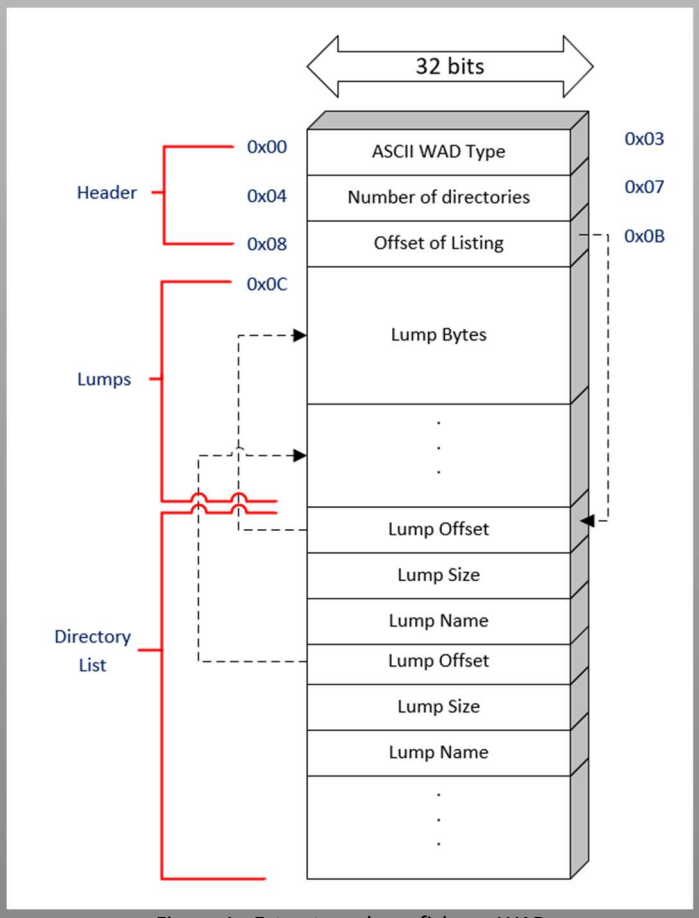


Figura 1 - Estructura de un fichero .WAD

Por tanto, se iba a intentar leer la mayor cantidad de recursos posibles directamente del .WAD original de la versión de shareware (gratuita y libre) a la hora de hacer el juego, lo que eliminaría la necesidad de, por ejemplo, crear niveles.

Esta decisión a la vez permite que los jugadores editen los mapas, texturas o número y posicionamiento de enemigos con las numerosas herramientas que hay disponibles en internet, y así el juego presente una apariencia muy diferente a la creada por nosotros.

Se han programado también métodos para leer diferentes tipos de datos del WAD, que no se detallarán, aunque sí se explicará la estructura de estos tipos de dato.



3. Graficos y renderizado

Este apartado es el que más tiempo de desarrollo ha necesitado, ya que antes de programar el juego, se necesitaba el motor que lo moviera, pero con el mismo tiempo que el resto de grupos, así que nos vamos a detener en cada etapa del renderizado de un *frame* para explicar los aspectos más relevantes, para poner en valor el esfuerzo realizado más allá de conseguir una jugabilidad similar al original.

En primer lugar, es necesario conocer un poco de cómo DOOM almacena los mapas, para entender por qué se utiliza el algoritmo que se utiliza, y a qué nos referimos con lo de que gestiona una dimensión aparte de las otras dos.

3.1 Estructura de un mapa

Hay varios tipos de datos almacenados en un .wad que permiten formar un mapa. Estos son:

- VERTEX: Un vértice en 2D, con coordenadas X e Y.
- LINEDEF: Un muro, dado por dos vértices. Incluye flags para gestionar sus funciones.
- SIDEDEF: Cada muro tiene dos lados, con información de sus texturas, entre otras cosas.
- SECTOR: Una habitación del mapa, es un conjunto de LINEDEFS. Contiene la información acerca de la altura del suelo y el techo común a todas esas paredes, el nivel de iluminación, etc.
- SEGS: Segmentos de un LINEDEF. A la hora de renderizar se utilizan estos para cálculos.
- NODES: Nodos de un árbol BSP, vital para renderizar. Se explica a continuación.

Hay más tipos (THINGS, BLOCKMAP) pero los más importantes son los aquí listados.

3.2 Arboles BSP

Como se trata de un juego de 1993 donde la optimización era clave, no se utilizaban algoritmos como el del pintor, ya que se estarían dibujando cosas innecesarias sobre las que luego se tendría que volver a dibujar.

En su lugar se utilizaba un árbol de particiones binarias del espacio (BSP), que viene ya preparado en el .wad y solo se ha tenido que leer. Este árbol contiene subdivisiones del mapa hasta pequeños sectores, que, como restricción añadida deben ser convexos por el funcionamiento del algoritmo. Por esto, los nodos hoja del BSP no tienen por qué ser los sectores del wad, son otro dato parecido llamado SSECTOR, que contiene los SEGS.

Para renderizar desde la posición del jugador (de más cercano a más lejos), se ha implementado de la siguiente manera aprovechando la recursividad:

```
void Renderer::RenderBSPNodes(int16_t nodeID, int i)
{
    //Comprobar con la máscara si es un nodo hoja
    if ((int16_t)(nodeID & SUBSECTORIDENTIFIER))
    {
        RenderSubsector((int16_t)(nodeID & (~SUBSECTORIDENTIFIER)));
        return;
    }

    if (IsPointOnLeftSide(m_pPlayer->GetXPos(), m_pPlayer->GetYPos(), nodeID))
    {
        RenderBSPNodes(m_pMap->getNode(nodeID).LeftChild, i+1);
        RenderBSPNodes(m_pMap->getNode(nodeID).RightChild, i + 1);
        //izquierda y luego derecha, de cerca a lejos
    }
    else
    {
        RenderBSPNodes(m_pMap->getNode(nodeID).RightChild, i + 1);
        RenderBSPNodes(m_pMap->getNode(nodeID).LeftChild, i + 1);
        //derecha y luego izquierda, de cerca a lejos
    }
}
```

Figura 2 - Recorrido del árbol BSP al renderizar subsectores



3.3 Renderizado de paredes

En el juego original y por tanto en nuestra implementación, se diferencia entre las paredes normales y las de las ventanas (portals), que son más complicadas y requieren más trabajo debido a su visibilidad parcial con lo que exista detrás de la misma. Por ahora se detalla el proceso seguido para renderizar las paredes comunes.

3.3.1 Paredes normales

El primer paso es determinar qué paredes está viendo el jugador y cuáles no. Se tienen las posiciones del jugador y de los vértices de la pared, así como el campo visual del jugador y el ángulo hacia dónde está mirando, por lo que se descartan las paredes que no se encuentren dentro del campo visual del jugador, haciendo también *culling* de las que se encuentran detrás suyo. Junto a este paso, se ajustan las paredes que están parcialmente en el campo visual para quedarse únicamente con la parte que sí lo está.

Este paso se hace utilizando el ángulo que forman hacia el jugador, en vez de las coordenadas x e y del segmento.

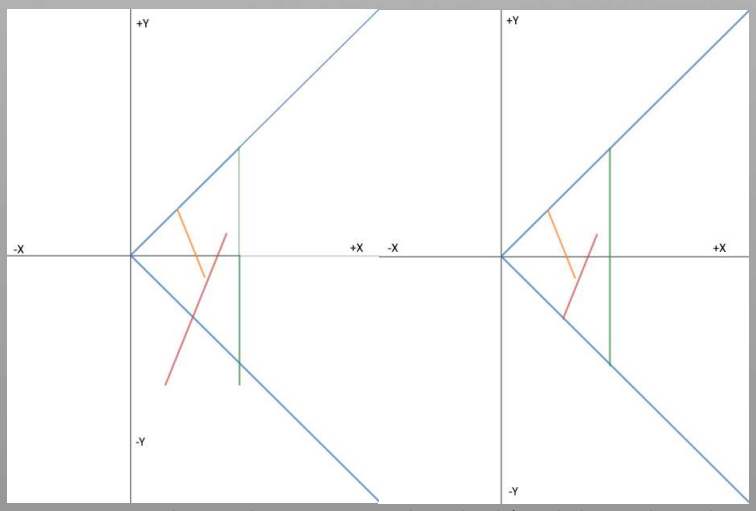
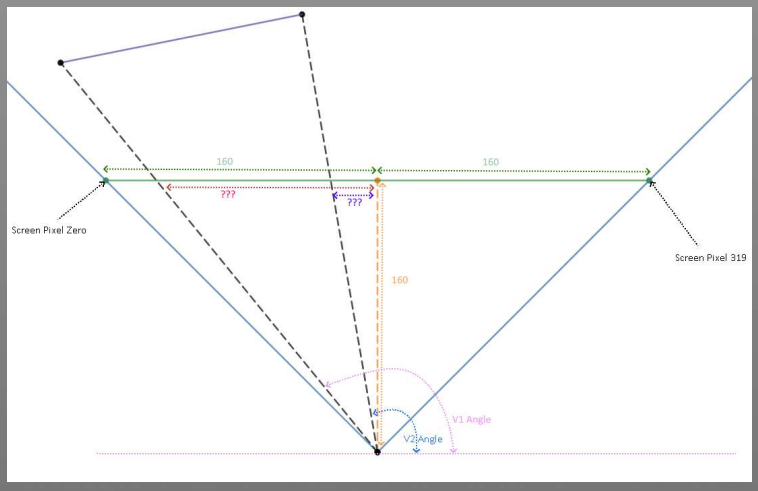


Figura 3 - Clipping de segmentos utilizando el ángulo hacia el jugador

A continuación, se proyecta al espacio de pantalla horizontalmente, ya que solo se calcula la columna en la que iría el primer vértice. Para ello, como se conoce que el campo visual es de 90 grados y que la pantalla tiene una anchura fija (320 píxeles de ancho en el juego original), se puede utilizar la tangente para ello:



$$px\ si\ \theta > 90 = 160 - \tan\theta * 160$$
$$px\ si\ \theta < 90 = \tan(90 - \theta) * 160 + 160$$

Figura 4- Cálculo de la proyección horizontal de una pared.

Antes de realizar la proyección vertical, se calculan las intersecciones de todas las paredes, de forma que si una tapa total o parcialmente a otra, no se tenga que computar ese segmento de la pared que va a quedar por detrás. Para ello se mantiene una lista durante todo el tiempo de renderización del frame, y se van comprobando todos los rangos ocupados con el inicio y el final del muro que se vaya a añadir en ese momento. A diferencia del juego original, en nuestra implementación no se utiliza esta lista para nada más, pero el juego original la utiliza para otros cálculos, como comprobar que todo el rango [0-319] de la pantalla ha quedado cubierto.

Para la proyección vertical de las paredes se ha procedido una vez más como lo hacía el motor gráfico del juego original. En este caso, lo que se hace es encontrar un factor de escalado. Para ello se tiene en cuenta que en el plano de proyección del jugador (situado a 160 unidades de distancia de él si la resolución y el campo visual son los valores estándar) una unidad de altura equivaldría a un píxel de altura, es decir, no hay escalado. Así, el cálculo se reduce a encontrar la distancia hasta el plano de visión (la pantalla) conociendo el ángulo entre el jugador y el vértice de la pared, y encontrar la distancia entre el jugador y el vértice, y dividir uno por otro para encontrar el ratio de escalado que deseamos.

Por tanto, si $\boldsymbol{\theta}$ es el ángulo entre el frontal del campo visual del jugador y el vértice para el que se quiere encontrar el factor de escalado, necesitamos encontrar una línea perpendicular a la pared y calcular su distancia hasta el jugador, así como el ángulo entre esta normal y la línea desde el jugador al vértice, lo llamamos α:

$$escalado = \frac{\frac{160}{\cos\theta}}{\frac{distNormal}{\cos\alpha}}$$

Figura 5 - Cálculo del factor de escalado de un vértice

Se calcula este factor para los dos vértices de la pared, y cuando se tiene, se calcula el valor diferencial para interpolar entre ambas columnas de pantalla X1 y X2 de forma iterativa:

$$\Delta escalado\ = \frac{escalado(X2) - escalado(X1)}{X2\ - X1}$$

Figura 6 - Cálculo de la variación del factor de escalado para interpolar

Y el escalado para una columna X_i cualquiera entre X1 y X2 queda:

$$escalado(X_i) = escalado(X1) + \Delta escalado * (i - X1)$$

Al implementarlo de manera iterativa, se puede hacer que se dibujen de izquierda a derecha varias líneas verticales que tengan en cuenta la altura del suelo y del techo de ese sector a renderizar, siendo una primera aproximación al renderizado de paredes.



3.3.2 Portals: Ventanas y escaleras

Ahora que ya se ha explicado cómo se renderizan las paredes normales, se va a explicar cómo se ha conseguido renderizar el otro gran tipo de paredes, los portales. Estos son un tipo especial de paredes que tienen una apertura a través de la que se puede pasar y ver. Por tanto, tienen una zona central abierta, lo que implica que hay dos "partes" de pared, desde el techo hasta la apertura y desde el final de la apertura hasta el suelo.



Figura 7 - Ejemplo de ventana y escaleras en nuestro clon

Además, con este tipo de geometría se pueden construir escaleras como las que se ven en la imagen de arriba, dotando al juego de mayor sensación de tridimensionalidad (aunque no permite tener al jugador en dos alturas diferentes para unas mismas coordenadas X e Y, es decir, no hay "pisos" ni nada parecido en los niveles).

Para añadir soporte a este tipo de geometrías y que se vea lo que hay detrás teniendo en cuenta que estamos dibujando de cerca a lejos, hay que llevar la cuenta de qué rango de píxeles siguen libres para dibujar en ellos. Esto se logra manteniendo dos listas en el tiempo de vida de cada frame, que para cada X en todo lo ancho de la pantalla nos permiten definir la Y más cercana al techo a partir de la cual se puede dibujar, y la Y más cercana al suelo a partir de la que ya se ha pintado algo. Por tanto, consultando los valores de ambas listas se puede definir un rango en el que se puede renderizar, y si los valores de las listas se cruzan o son lo mismo, es que ya se ha renderizado algo a lo alto de toda esa columna en concreto.

3.4 Renderizado de suelos y techos

En el motor original, se definían unas estructuras de datos llamadas "visplanes", que definían un rango para los suelos de forma parecida a las listas de oclusión para el suelo y el techo, donde se encolaban los elementos para ser renderizados más adelante. De esta forma, cada pared tiene su fragmento de techo y de suelo que renderizar, y para optimizar se pueden fusionar los fragmentos coincidentes en coordenadas que compartan textura.



Figura 8- Escena del juego original, con 179 fragmentos de techo o suelo, que al optimizarse pasan a 28.



No obstante, en nuestra implementación no se han creado estas estructuras porque debido a la potencia computacional de la que disponemos, podemos permitirnos renderizar cada fragmento de suelo por separado, aunque pueda haber 6 veces más por frame, además de que el número de columnas de pantalla que hay que implementar es invariante, aunque haya menos fragmentos que las agrupen.

3.4.1 Glitches menores en nuestro clon

En nuestra versión se renderizan los techos y suelos a la vez que las paredes, en vez de renderizar primero las paredes y guardar el resto para más adelante. Esto puede causar algunos problemas debido al funcionamiento de las listas que gestionan qué el clipping de coordenadas del techo y el suelo. En concreto, si el jugador mira hacia adelante y tiene una ventana (o la sección inferior de un portal) por encima de su cabeza, a veces el renderizador deja esa área sin pintar:

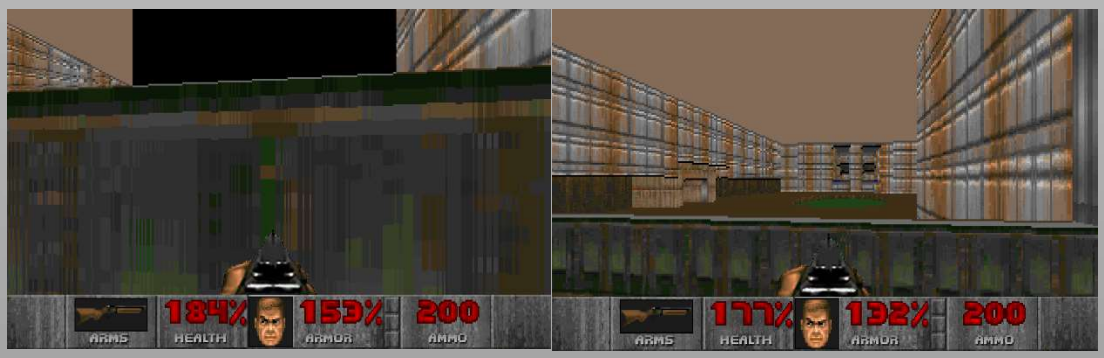


Figura 9- Glitch gráfico fruto de la implementación del renderizado de techos y suelos.

3.5 Mapeado de texturas

La mayor parte del tiempo de desarrollo se invirtió en obtener un renderizador funcional, por lo que se utilizaban colores aleatorios para pintar cada sección de pared, suelo o techo, basándose en un diccionario que como clave utilizaba el nombre de la textura que debería ir allí. De hecho, en la versión de prueba no se tenían texturas funcionales todavía debido a la complejidad de lograr un método convincente para mapearlas, pero finalmente se logró una implementación que, aunque no es perfecta, se asemeja bastante al acabado que presenta el juego original.



Figura 10 - Comparativa entre nivel sin texturas y con texturas. Aparentemente texturizar te activa Windows.



3.5.1 Paletas de colores

No se va a entrar en muchos detalles acerca del funcionamiento de las texturas y de cómo leerlas y utilizarlas, puesto que ya está suficientemente documentado en otras fuentes, pero sí que hace falta decir que las cargamos del .wad original y que cada texel es en realidad un número que indexa uno de los 256 colores de la paleta de colores que estemos utilizando en ese momento. El juego original tiene varias paletas de colores para varias situaciones, como, por ejemplo, al recibir daño cambia a una paleta con tonos rojos para representar la sangre en la visión del jugador al recibir daño.

En nuestro caso, cargamos las paletas del .wad pero para mantener la implementación simple y realizable en el plazo de tiempo disponible, hemos optado por cargar únicamente la paleta por defecto que se usa durante la mayoría del tiempo de juego en una partida, llamada PLAYPAL.

3.5.2 Mapeado de texturas en paredes al estilo DOOM

Uno de los grandes problemas del mapeado de texturas de la época era lo caro que era computacionalmente el interpolar correctamente las texturas, sin que se produjera una deformación causada por no tener en cuenta la profundidad al interpolar u. Sin embargo, en DOOM solucionaron ese problema aprovechándose de que se podía usar una interpolación basada en u/z (siendo z la profundidad) en lugar de u. Esto se debe a que siempre se cumple que todos los muros del juego son perpendiculares al suelo y no hay ningún tipo de planos inclinados, una restricción que permite realizar esta interpolación sin un coste exagerado (para su época).

$$u_{\propto} = \frac{(1-\propto)\frac{u_0}{z_0} + \propto \frac{u_1}{z_1}}{(1-\propto)\frac{1}{z_0} + \propto \frac{1}{z_1}}, \quad 0 \leq \propto \leq 1$$

Figura 11 - Interpolación lineal de la coordenada u estilo DOOM

En nuestro clon se ha intentado reproducir esta interpolación lineal, de forma que la distancia desde la columna X de pantalla que se debe renderizar en ese momento, proyectada hasta la pared en coordenadas del mundo, y el primer vértice de la pared se ve dividida por la longitud total de esa pared para sacar un valor entre 0 y 1 que utilizamos como $\propto$ en la interpolación.

Como z_0 y z_1 se utilizan las distancias hasta los dos vértices de la pared, y como u_0 y u_1 se tienen en cuenta 0 y la anchura de la pared, dado que luego al indexar el texel correspondiente se hace un and bit a bit con la anchura de la textura, debido al funcionamiento interno del original.

La coordenada v se calcula con una interpolación lineal simple, sin dividir por la z interpolada.



Figura 12 - Ejemplo de mapeado de texturas en paredes



Esta forma de calcular el texel de la textura que se debe utilizar en cada momento se asemeja lo suficiente al juego original, aunque no es perfecta y presenta algunos pequeños defectos al observar desde ángulos casi paralelos las paredes normales. Esto se puede deber a otras variables de los Linedefs, Sidedefs y demás estructuras de Doom que pueden modificar cosas del mapeado de texturas (flag DONTPEGBOTTOM, por ejemplo, permite ajustar una textura respecto al techo o al suelo).

Además, en las secciones inferiores de algunos Portals las texturas se deforman bastante al verlas de cerca. Somos conscientes de este pequeño error en el mapeado de texturas, pero se ha decidido conservar de cara a la versión final al observar que la mayoría de paredes falsas del juego, que son atravesables o son puertas ocultas, presentan esta ligera deformidad en las texturas al acercarse, y puede tratarse de una buena forma de alertar al jugador de ello.



Figura 13 - Textura de una pared falsa en algún lugar del nivel E1M2.

3.5.3 Texturas en suelos y techos

Para programar el mapeado de texturas en suelos y techos no se disponía de demasiado tiempo, ya que su mapeado funciona diferente a las texturas, y se almacenan en el .wad de otra forma diferente a las texturas convencionales normales (se renderizan en horizontal y no en vertical, y se almacenan rotadas 90 grados para aprovechar el precacheamiento de los ordenadores de su época), por lo que habría que programar también el cómo leerlas de allí.

Además, invertir ese en intentar obtener un resultado decente habría sido contraproducente porque no se podría haber utilizado en hacer numerosas pequeñas mejoras en el resto de áreas del juego, afectando más negativamente que positivamente al resultado final.



Figura 14- Techo y suelos con colores aleatorios producen un mal contraste



Aun así, se consideraba que pintar colores aleatorios según la textura que debería ir ahí producía resultados poco agradables al jugador y que se debía hacer algo al respecto. De esta forma, se buscó un compromiso entre tiempo invertido y resultado final que resultó ser bastante eficaz: Con un editor de wads se fueron comprobando todas las texturas de suelo y techo una por una, y se buscaba el color de la paleta de colores más común y representativo de cada textura, y se almacena en un diccionario al iniciar el renderizador para que al pintar los frames se pudiera acceder directamente en base al nombre de la textura deseada.

El resultado, como se puede ver, mejora sustancialmente y se asemeja a la versión original:

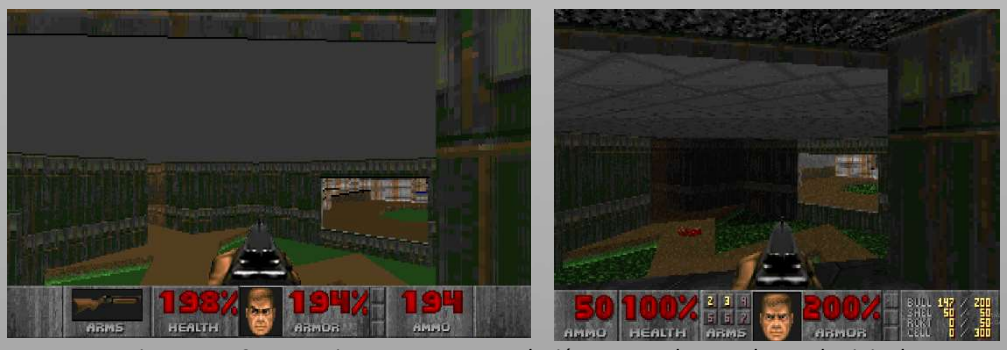


Figura 15 - Comparativa entre nuestra solución para suelos y techos y el original.

3.6 Things y aspectos descartados

Things es el nombre por el que se conoce a diferentes elementos del escenario, en su mayoría decorativos, como un cadáver en el suelo, un árbol, un candelabro, pero también pueden ser interactuables, como los barriles explosivos o los botiquines.

Por desgracia, en comparación con el original en este apartado no hay mucho que contar, ya que no hemos implementado su renderizado debido a su coste y el tiempo del que se disponía para todas las demás tareas. Esto quiere decir que no hay botiquines ni objetos recogibles en el escenario, se detallará el cómo se ha suplido su ausencia en la sección dedicada a la jugabilidad.

Además, tampoco se ha renderizado la imagen de fondo que simula el cielo y el paisaje lejano que se puede ver en algunos puntos de E1M1 o E1M2, por las mismas razones, aunque sí que está cargada en memoria al estar considerada una textura normal en la estructura del wad.

En la misma línea, tampoco se ha podido hacer que los colores utilizados se oscurezcan más conforme se renderiza lejos de la cámara para simular iluminación, ni se han utilizado los mapas de iluminación que se indexan con datos de los sectores para dar iluminación a zonas concretas del mapa. Esto se puede observar en la figura de arriba, o más claramente en el segundo nivel, E1M2 donde en algunas zonas el juego original recrea una oscuridad casi total mientras que en nuestro clon se ve iluminado.

El único apartado de things que se utiliza actualmente es el que representa a los enemigos. De ahí se saca la lista de todas las entidades, con su información respecto al punto de inicio x e y de cada uno. Estos son guardados en un vector para poder actualizarlos y renderizarlos según se necesite en el flujo de juego.



3.7 Renderizado de enemigos

El renderizado de enemigos empieza una vez se haya generado y texturizado todo el ambiente, es decir, paredes techo y suelo. Su renderizado consiste en recorrerse la lista de enemigos del vector y comprobar si son visibles para el jugador. Esto es, comprobar un booleano que se actualiza en la fase de update a true si el jugador puede ver a la entidad, es decir, no hay paredes solidas entre el enemigo y el jugador. En el caso de que sea visible y esté dentro del campo de visión, se ejecuta su función renderEnemy.

3.7.1 Transicion entre sprites

Las animaciones de movimiento y muerte de los enemigos constan de 4 Sprites cada una. Para generar un movimiento decente en ellos hay que ir alternando dichos Sprites a un ritmo considerable. En nuestro caso esto se ha logrado generando un thread independiente para cada enemigo que va aumentando un contador n modulo 4 cada 100 ms hasta que dicha entidad muere, con lo cual reproduce los últimos 4 cambios en bucle entero representando la animación de muerte y luego deja un -1 antes de terminar indicando que ese enemigo ya no necesita dicho contador porque ha terminado su animación de muerte y el único Sprite que le queda por mostrar es el final.

Hay algunas animaciones puntuales como la de recibir daño que se corresponde a un único Sprite que se renderiza antes de seguir de nuevo con el correspondiente n de la animación actual.

3.7.2 RenderEnemy

La función renderEnemy se encarga de elegir el Sprite adecuado, escalarlo según la distancia, posicionarlo en el eje X / Y correspondiente y para finalizar, dibujarlo en pantalla.

Para elegir el Sprite correspondiente a la acción del enemigo se usa un árbol de decisiones. Dicho árbol analiza el estado del enemigo, el jugador y el valor n para la animación si se necesitase. Lo primero que hay que analizar es si sigue vivo el enemigo, puesto que, en caso contrario, bastará con mostrar la animación de muerte pasando por los cuatro sprites marcados por el valor n. En caso de que el valor n sea igual a -1 significa que su animación ya ha pasado, con lo cual el Sprite a elegir será el último de muerte, correspondiente al cuerpo en el suelo.

El siguiente apartado que tiene prioridad es el de disparo, debido a que es el que más interesa al jugador, pudiendo plantear una estrategia con respecto al mismo. Si está disparando mostrará dichos sprites mirando al jugador, en caso contrario significa que simplemente se está moviendo.

La animación de movimiento (o de herido) depende del ángulo que se obtenga entre el ángulo al que mira al jugador y el ángulo al que mira el enemigo. Si este valor es de 180 grados (+- 45 para representar 90 grados en total) quiere decir que se están mirando de frente. Si el valor es de menor de 135 está girado hacia la izquierda y si el valor mayor que 225 estará mirando hacia la derecha. Una vez conocido el lugar hacia donde mira con respecto al jugador, solo queda saber si hay que dibujar el Sprite herido o el de simple movimiento dependiendo si el jugador le ha disparado recientemente.



Este árbol de decisiones se puede resumir en el siguiente diagrama:

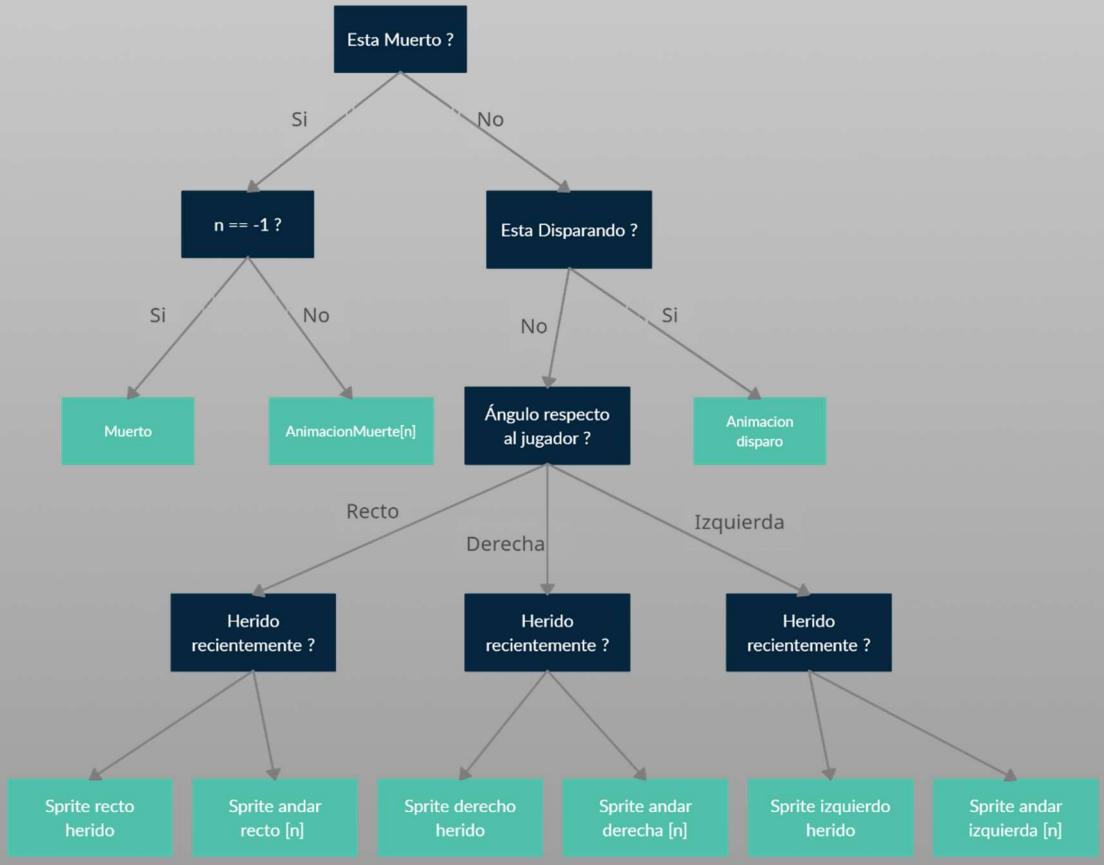


Figura 16 - Cómo determinamos el sprite de enemigos a renderizar.

En este momento ya se conoce el Sprite a mostrar, con lo cual solo queda escalarlo y posicionarlo.

Para el escalado, están calculados diversos intervalos lineales, los cuales crecen más despacio cuanta más distancia haya, y creciendo exponencialmente conforme se acerca, simulando el mismo efecto que ocurre en la vida real.

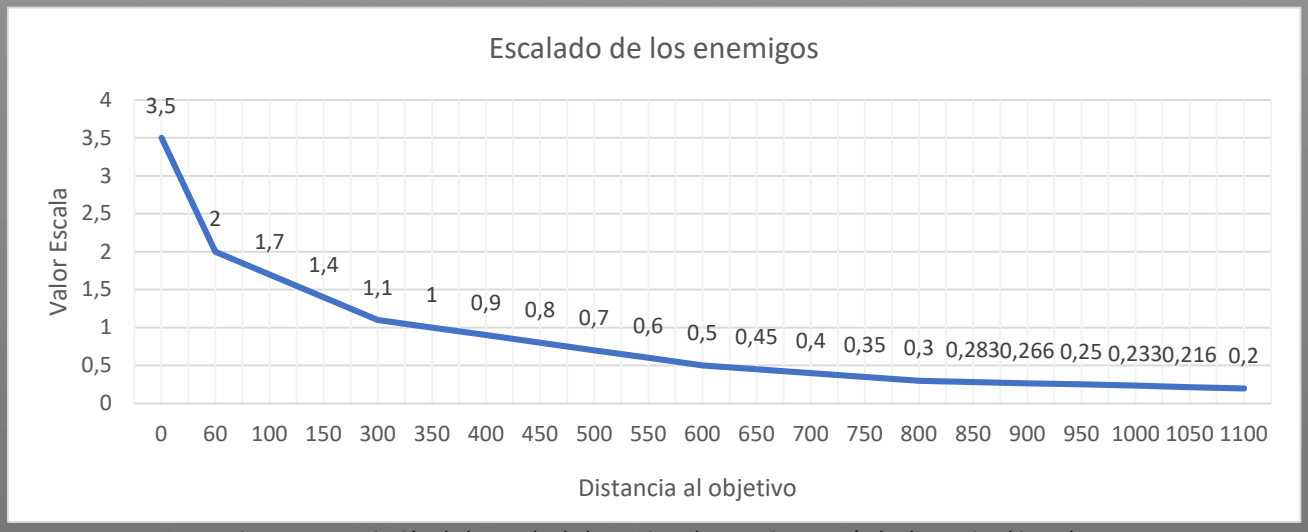


Figura 17 - Variación de la escala de los sprites de enemigos según la distancia al jugador.



Este valor del escalado está acotado inferiormente para que al menos sea visible para el usuario, con lo cual 0.2 es su valor mínimo. La clase Sprite de SFML cuenta con su propia función scale la cual aumenta el tamaño multiplicándolo por un valor n (o disminuye si ese valor es negativo). Dicho esto, escalarlo por un valor 0,5 a 600 unidades de distancia corresponde a disminuir a la mitad el tamaño, y ampliarlo a 2 a 60 unidades significa duplicar su tamaño.

El último paso es conocer la ubicación del Sprite escalado. Para conocer la ubicación del eje X se comprueba con qué ángulo lo está viendo el jugador. Si este lo está viendo a 10 grados y se conoce que doomguy tiene un field of vision de 90 grados, estará a 1/9 de la pantalla. A partir de esta simple regla de tres, se puede conocer el eje X. Hay que tener en cuenta que esta posición es el centro del enemigo, con lo cual hay que restarle la mitad del tamaño del Sprite por el escalado que se haya calculado previamente.

Para conocer el eje Y del enemigo hay que tener en consideración la diferencia de altura que exista entre el jugador y el enemigo a renderizar. Con esto obtendremos un valor al que mover el Sprite sumando o restándolo a la mitad de la pantalla y (correspondiente al centro). Además, se ha diseñado una pequeña mejora la cual cuando estás muy cerca del enemigo, este Sprite baja en el eje Y para dar impresión de que cuando estás cerca le miras completamente a los ojos y no las piernas.

El último paso una vez posicionado y escalado el Sprite correcto es llamar a la función draw de la ventana y el enemigo estará dibujado correctamente por pantalla. Para evitar efectos ópticos dispares (como renderizar un enemigo lejano delante de un enemigo cercano, siendo que éste le debería de tapar) se ejecuta en orden descendente a la distancia al jugador, dibujando primero a los más lejanos y luego a los más cercanos, para que esos “tapen” a los enemigos de atrás, si existen.

3.8 Resolucion y framerate

El último aspecto gráfico y técnico que se va a cubrir es la resolución de renderizado, así como la tasa de refresco del juego. La resolución del juego original era de hasta 320x200 píxeles de alto y ancho, y funcionaba a una tasa de refresco fijada a 35 cuadros por segundo.

3.8.1 Resolucion interna

En nuestra versión al principio se renderizaba a 1920x1080 y se le daba a la ventana del juego el mismo tamaño que la resolución interna del juego, algo que posteriormente se cambió, gracias a SFML. Para ello, la ventana se crea de un tamaño 4 veces superior a la resolución interna, aunque la ventana puede escalarse en el ordenador del jugador. El renderizador conoce la resolución a la que funcionar desde que se inicializa, y acorde a ello, mantiene constantemente un array de tipo `uint8_t` de un tamaño equivalente a 4 veces la resolución del renderizado. Esto no tiene nada que ver con el tamaño de la ventana, sino que es para tener canales R, G, B y Alfa. Así, el renderizador no hace las llamadas de SFML para pintar en pantalla directamente, sino que usa este array como un buffer en el que ir pintando el frame y luego con él se crea una textura, que por funcionamiento de SFML, se usa para crear un sprite que se muestra en toda la ventana. De esta forma se consigue desacoplar resolución interna y tamaño de la ventana, imitando aún más el aspecto del juego original.

También hay que decir que esto solo aplica al renderizado del escenario, no al HUD, la escopeta o los enemigos, estos funcionan diferente pero también son sprites de SFML por lo que escalarán si se redimensiona la ventana.



3.8.2 Framerate interno y bucle principal

El juego se puede bloquear para no pasar de una tasa de refresco determinada, y de hecho, es lo que se hace. Se fuerza una tasa global de refresco de 35 actualizaciones por segundo, es decir, 35 iteraciones al bucle principal.

Este número no se ha elegido al azar ya que es la tasa que mantenía el juego original de 1993, no obstante, se ha comprobado que nuestro clon es capaz de ejecutarse a 60 *frames* por segundo, el estándar de hoy en día, de forma fluida y constante.

Para conseguir esta tasa de refresco no se ha utilizado ninguna función de la librería SFML que lo fuerce, sino que se han realizado mediciones de tiempo al principio de cada iteración del bucle principal y al final, y luego se invoca a una función llamada `Delay()` que se encarga de no hacer nada hasta que pasa el tiempo que falta para asegurar el tiempo objetivo de cada iteración (1 / 35 de segundo).

El main loop, por tanto, es el siguiente:

```
while (!doom_game.IsOver())
{
    deltaTime = deltaClock.restart();
    doom_game.setDeltaTime(deltaTime.asSeconds());
    doom_game.ProcessInput(doom_game.getStatus());
    doom_game.Update();
    doom_game.Render();
    elapsed = deltaClock.getElapsedTime();
    doom_game.Delay(elapsed.asSeconds());
}
```

Figura 18 - Bucle principal del juego. Se encuentra en main.cpp

```
void Game::Delay(float elapsedFrameTime)
{
    float timeToWait = (1.0f / (float)TARGETFRAMERATE) - elapsedFrameTime;
    if (timeToWait > 0)
    {
        sf::sleep(sf::seconds(timeToWait));
    }
}
```

Figura 19 - Función encargada de asegurar la tasa de refresco deseada

La resolución interna y framerate se encuentran definidos en el fichero de código `doomdef.h`

4. Sonido

La música que se escucha actualmente en el juego, al igual que todos los efectos que se oyen para sumergir al jugador en la experiencia DOOM no están obtenidos del propio archivo wad de datos. En su lugar, se ha usado la herramienta de la librería SFML para reproducir sonidos, ya que proporcionaba un sonido más claro y facilidades a la hora de editarlo, como por ejemplo con el volumen de los mismos. Esto significa que cuando se carga un archivo de sonido nuevo, ya sea a la hora de creación de enemigos o generando el nuevo player entre niveles, se llama a la función encargada de controlar el volumen de los mismos con el valor ajustado por el usuario al inicio del juego, en la pestaña de opciones del menú principal.

Estos están divididos en Music o sound, dependiendo de si se busca un efecto pequeño o es la música ambiente del nivel.



Respecto a la clase Music, esta carga por streaming el fichero wad, y por ello su uso se reduce a la música ambiente que se escucha tanto en el menú principal, como en cada uno de los niveles y en las pantallas de descanso entre niveles con carga del siguiente nivel. Al ser por streaming está limitado a una reproducción de la clase music simultáneamente, por lo tanto, se está obligado a parar la canción actual antes de reproducir la siguiente.

La clase Sound carga todo el fichero en memoria antes de reproducirlo, con lo cual está diseñada especialmente para efectos de sonido de breve duración, tales como disparos de arma, heridas, etc. Estos se usan en la clase Player para representar el efecto de bala al disparar, el sonido de herida al recibir un disparo y el efecto de muerte. También tienen utilidad en la clase Soldier que representa a los enemigos, en donde cada tipo de IA cuenta con sus propios sonidos de despertar, ser heridos y morir acorde con su personalidad elegida al ser generados entre las tres posibles: aleatoria, agresiva o escurridiza.

Para finalizar con los sonidos añadidos, se han añadido efectos especiales de sonido en los menús para darle ese toque doom, pequeños detalles pero que sin ellos no parece la misma interfaz. Exactamente se han añadido, por ejemplo, efectos de disparo de pistola mientras navegas por el menú principal o el efecto de recarga cuando saltas las estadísticas al terminar un nivel.

5. Jugabilidad

5.1 Logica de juego

El juego se divide en diferentes pantallas y estados, cuya unión da lugar a una partida en donde se pueden modificar las opciones, elegir dificultad, jugar hasta completar todos los niveles y poder ver así la pantalla de desenlace.

En un primer estado llamado inicio, el jugador se encuentra en la pantalla de introducción, y al saltarla, llegará al menú principal. En él puede hacer diversas opciones, como salir del juego, ver los créditos o entrar en opciones en donde podrá regular el sonido del juego en 4 posibles valores. La cuarta opción de este menú será empezar a jugar, la cual, al darle, mostrará cuatro posibles dificultades para el juego. Estas dificultades modifican aspectos del juego como número de enemigos por nivel o la cantidad de puntos de vida, armadura y munición que regeneras tras cada pantalla.

Una vez seleccionada la dificultad, da comienzo la partida. La lógica interna pasa a un estado de game en donde se renderiza el escenario con los recorridos BSP del escenario, los enemigos, etc. En cualquier momento del nivel el jugador puede pulsar la tecla ‘esc’ para pausar el juego, con lo cual el personaje quedará estático, al igual que los enemigos. Para reanudar el juego hay que volver a pulsar la tecla ‘esc’ o bien, si se desea salir, también es posible desde este estado. Por otra parte, si el jugador muere, tiene la opción de reiniciar todo el nivel desde el principio con los mismos ajustes de dificultad. El nivel se completa cuando el jugador llegue a la salida, marcada con el cartel de “exit”.

Cuando termine un nivel, pasará a la pantalla de puntuaciones en donde se le da un descanso y se muestra el porcentaje de enemigos asesinados durante dicho nivel. Cuando quiera saltar esta página, dará paso a la carga del siguiente nivel para repetir el bucle.

Una vez haya superado los tres niveles, se mostrará una escena del final del juego con el desenlace de la historia y volverá al menú principal para poder empezar una nueva partida.



5.2 Dificultad

Doom cuenta con 4 niveles de dificultad, dependiendo de las habilidades del propio jugador con los first person shooters:

- **Soy muy joven para morir**: Nivel de dificultad muy fácil. Está diseñado para que con él cualquier persona pueda llegar a completar el juego. El número de enemigos se limita en gran medida y obtienes grandes recompensas de vida y armadura cuando terminas cada nivel.
- **Hazme daño**: Dificultad recomendada para jugadores con conocimientos en este tipo de juego. La cantidad de enemigos en el mapa es decente y el último nivel supone un desafío ya que se limita la obtención de recursos cuando completas el nivel, con lo cual es muy probable que se llegue herido.
- **Ultra violencia**: Dificultad recomendada para profesionales. La cantidad de enemigos es desorbitada. Planear cada paso y con cabeza es la clave para superar esta dificultad. Los recursos obtenidos son iguales que en la segunda dificultad, sumado al aumento de enemigos hace que la cantidad de munición del jugador sea otro factor importante a tener en cuenta.
- **PESADILLA**: Para quienes buscan realmente un desafío. Cantidad desorbitada de enemigos y sin obtener recursos entre niveles. Aprovecha cada cobertura y cada bala, pues lo necesitarás. Hasta a los desarrolladores nos parece injusta esta dificultad.

5.3 Diagramas de navegacion

En este diagrama se resume cómo es la navegación a través de las pantallas tal y como se ha desarrollado en la lógica del juego:

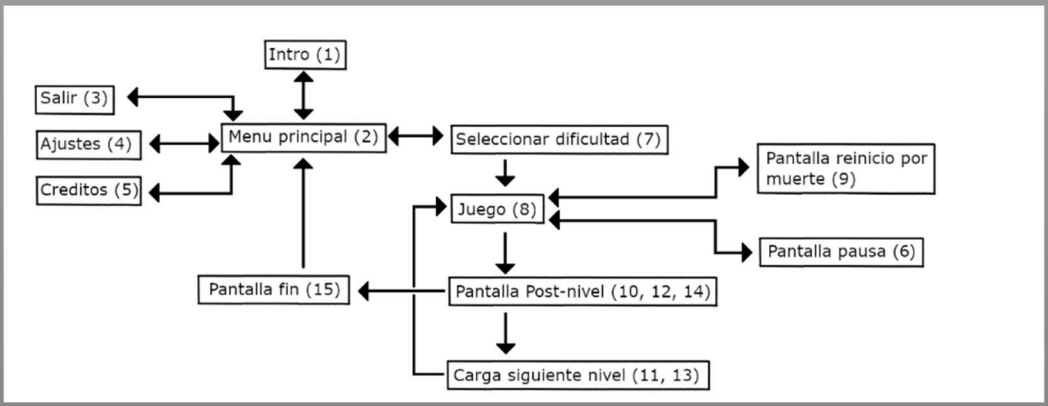
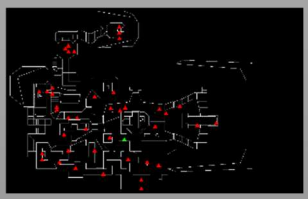


Figura 20- Diagrama que ilustra el flujo del juego



En la siguiente tabla se muestran las pantallas de cada uno de los apartados mencionados:

			
1.Intro	2.Menu Principal	3.Salir	4.Ajustes
			
5.Creditos	6.Pausa	7.Seleccion dificultad	8.Juego
			
9.Reinicio muerte	10.1º Post pantalla	11.Carga lvl2	12. 2ºPost Pantalla
			
13.Carga lvl3	14.3ºPost Pantalla	15.Pantalla fin	Extra: Vista 2D

6. Jugador

En lo referente a la implementación del jugador, hay varios aspectos que comentar, como por ejemplo el funcionamiento de su movimiento, las colisiones con el entorno y el de su arma, la escopeta, aparte de cosas menores como el funcionamiento de la salud y la armadura y en qué se diferencian.

6.1 Movimiento del jugador

El juego original contaba con dos formas de mover al personaje: La primera, el movimiento habitual hacia adelante, atrás, izquierda y derecha, aunque el movimiento lateral estaba mapeado a las teclas alt + flecha, lo que lo hacía poco conocido a muchos jugadores. Y la segunda, donde las flechas adelante y atrás eran para mover al personaje en esas dos direcciones, mientras que la izquierda y la derecha servían para rotar al jugador sobre su propia posición.

Se decidió implementar este esquema de control, ya que resultaría más sencillo hacer cálculos referentes al movimiento en un solo eje que en dos a la vez, sobre todo si hay que controlar el movimiento en diagonal, por ejemplo. Hay un tercer sistema de control (mover el ratón hacia



delante mueve el jugador en esa dirección y se rota moviéndolo hacia los lados, pero es tan incómodo que ni se planteó como opción en ningún momento).

En la implementación, el motor lee las teclas y si están presionadas marca el flag de que así es, y más adelante, en la función Update() se llama a la de movimiento del jugador, donde se comprueban estos flags uno por uno y se actúa acorde a todos ellos.

6.1.1 Desplazamiento frontal

Para el movimiento hacia adelante y atrás, se aplica directamente una velocidad al jugador. El juego original aplicaba aceleración y también un rozamiento, y en nuestro código se puede ver algún rastro de que esto iba a ser incluido, pero se dejó para el final en caso de que sobrara tiempo para mejoras y finalmente se decidió pulir más otras áreas. No obstante, el resultado es muy convincente y fluido, aunque no sea completamente fiel al original.

Se consultaron diferentes fuentes para darle al jugador una velocidad de desplazamiento lo más cercana posible a la del juego original. Tras esto, se determinó que las velocidades de movimiento serían:

- Andando: 280 unidades de mapa por segundo
- Corriendo: 560 unidades de mapa por segundo

Y la fórmula utilizada para calcular la nueva posición del jugador es la siguiente:

$$pos'_x = pos_x + \cos(\theta) * vel_{mov} * \Delta t$$
$$pos'_y = pos_y + \sin(\theta) * vel_{mov} * \Delta t$$

Figura 21- Cálculo de la nueva posición del jugador en cada frame

Donde Δt se refiere al tiempo que ha transcurrido entre las dos últimas actualizaciones completas del motor del juego, es decir, los dos últimos *frames* renderizados, que incluyen el cálculo de la posición. θ hace referencia al ángulo de rotación del jugador, y el signo de la velocidad de movimiento del personaje será negativo si el movimiento es hacia atrás.

6.1.2 Rotacion a izquierda y derecha

La rotación se ha implementado de manera muy similar al desplazamiento, teniendo dos velocidades dependiendo de si se está pulsando la tecla de correr o no (asignada a la Mayúsculas Izquierda). Las velocidades de rotación son:

- Andando: 100 grados por segundo.
- Corriendo: 150 grados por segundo.

Una vez más, se han puesto los valores a mano tras haber consultado diferentes fuentes que hablaban del funcionamiento de esta mecánica, y tras haber comparado con el juego original, para mantener la sensación lo más fiel posible.

El cálculo del nuevo ángulo de rotación es:

$$\theta' = \theta + vel_{rot} * \Delta t$$

Figura 22 - Cálculo de la rotación del jugador



6.1.3 Actualizacion de la posicion vertical del jugador

Como todo buen juego 3D, también se actualiza la altura a la que se encuentra el jugador, aunque en este caso se ha hecho de manera simple. En cada llamada a Update(), se fija que el jugador se encuentra a la altura del sector en el que se encuentre. Para ello, se recorre otra vez el árbol BSP hasta encontrar el sector en el que está, algo que en 1993 no sería óptimo, pero en 2021 no va a hacer sudar al procesador. Además, a este valor se le suma el de la altura de los ojos del jugador, que por defecto es 41.

El único problema es que, como ya se verá un poco más adelante cuando se hable de colisiones, en algunos casos esto implica que el jugador escalará inmediatamente algunos desniveles elevados que no debería, como el camino hacia la puerta del mapa E1M3, o si se cae a algún pozo como el de la zona abierta del segundo mapa podrá subir de vuelta hacia arriba con solo caminar hacia el camino del que se ha caído.

Somos conscientes de ello, pero se ha decidido mantener ya que a esta versión no se han podido introducir las partes de suelo que pueden subir y bajar, incluyendo por ejemplo las escaleras de justo antes del final de E1M3 que normalmente no están hasta que se pulsa un interruptor (no implementado en nuestra versión), o las puertas, que son la sección superior de un portal que llega hasta el suelo, pero puede subir hasta el techo.

De esta forma, ese teletransporte en la altura en algunas circunstancias permite mantener algunas partes de los niveles accesibles sin tener que implementar esas plataformas, algo que se deseaba hacer, pero el tiempo disponible y la complejidad global del juego no han permitido hacer. Si se tuviera más tiempo, sería una de las primeras características que se añadirían.

6.2 Movimiento de la cabeza y el arma

Una característica que se incluyó justo antes de la entrega beta y reunió gran parte de los comentarios positivos fue el cabeceo característico de DOOM al mover al jugador hacia arriba y hacia abajo para dar la sensación de mayor fluidez en sus movimientos.

Se consiguió modificando el código de Update() que fijaba la posición de los ojos a la del suelo sumado a la altura del jugador. Ahora entra en juego una tercera variable *offsetHeight* que se encarga de añadir esta pequeña variación en la altura, positiva o negativa.

Se calcula de la siguiente manera:

$$offsetHeight = \sin\left(\frac{2\pi}{40} * step\right) * \frac{scale}{(\Delta t * \text{FRAMERATE})}$$

Figura 23 - Cálculo del cabeceo del jugador al desplazarse

En esta ecuación algunos valores fueron ajustados a mano para parecerse a la versión original. En este caso, *step* es un número entre 0 y 40 para tener 40 posibles valores del rango del cabeceo del jugador. El valor *scale* es la altura a la que puede llegar a subir (o bajar) la cabeza cuando el término del seno alcanza su valor máximo, que depende, de nuevo, de si el jugador está corriendo o andando:



- Andando: *scale* tiene un valor de 10.
- Corriendo: *scale* tiene un valor de 15.

De esta forma, el cabeceo del jugador será más exagerado si corre que si anda, aunque la velocidad con la que alcanza su máximo y mínimo es la misma.

El último término que incluye a Δt es para hacer el movimiento independiente de la velocidad de ejecución, y para ello se utilizan el tiempo entre frames y FRAMERATE, que hace referencia a la tasa de refresco a la que se fuerza el juego. En el original eran 35 *frames* por segundo, así que en este se ha reproducido esa tasa, aunque si se edita `doomdef.h` se puede fijar a 60.

Para el movimiento del Sprite de la escopeta se utiliza una técnica parecida, utilizando una función senoidal y también se divide el rango en pequeños *steps*. No obstante, no se muestra porque es muy parecida y un poco más engorrosa ya que se tienen que diferenciar casos especiales como si es el primer ciclo completo del movimiento, para evitar que el Sprite se mueva doble hacia la derecha solo esa vez y se descentre.

6.3 Colisiones

Las colisiones entre el jugador y las paredes fueron una de las últimas cosas en implementarse, debido a la urgencia de otros apartados, aunque este también fuera urgente y vital para tener una buena sensación de jugabilidad por parte de los usuarios.

A pesar de que en principio parecían difíciles, se logró implementar un modelo funcional de colisiones bastante rápido, aunque no perfecto al 100%. Se explican a continuación los pasos que sigue nuestro algoritmo de detección y tratamiento de colisiones:

1. Antes de calcular las posiciones nuevas del jugador, almacenamos las viejas
2. Calculamos las nuevas posiciones del jugador aplicando el movimiento que corresponda, pero solo en X y en Y. No aplicamos las nuevas posiciones de momento.
3. Utilizando las posiciones "viejas" del jugador se calcula para todos los SEGS del sector donde se encuentra el jugador:
    a. Trazamos 32 rectas alrededor del jugador, equidistantes en ángulo entre sí.
    b. Para cada una, se calcula si hay intersección con la recta definida por los vértices del SEG. Si no hay, hemos terminado.
    c. Si hay intersección a una distancia menor o igual que la **anchura del jugador**, se comprueba que sea en un punto entre V1 y V2 del SEG, no en un punto que pertenezca a la recta matemáticamente, pero no al muro.
    d. Si esto se cumple, se comprueba en los Flags del muro que sea colisionable. Si lo es, hay colisión y si no, no la hay.
4. En caso de que no haya colisión, se repite el proceso desde el paso 3 utilizando las nuevas coordenadas calculadas para el jugador.
5. Si para las coordenadas viejas o para las nuevas se ha detectado una colisión, no se aplican las nuevas coordenadas al jugador. En caso contrario, sí se le aplican.
6. Una vez se sabe la posición en X y en Y, se calcula la nueva altura como ya se ha descrito anteriormente.

Figura 24 - Algoritmo de detección y tratamiento de colisiones



Este código se encuentra en el motor del juego, en Update() y en la función CollisionDetect(). Es reutilizable también para los enemigos, con quienes se sigue el mismo procedimiento poco después de haberlo aplicado al jugador.

En teoría no debería ser necesario calcular que las coordenadas anteriores del jugador colisionen, pero se observó que en algunos casos muy concretos ayudaban a prevenir que el jugador atravesara las paredes.

En cuanto a la anchura del jugador, se ha utilizado un valor de 16 unidades, de forma que se puede pensar en que el jugador está encerrado en un círculo de radio 16. Este valor se extrajo del código original del juego, aunque este maneja las colisiones diferente, tanto en algoritmo como en estructura de datos, ya que no se les ocurrió hasta después del lanzamiento que se podía encontrar el sector del jugador para comprobar solo con esos muros. Para optimizar más, tenían el BLOCKMAP, una cuadrícula en la que cada pequeña región listaba los segmentos con los que intersectar con el jugador si se encontraba dentro de esa casilla, y venía almacenado en el wad.

6.3.1 Limitaciones de nuestras colisiones y puertas

El algoritmo no es perfecto, pero se ha pensado para ser implementado de forma rápida y cumple su función lo suficientemente bien. En un mundo ideal se habría intentado hacer como se explicó en las clases de la asignatura, encontrando el momento exacto de la colisión y previniendo solo la parte del movimiento en la que se atraviesa el muro. Además, como ya se ha explicado al hablar del cálculo de la altura, algunas paredes elevables o que forman parte de un Portal no tienen el flag deseado activado, permitiendo ser escaladas al instante al acercarse a ellas. Se ha dejado así ya que no se han implementado las plataformas elevables.



Figura 25- En el tercer mapa, E1M3, deberían levantarse unas escaleras hacia la salida del nivel al accionar un interruptor. En su lugar, hay que caminar hacia la pared y se escala automáticamente.

Por el mismo motivo, las puertas que se ven cerradas en el mapa no se abren, así como las paredes que esconden secretos con las que se debía interactuar, ambas deben ser atravesadas por el jugador. Esto sería lo primero que se mejoraría en caso de tener más tiempo de desarrollo.



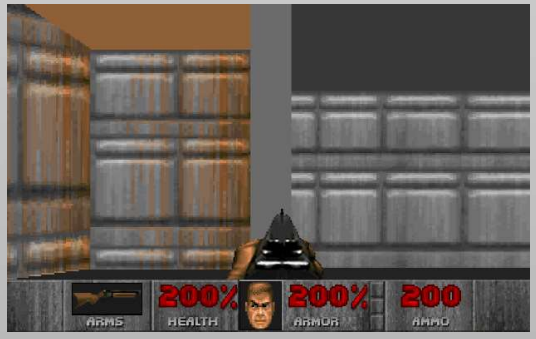


Figura 26- Atravesando la primera puerta de E1M1, el primer nivel.

Además, se pensó en hacer como en el juego original y desplazar al jugador a lo largo de la pared si colisiona con ella, pero a falta de unos pocos días para implementarlo, se tuvo que descartar la idea, que habría proporcionado una experiencia de juego y movimiento más fluidos. Esto sería probablemente otro de los primeros puntos que se mejorarían en un futuro.



Figura 27 - En el original se movería paralelo a la pared al intentar desplazarse hacia adelante, en nuestro clon es imposible moverse hacia adelante en esta situación y se debe separar del muro.

6.4 Disparo del Jugador

El disparo del player se activa cuando en la fase de readInput el jugador pulsa una de las dos teclas control. En ese momento, en la fase de update se realizan varias acciones: bloquear el flag de disparo, se inicia un thread de cambios de Sprite de la escopeta y se calcula a qué enemigo ha impactado la bala.

Este nuevo hilo se encarga de alternar el Sprite cada cierto tiempo y se vuelve a dormir, para conseguir ese efecto de animación de escopeta. Cuando termina la animación, desbloquea el flag de disparo para permitir volver a disparar y finaliza su ejecución.

Con respecto al enemigo impactado, se recorre la lista de enemigos actuales en el mapa, ordenados del más cercano al más lejano. Como es obvio, el disparo impactará antes a un enemigo a 50 unidades de distancia que otro a 200, siempre y cuando estos enemigos estén alineados. Para cada enemigo, se comprueba si es visible (pues, si no, sería como golpear a una pared) y luego el ángulo que forman con respecto al jugador. Para distancias cortas se acepta que el enemigo esté a +-12 grados (dando un arco de 24 grados en total) con respecto al centro de la pantalla, puesto que el Sprite es más grande y se acepta la idea de que le dé en el brazo.

Este arco se va disminuyendo conforme el enemigo esté mas alejado, llegando a ser de +-6 grados (12 en total) para enemigos lejanos. En el momento en donde se reconozca un impacto, se reconocerá que ese es el enemigo al que hay que restarle su salud y terminará la búsqueda.



Respecto a la vida que pierde el objetivo, al ser una escopeta hay que tener en cuenta que un golpe certero consigue que más perdigones impacten contra su objetivo que en el caso de que se dispare en, por ejemplo, el borde del brazo, además de que hay mas órganos vitales cuanto más en el centro se dispare. Esto se ha traducido a que el daño que se infringe a un enemigo es proporcional cuán alejado esté del centro de la pantalla, obteniéndose así el siguiente efecto:

- Si el disparo ha impactado de lleno en el blanco (ha rozado la perfección y esta desviación mínima de +-1 grado en situaciones normales, el objetivo sufre una muerte instantánea (aumenta o disminuye ese valor en distancias muy largas o cortas)

- Si se ha desviado +-4 grados con respecto al centro, el objetivo sufrirá 60 puntos de vida (un enemigo normal tiene 100 puntos de hp)

- Si se ha desviado más de la anterior marca, ha sido un disparo reconocido al borde, con lo cual sufrirá daño mínimo, 40 puntos de salud.

Las medidas de daño se han estudiado para que se pueda realizar el juego en dificultad pesadilla (sin recarga de munición) matando a todos los enemigos sin necesidad de hacer todos los disparos casi perfectos. Por otro lado, penalizar los golpes muy lejanos si son reiterados. Véase que, de dos golpes externos el objetivo sobrevive, pero con 1 disparo intermedio y el siguiente externo sus puntos de vida se reducen a 0.

7. Enemigos: Inteligencia Artificial

Todos los enemigos salen de la clase abstracta Enemy. Esta clase contiene varias funciones y valores que todos los enemigos deben tener, entre los que se encuentran su posición x e y, la vida, funciones de realizar y recibir daño, movimiento, etc. El comportamiento de los enemigos depende de estados, los cuales transicionan en un orden concreto.

7.1 Salud de los enemigos

Los enemigos empiezan siempre en un estado sleep en donde su único trabajo es esperar. Se podría decir que es el sinónimo de enemigo no alertado, y, mientras permanezca en este estado, ni se moverá ni disparará.

Para despertar o alertar a un enemigo tiene que ocurrir una de las tres opciones:

1. El jugador se mueve cerca suyo (350 unidades de medida).
2. El jugador dispara en un área próxima (900 unidades de medida).
3. El enemigo recibe daño por parte del jugador.

Cuando un enemigo despierta, inicia un proceso externo que se encarga de calcular sus siguientes movimientos y/o de disparar. Con esto se consigue que se auto actualice cada cierto tiempo un estado de movimiento, la dirección a donde mira el enemigo y cuándo le toca disparar. Esta parte es diferente en cada tipo de enemigo e IA, y se explicará en el siguiente apartado, pero en lineas generales su trabajo consiste en cambiar el estado, angulo y dormir, o disparar y ejecutar la rutina de disparo.



La rutina de disparo consiste en un sleep corto de medio segundo en donde se poen el estado de apuntando para que el sprite a renderizar sea el de disparar, luego el estado disparando en donde se renderiza durante un pequeño periodo de tiempo el sprite con el fuego del arma y luego, de nuevo, otro cierto tiempo apuntando antes de volver al movimiento. Para comprobar si ha impactado o no un disparo, se comprueba por rng si ha impactado. Hay que decir también que si el jugador está corriendo hay menos probabilidad de que golpeen las balas, puesto que es más dificil apuntar a un objetivo en movimiento que a uno parado.

El daño que hacen dichas balas corresponde a un valor 15 fijos al que se le puede o sumar un total de 10 de daño para hacer 25 o restarle hasta 7 de daño para infligir una cantidad de 8 puntos de vida, dependiendo cuán alto ha sido ese valor a la hora de impactar.

Con respecto al movimiento, por su parte, calcula el ángulo de visión al que mira el personaje para que, en el momento de updatear los valores del juego, se mueva un espacio equivalente a deltaTime unidades hacia esa dirección. Una vez obtenidas la dirección, se usa la misma fórmula para aumentar sus coordenadas x e y en el proceso de update.

$$pos'_x = pos_x + \cos(\theta) * vel_{mov} * \Delta t$$
$$pos'_y = pos_y + \sin(\theta) * vel_{mov} * \Delta t$$

Figura 28- Cálculo de la nueva posición en cada frame

De igual forma, se usa el mismo proceso para calcular si el enemigo colisiona con un objetivo. Toda la IA e inteligencia del enemigo acaba cuando este muere, pues los dos procesos de elección de movimientos terminan, pasa a un estado de death y el único proposito de continuar con la entidad enemiga en memoria es el conocer la ubicación en donde murió para renderizar su cadáver.

7.2 Tipos de enemigos: IA

Hay 2 tipos de enemigos, pero uno de ellos dispone de 2 comportamientos totalmente diferentes, con lo cual se puede hablar de 3 tipos de enemigos en total, correspondientes al número de sonidos para despertar y morir de la IA:

Zombie: enemigo básico dentro del juego DOOM. Es la que dispone de 2 IAs distintas. A simple vista no se puede diferenciar contra cuál de las dos te enfrentas hasta que no se despiertan, pues su marca distintiva es el sonido que hacen tanto al despertarse como al morir. La primera IA básica fue la que se presentó en la beta, a base de números aleatorios decide si prefiere atacar o moverse hacia una dirección aleatoria nunca repetitiva de la anterior y la mantenga durante un tiempo de entre medio segundo y 3 segundos, aunque se asegura que disparará (con mayor o menor precisión) tras 5 movimientos que no hayan sido disparos. Se podría definir como los científicos convertidos con armas sin mucha cabeza.

La segunda IA de los zombies se ha titulado escurridiza, puesto que depende de movimientos rápidos hacia la derecha o izquierda del jugador. Esto se consigue calculando el ángulo que tiene que obtener el enemigo desde su posición con respect al player y sumando o restando 90 grados a dicho eje. Con esto se consigue que haga “círculos” con el centro el propio jugador, haciendo que este tenga o muy buena punteria o necesitando varios cartuchos para acabar con ellos.



Siempre dispara tras 5 movimientos y, como característica especial, el tiempo que necesita para disparar se reduce en gran medida (siendo de 0,66 segundos, mientras que el resto de los enemigos necesita de 1,1 segundos) a cambio de un daño reducido si impacta. Además, se mueve a mayor velocidad que la otra IA vista, aumentando su valor de 120 a 200 por el tiempo delta (frecuencia de renderizado, de normal 1/60) y sus recorridos en dirección fija antes de volver a calcularlo pasan a ser de 0,2 segundos a 1 segundo. Lo definen como el marine que por acto reflejo sigue sus instintos de guerra y busca resguardarse de las balas de sus enemigos y buscar cobertura.

Sargento Zombie: Este es el segundo enemigo implementado en DOOM. Es fácilmente reconocible por su Sprite en negro y su rugido al detectar al jugador. Su IA busca automáticamente formas de acercarse al jugador lo máximo posible, y todos sus movimientos dependen de la distancia que le separe del jugador. La probabilidad de que dispare aumenta según disminuya esta distancia, siendo por ejemplo de un 70% de probabilidades de disparar al calcular su siguiente movimiento cuando está a menos de 90 unidades de distancia y se disminuye gradualmente, llegando a ser casi nula (1%) con distancias mayores a 600. Su movimiento consiste en calcular el ángulo para mirar a la posición del jugador y sumarle +-45 grados calculados aleatoriamente para que cueste más apuntarle. Debido a que se tiene que acercar al jugador, goza de un aumento de salud, haciendo que de media aguante un disparo extra y ofreciéndole así la oportunidad de castigar la mala puntería con uno o dos disparos hacia el jugador.



Figura 29- 1º Sprite frontal del Zombi y del Sargento, respectivamente

8. Conclusiones

En general el equipo está bastante satisfecho con el trabajo desempeñado durante el desarrollo de este clon, sobre todo teniendo en cuenta las dificultades adicionales que implica desarrollar un juego en 3 dimensiones, teniendo que desarrollar primero un motor que mueva el juego antes de poder poner una línea de código dedicada a los aspectos jugables.

Al principio se pensaba que sería un raycaster del estilo de Wolfenstein, el juego predecesor a DOOM lanzado por la propia id Software. Poco a poco nos dimos cuenta de la complejidad de renderizar un *frame* en pantalla era mucho mayor a la de este último, y que en cuanto a complejidad DOOM era muy superior. Aun así, aceptamos el reto de hacerlo e íbamos a conseguirlo.

El desarrollo del renderizador nos consumió una parte enorme del tiempo desarrollo, aunque finalmente mereció la pena, pues el apartado visual está bastante logrado y se ha conseguido el mayor compromiso posible entre tiempo disponible y calidad. Después de todo, apenas han sido poco más de dos meses y medio desde la entrega del GDD de principios de marzo.



Como consecuencia negativa de este desarrollo se han visto resentidas algunas características jugables. Aún con todo, las últimas semanas del desarrollo han sido frenéticas y a contrarreloj para poder llegar a implementar el mayor número de características posible y se ha conseguido, llegando a implementar en poco menos de tres semanas el mapeado de texturas, las colisiones, la inteligencia artificial y el renderizado de enemigos, la posibilidad de jugar en más niveles y de progresar entre ellos tocando el interruptor de la meta, la pantalla de *intermission* entre niveles, los disparos de la escopeta junto con la mayoría del sistema de sonido y el máximo refinamiento posible de la jugabilidad.

Ha sido una experiencia casi tan frenética y demoniaca como nuestro juego, y esperamos que os guste tanto abriros paso por el infierno con una escopeta como a nosotros programarlo.

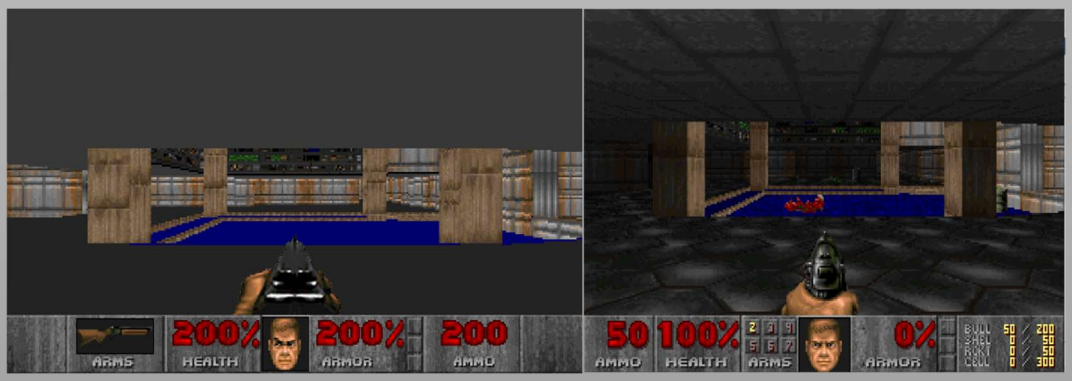


Figura 30 - Comparativa del resultado final entre nuestro clon y Chocolate DOOM, un port open source de DOOM



9. Referencias y bibliografía

https://github.com/amroibrahim/DIYDoom - Repositorio donde se explica el funcionamiento paso a paso del renderizador.

https://github.com/id-Software/DOOM - Código fuente del juego original

https://github.com/chocolate-doom/chocolate-doom - Código fuente del port a ordenadores modernos

https://doom.fandom.com/wiki/ - Wiki de DOOM con información útil sobre variables y funcionamiento del juego

https://fabiensanglard.net/doomIphone/doomClassicRenderer.php - Breve resumen del funcionamiento del id Tech 1

https://www.doomworld.com/forum/topic/102392-the-definitive-guide-to-visplanes/ - Resumen del funcionamiento de suelos y techos

https://www.sfml-dev.org/tutorials/2.5/audio-sounds.php - Tutorial de cómo usar las clases de audio

https://www.sfml-dev.org/tutorials/2.5/graphics-sprite.php - De igual forma, tutorial de cómo usar texturas y sprites en SFML

https://www.deviantart.com/elitesniper177/art/Doom-Sprite-Sheet-52238630 - Lista de sprites conseguidos a través de esta web

http://www.wolfensteingoodies.com/archives/olddoom/music.htm - Lista de todos los sonidos del videojuego

https://archive.org/details/gebbdoome - Game Engine Black Book: DOOM, libro muy muy completo donde se detalla mucho el funcionamiento interno del juego

https://c.eev.ee/doom-text-generator/ - Página para conseguir el texto con los colores múltiples característica de doom

https://www.reddit.com/r/programming/comments/gg298c/how_dooms_enemy_ai_works – Videotutorial de cómo funciona la IA.



Anexo I: Doom es 3D y no 2.5D

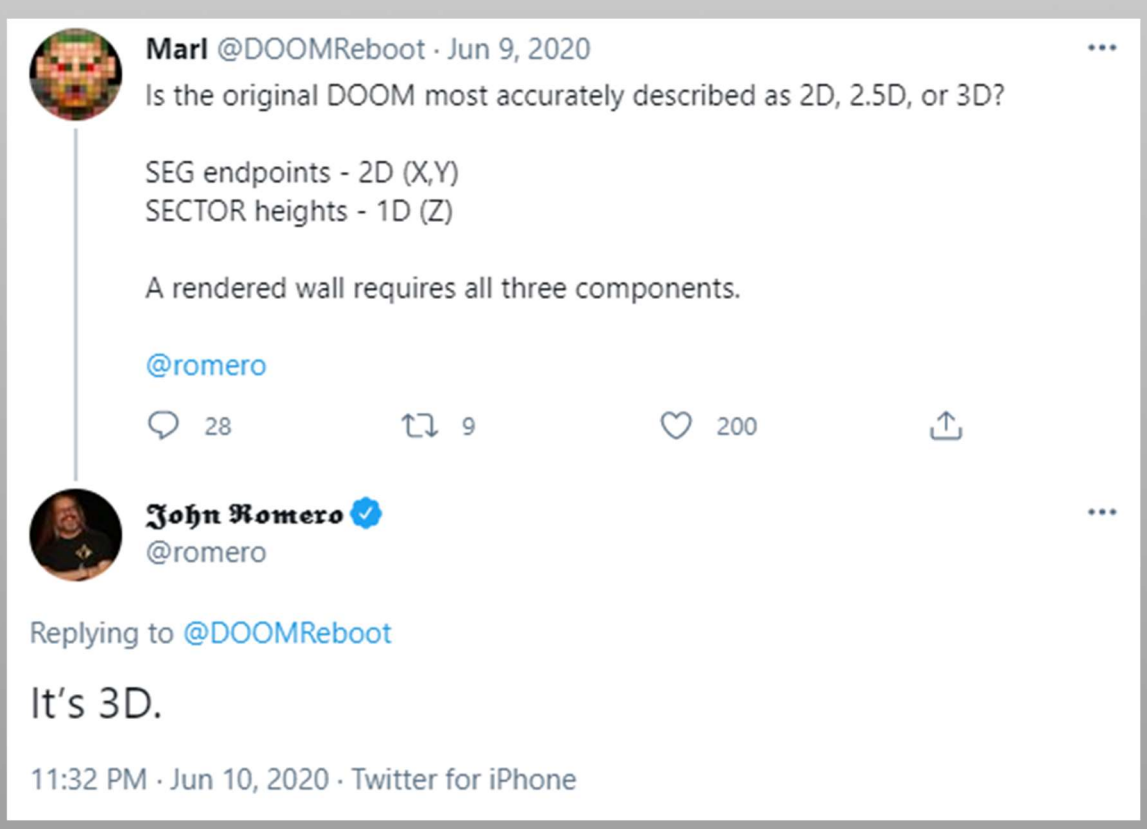


Figura 29 - Prueba irrefutable de que DOOM es 3D. John Romero es cofundador de id Software y diseñador del juego.

Anexo II: Controles del juego

- Flecha direccional arriba: Caminar / correr hacia adelante.
- Flecha direccional abajo: Caminar / correr hacia atrás.
- Flecha direccional izquierda: Girar a la izquierda.
- Flecha direccional derecha: Girar a la derecha.
- Control izquierdo: Disparar el arma.
- Mayúsculas izquierda: Mantener pulsada para esprintar.
- Enter: Confirmar acción en los menús.
- Escape: Retroceder en los menús / pausa durante el juego.
- Tabulador: Mantener pulsado para ver minimapa del nivel al jugar.
- Z: En la versión beta, activa y desactiva la inmortalidad del jugador.



Anexo III: GDD entregado en marzo



DOOM – Game Design Document

Grupo 4

Víctor Martínez Lascorz - 738845
Pedro José Pérez García – 75664

Zaragoza, 28 de febrero de 2021



Contenidos

Introducción 3

Historia y concepto 3

Género 4

Audiencia objetivo 4

Características principales 4

Fujo de juego 4

Mecánicas y experiencia del usuario 5

Personajes y enemigos 6

Armas 8

Consumibles 9

Audio y música 10

Arte y animación 11

GUI 11

Niveles 12



Introduccion

Este es el documento de diseño de Doom, un remake del clásico videojuego Doom realizado para la asignatura de videojuegos. En el presente documento de diseño se pretenden plasmar los conceptos clave del diseño de Doom.

Doom es un videojuego de acción en el que se controla a un soldado enviado a una luna de Marte en donde existe una estación de investigación para traspasar materia a través de portales. Por un fallo en su funcionamiento, de este empiezan a salir demonios y zombis que dominan en segundos a todos los científicos. Un escuadrón de soldados de élite al que pertenece el jugador es enviado allí con el objetivo de cerrar el portal y exterminar a los demonios. Al aterrizar allí son rápidamente masacrados a excepción del jugador. Como único superviviente, parar el apocalipsis en ese lugar depende ahora solamente del protagonista.

Doom es un shooter en primera persona con un alto contenido de acción frenética en donde se centra más en el rápido movimiento y ejecución que en estrategia, aunque también incentiva y premia la exploración del mapeado. Su estructura está dividida en varios actos que a su vez se dividen en niveles lineales (con inicio y final únicos y fijos) con múltiples enemigos, objetos recogibles y secretos cada uno.

Nosotros, el grupo 4 de la asignatura, estaremos al cargo de realizar un remake a la altura del original, donde las dosis de acción frenética están aseguradas. La experiencia shooter matademonios definitiva está aquí, y es nuestra visión de Doom.

Historia y concepto

Eres un marine de élite, de los mejores, entrenado para matar, en un mundo futurista en donde la humanidad ha colonizado diversas lunas, entre ellas Phobos y Deimos en Marte, las cuales se usan como centros de investigación. Entre estas dos lunas se diseña un sistema de portales para trasportar material en desarrollo. Este sistema funcionó a la perfección para objetos inertes, como pueden ser bolígrafos, papeles o pequeños instrumentos. En un momento dado se probó a traspasar sujetos humanos para experimentar, pero, al entrar por el portal, ocurrían cosas desconcertantes con ellos. Algunos nunca se llegaron a trasportar al otro lado del portal, otros de ellos salían muertos con síntomas de mordeduras y quemazos por toda la piel, y, los más afortunados estaban al borde de una esquizofrenia sin dedos y con la lengua cortada.

Los experimentos continuaron sin obtener ninguna respuesta clara, hasta que, pasado un tiempo, la luna de Phobos envió un mensaje de auxilio militar a Marte con urgente. Había zombis y demonios saliendo del portal de teletransporte y arrasando el lugar. En ese mismo momento, la segunda luna de Marte, Deimos desapareció en el espacio sin dar señales de vida.

Se realizaron numerosos intentos para volver a establecer conexión y contactar con Phobos, pero todos los intentos fracasaron. Las comunicaciones nunca fueron respondidas, los militares enviados allí no regresaban jamás, etc...

Como último intento desesperado, se envía un gran grupo de soldados a reconquistar la luna de Phobos de los demonios.



El personaje protagonista, Doomguy, se queda aguardando para ofrecer una salida, mientras que el resto de los soldados entran en el perímetro. Durante un largo tiempo, solo se escuchan disparos y gritos en la radio, y después el silencio.

En ese instante de silencio nuestro protagonista entra por la puerta de la base. Nada más entrar hay cadáveres de sus compañeros soldados. De todo el escuadrón, solo él queda con vida, y ahora todo depende de él. Hay que cerrar el portal y exterminar sin piedad a todo tipo de zombies y demonios que andan por las instalaciones.

Genero

Se trata de un shooter en primera persona, con la cámara a la altura de los ojos, en donde se tiene que ir avanzando sobre una serie de niveles con nuevos enemigos, armas y potenciadores para que el jugador supere diferentes desafíos y retos planteados.

Audiencia objetivo

Este juego no es recomendable para un público de corta edad, puesto que contiene altas cantidades de sangre, violencia e incluso terror.

En el resto de sectores del público que nos quedan, se puede establecer como objetivo una audiencia joven y a la que le guste la acción, o los entusiastas de lo retro, ya que después de todo estamos ante un clon del Doom original. Dentro de estos grupos se pueden encontrar el profesor y los alumnos de esta asignatura de videojuegos, por ejemplo.

Caracteristicas principales

Nuestra versión de Doom ofrecerá a los jugadores:

- Una experiencia inmersiva como la original de 1993 planteada por id Software.
- Gameplay ágil y dinámico, con controles similares al original.
- Planteamiento sencillo: Mata demonios y que no te maten para llegar a la salida.
- Escopeta o motosierra: un arsenal variado para elegir cómo atravesar el infierno.
- La música deberá ser como en el original, atrapante y que haga vibrar a los jugadores.
- Se fomentará la exploración del mapeado mediante secretos ocultos.

Flujo de juego

En Doom, cuando el jugador inicia el juego, se muestra un menú principal que ofrece al jugador diferentes opciones, como cerrar el juego, cambiar ajustar la configuración o iniciar una partida nueva. De momento nos centramos en esta última. Si el jugador decide iniciar una partida nueva, le preguntará al usuario qué episodio desea jugar y la dificultad deseada, y en el caso de que sea la más alta le pedirá confirmación, para dar sensación de que se enfrenta a un desafío mayúsculo.



Dentro del juego en sí, los niveles de cada episodio estarán organizados de forma secuencial, de forma que al completar uno al llegar a la salida se pasa automáticamente al siguiente (Puede haber excepciones con alguna salida secreta que lleve a niveles alternativos, aunque tras completarlos se seguirá el orden normal de niveles)

El *gameplay loop* se basará en recorrer un mapa tridimensional mientras se eliminan a los múltiples enemigos que intentan matar al jugador, recogiendo los objetos que haya por el suelo para restablecer la salud y la armadura, o para obtener nuevas armas o ventajas temporales como ser invisible unos minutos.

El jugador podrá cambiar de arma en todo momento y podrá disparar solo si el arma tiene munición. También podrá interactuar con el entorno aprovechando la tridimensionalidad de este y las diferencias de altura del escenario, con elementos como por ejemplo interruptores que accionen plataformas elevadoras, o puertas aparentemente invisibles, camufladas como paredes.

Cuando el jugador llegue a una salida, si interactúa con ella se le informará de su tiempo y su progreso en una pantalla que servirá de pausa entre niveles. Al volver a pulsar alguna tecla, se cargará el siguiente nivel, donde se procederá de forma parecida.

Si la salud del jugador llegara en algún momento a 0, este cae al suelo abatido y muere, pero no representa el final de su partida, sino que al pulsar un botón se reiniciará el nivel, pero penalizando al jugador, haciéndole perder todas las armas y municiones obtenidas hasta el momento menos aquellas con la que se comienza el primer nivel.

Si se pausa la partida, se mostrarán en pantalla unas opciones parecidas a las del menú principal, con una disposición parecida, y mostrando de fondo la partida congelada hasta que el usuario decida retomar su partida.

Si el usuario decide salir del juego, ya sea desde el menú de pausa o el principal, se le solicitará confirmación de la acción, junto con un mensaje de mofa que le anime a seguir jugando.

Mecanicas y experiencia del usuario

Para conseguir que el usuario se sienta inmerso en el juego y en la misión de exterminar los demonios que querrán acabar con su partida, no se le puede mantener alerta en todo momento usando una cantidad de enemigos desproporcionadamente injusta, sino que hay que buscar diferentes mecanismos y mecánicas del gameplay que generarán satisfacción o sensación de desafío en el usuario y que propiciarán de manera mucho más fácil su inmersión en la partida. Ejemplos de esto son:

- **Es un juego demoniaco.** Hay que matar demonios, y la música va a apoyar esta atmósfera de violencia mezclada con acción y frenetismo. Se puede apoyar este punto desde diferentes sitios, como por ejemplo al cerrar el juego, donde se mostrarán mensajes que tienten al jugador a seguir la partida.
  Se trata de mantener una sensación de tensión y hostilidad continua pero no muy intensa y que presente picos. Por ejemplo, se puede hacer que aparezcan más enemigos de una pared falsa cuando esa sala se creía vacía.



- **Salud y armadura.** Como en cualquier juego del género, la salud está presente, generando presión sobre el jugador cuando está a punto de agotarla y de morir.
  Ahora también aparece la armadura como otra variable a tener en cuenta para el jugador. Esta, siempre y cuando no esté a cero, absorberá una parte del daño que recibiría la salud del jugador y se desgastará. Con esto se logra que el jugador tenga que buscar y recoger del suelo los botiquines y las armaduras que pueda, ya que la recompensa por hacerlo puede ser mejor que ignorarlos.

- **Secretos e interacción con el entorno.** En Doom el jugador puede interactuar con el entorno, desde disparar a barriles que explotan y dañan a los enemigos hasta descubrir pasadizos o recoger llaves que son indispensables en algunos casos para avanzar de nivel. Todo esto ayuda a crear una sensación de inmersión del jugador con el juego, ya que le resulta satisfactorio y desafiante (pero no demasiado) el tener que estar alerta en el mapa para no perderse nada, y para sacarle el máximo provecho a las opciones de las que dispone ante los enemigos que tenga en frente.

- **Variedad de armas.** En el juego habrá una variedad amplia de armas para que los jugadores elijan la que más se ajusta a su forma de juego, o la más efectiva para cada situación, o la que más mata. Además, la mayoría deberán ser encontradas en los niveles para poder usarlas, lo que puede obligar a explorar los mapas a los jugadores, antes que lanzarse corriendo a por la salida. Esta exploración también puede crear una sensación de que los niveles son más largos de lo que parecen, potenciando la inmersión por parte del jugador.

- **Munición.** Las armas no tienen por qué compartir tipos de munición, y si se quiere usar únicamente un tipo de arma habrá que tener cuidado de no fallar los disparos para ahorrar munición. En el caso de Doom, algunos enemigos abatidos proporcionarán munición, lo que refuerza la conducta que se busca en los jugadores, que no ignoren a los enemigos para llegar a la salida, y que les enfrenten. También se podrá encontrar abundante munición como recompensa en algunas salas secretas de los mapas para favorecer la exploración.

Personajes y enemigos

Doomguy



El único personaje jugable será Doomguy, soldado de la Marina y único superviviente del escuadrón. Su misión consistía en defender las naves mientras el resto del equipo entraba al centro y cerraba el portal que conlleva al inframundo, pero todo su equipo ha fallado en su misión, y ahora, su vida todo el destino de la humanidad dependen de él.



Zombi



Marine que ha sido convertido en zombi al morir a causa de un demonio, tanto al ser teletransportado con un portal como en uno de los sucesivos intentos de establecer contacto de nuevo con la luna. Llevan un uniforme marrón básico manchado de sangre y usan un fusil de asalto para disparar. Como rasgo característico, sus ojos rojos demuestran su naturaleza de zombi, lo que los diferencian de su forma humana. Es el enemigo más débil del juego.

Zombi Sargento



Enemigo parecido al zombi, solo que estos ocupaban un rango superior a los soldados habituales. Es por ello que su armadura es negra y usan una escopeta como arma de fuego para atacar en lugar de un rifle de asalto. Tienen un daño y vida superior a los del zombi básico.

Diablillo



Demonio básico salido del portal de teletransportación. Tiene forma humanoide de color marrón con pinchos saliendo de su cuerpo. Aunque sea el más débil de los demonios, tiene una resistencia superior a los zombis, pero carece de velocidad. Para atacar a distancia puede usar una bola de fuego a distancia que se mueve lentamente, pero hace una cantidad de daño significativo. Por otra parte, si el jugador se acerca lo suficiente a él, el diablillo podrá usar sus garras para atacar.

Demonio



Monstruo rojizo, con una gran musculatura, cuernos y dientes afilados. Puede recorrer largas distancias en cuestión de breves lapsos de tiempo. Es por eso que está especializado en el combate cuerpo a cuerpo, tanto dando embestidas como mordiscos al jugador. Puede llegar a ser una gran amenaza si llega a encontrarse al lado del jugador y le cierra el paso.

Alma perdida



Tienen la apariencia de una calavera en llamas. Son sigilosas y se suelen mover en grupo. Su forma de ataque consiste en realizar embestidas al jugador desde lo alto, en donde se ocultaban hasta realizar el ataque. Tienen escasa vitalidad, por lo tanto, una sola alma perdida no llega a ser un gran problema, pero en grupo pueden llegar a ser mortales.



Cacodemonio



Gigante esfera roja con pinchos grises, de los que sobresalen 2 que se puede llegar a entender que son los cuernos. Tiene un ojo verde entre los cuernos y de entre la boca se dejan aparecer múltiples dientes sucios y una lengua azul. Siempre guardan la distancia con respecto al jugador, prefieren luchar a distancia y lanzar bolas de plasma por la boca. Estas, como las bolas de fuego del diablillo, se mueven lentamente hacia el jugador, explotando cuando golpean, e infringiendo una considerable cantidad de daño al usuario si le golpea directamente.

Armas

El protagonista contará con dos armas iniciales, el puño y la pistola, y luego existirán diversas armas a lo largo de los mapas que podrá recoger para aumentar su arsenal destructivo. Entre ellas están las siguientes:

Puños.

Ataque cuerpo a cuerpo básico del personaje. Pega un puñetazo con la mano izquierda que infringe daño al enemigo. Cuando el usuario se quede sin balas, siempre puede recurrir a esta arma para abrirse paso entre los enemigos.

Motosierra.



Ataque cuerpo a cuerpo alternativo del puño con más cadencia de fuego, pero un daño por golpe reducido, aunque el balance de daño por segundo sea mayor. Será un arma recogible, ubicada en el mapa. Más cómoda de usar que el puño, pues se podrá mantener el clic para ataques continuos.

Pistola sencilla.



Es el arma con la que se empieza el juego. Destaca por tener mucha munición, aunque por ello tiene un daño medio y una baja cadencia de fuego, teniendo que pulsar el botón cada vez que se quiera realizar un disparo con ella.



Escopeta.

Escopeta de doble cañón, la cual ofrece un gran daño a corta distancia, pero a medida que el enemigo se aleja del jugador, su daño disminuye considerablemente. Al igual que la pistola, requiere pulsar el botón cada vez que se quiera ejecutar un disparo.



Ametralladora rotatoria.



Arma automática de tiro rápido. Cuenta con un cañón rotatorio y la mayor cadencia de fuego con respecto a todas las demás armas, pero un daño por bala menor que el de una pistola. Comparte tipo de munición con la pistola. Será decisión del jugador el usar este tipo de munición con la pistola o la ametralladora, derrochando munición, pero ganando en daño realizado por segundo.

BFG9000

Comúnmente llamada “Big Fucking Gun 9000", esta gran arma de metal sólido brillante dispara un gran orbe de energía verde que avanza lentamente hasta chocar con una criatura o una pared, generando una gran explosión que puede dañar al jugador si se encuentra en las cercanías. Necesita cargar antes de disparar, pero su poder destructivo es superior con diferencia a cualquier arma.

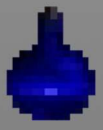


Consumibles

Durante el juego, el jugador se encontrará distribuidos por el mapa diversos consumibles para aumentar la vida y la armadura del jugador, o para obtener todo tipo de ventajas. Además, algunos enemigos también pueden soltar objetos de este tipo al morir.

Poción de salud

Pequeña poción de color azul. Recoger este consumible aumenta la salud del jugador en un 1%, hasta un máximo de 200%.



Paquete de salud



Pequeña caja blanca con una cruz roja. Recoger este consumible aumenta la salud del jugador en un 10%, hasta un máximo del 100%.



Botiquín

Gran caja blanca con una cruz roja y con distintos bolsillos. Recoger este consumible aumenta la salud del jugador en un 25%, hasta un máximo de 100%.



Casco



Pequeño casco que ofrece un bonus de armadura pequeño al jugador. Recoger este consumible aumenta la armadura del jugador en un 1%, hasta un máximo de 200%.

Armadura

Armadura de combate verde que ofrece un bonus de armadura al jugador. Recoger este consumible establece la armadura del jugador al 100%.



Mega armadura



Rara armadura de combate azul que ofrece un bonus de armadura al jugador. Recoger este consumible establece la armadura del jugador al 200%.

Balas

Cargador de balas con forma de clip. Recoger este consumible aumentará la munición del tipo balas, para usarlas con la pistola o la ametralladora.



Cartuchos



Cargador de balas con forma de Cartucho. Recoger este consumible aumentará la munición del tipo cartucho, para usarlas con la escopeta.

Audio y musica

El juego tendrá una música ambiental de acción para incitar al usuario a seguir avanzando mientras dispara. Además, acompañando a esta, todas las acciones que realice el jugador tendrán efectos sonoros extras.

Las armas contarán con un sonido de disparo característico de cada una cuando son disparadas.

Los enemigos tendrán un sonido cuando el jugador entre en su rango de visión y sean provocados. De entre ellos se pueden destacar el sonido de radio para los zombis, o los gruñidos para los diablillos.



De igual forma que con las armas del usuario, cuando los enemigos atacan realizarán un sonido característico según con que hayan atacado, pudiendo ser una rasgadura cuando atacan con garras, un disparo si dispara un zombi, el sonido de generación de fuego para las bolas de fuego, etc...

Cuando los enemigos son derrotados, generarán un efecto de angustia, para así ayudar al usuario a percatarse de ello y poder centrarse en el resto de los enemigos de la sala actual.

Los consumibles y las armas recogidas tendrán un efecto sonoro cuando son recogidos, para alertar al usuario de que el objeto que había en el suelo ha sido recogido. Además, como se menciona más adelante, se mostrará por pantalla un texto que informe de ello al jugador.

Arte y animacion

Se diseñarán gráficos y animaciones para todas las acciones que se han descrito en este documento. Cada arma contará con un Sprite cuando está en el suelo para recoger y con otro diferente cuando la está usando el usuario, centrada en su vista en 1º persona. La idea general es demostrar un ambiente tenso en donde el jugador esté siempre alerta y en movimiento, intentando que los enemigos parezcan vivos con todo tipo de movimientos.

Todos los consumibles se diseñarán según las descripciones dadas de su forma y color para ser incluidas en los mapas Se diferenciarán del entorno por tener un color más brillante.

Los enemigos tendrán animaciones para las acciones de moverse, disparar y morirse. Esto quiere decir que mientras patrullan por el mapa estarán andando. Cuando disparen al jugador ejecutarán la animación de disparo y cuando el jugador acabe con ellos, se creará una animación de muerte del demonio correspondiente. Cada uno, debido a sus distintas formas físicas, tendrán distintas animaciones.

También se generarán distintos efectos típicos de los shooters como los destellos de las armas al disparar, los movimientos de las armas debido al retroceso, etc...

GUI

El juego contará con una GUI simple que ocupará toda la parte inferior de la pantalla. En ella, se mostrará el estado completo del jugador. En un fondo gris aparecerá en números rojos para que destaque la munición restante del arma en uso, los porcentajes de vida y armadura restante del jugador y la lista de armas disponible que ha ido recogiendo a lo largo de todos los niveles.

En la versión original se puede ajustar el tamaño de la pantalla con unas barras verdes que ocuparán el espacio que no ocupe la ventana principal de juego. El diseño de las páginas de este GDD está inspirado directamente en la interfaz del videojuego.





También aparece información sobre algunas acciones del jugador de forma escrita en la parte superior izquierda de la pantalla cuando interactúa con ciertos objetos, por ejemplo, para informar de que ha recogido algún objeto o de que necesita una llave para abrir una puerta determinada.

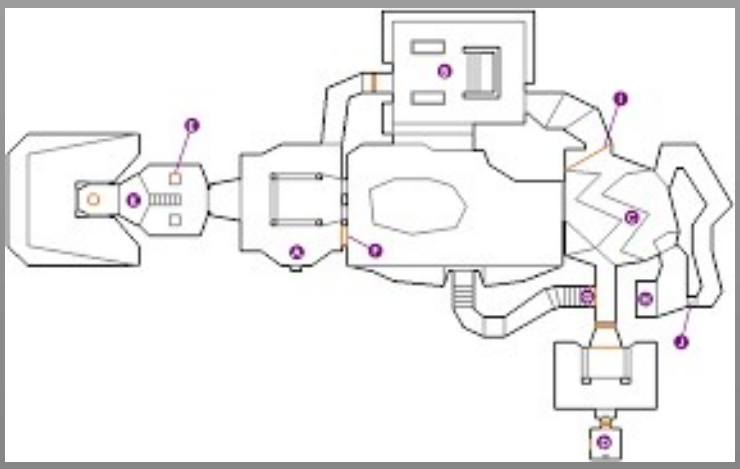
Niveles

El diseño de niveles estará centrado para ofrecerle al usuario una experiencia de acción desenfrenada, en donde pueda ir avanzando por distintas salas interconectadas disparando a todos los demonios que hay en ellas sin descanso.

Por los distintos niveles estarán distribuidos todo tipo de consumibles para ayudar al usuario en la jugabilidad, pero, los más poderosos, como pueden ser nuevas armas, botiquines o mega armaduras, estarán escondidos en secretos dentro de las salas. Estos secretos pueden ser de tipo pared invisible con marcas que se puede atravesar, pequeño botón activable en una pared que abre una sala, o simplemente un ascensor escondido.

Es decisión del jugador el pararse por las salas a buscar estos consumibles y así armarse con más vida y armadura o, por el contrario, decide avanzar por las salas rápidamente con el equipamiento actual.

En cuanto a la variedad de niveles que se ofrecerá a los jugadores en esta versión de Doom, aún está por determinar si se añadirán niveles directamente extraídos del original, recreaciones de estos adaptadas a la nueva versión, o niveles completamente nuevos. Probablemente se optará por una mezcla de las dos últimas, intentando recrear con algunos cambios unos pocos niveles originales y uno o dos niveles de elaboración propia, a modo de demostración para la asignatura.



Esta imagen es un ejemplo del primer nivel del juego original, el cual se tratará de recrear en esta nueva versión de Doom. Como se puede ver, presenta varias salas con desniveles y obstáculos, como la sección en zigzag, para poner a prueba al jugador a la vez que le introduce a la mayoría de conceptos y mecánicas que se presentan en los niveles del juego casi a modo de tutorial.

