Proyecto Final

First Person Shooter

UNIVERSIDAD DE JAÉN

Informática Gráfica y Visualización

Jesús Morales Villegas	---	jmv00037	---	77770715X
Samuel Manzano Álvarez	---	svg00018	---	20620087N

Índice

1.Introducción

El proyecto realizado es un videojuego de disparos tridimensional en primera persona o FPS (del inglés, first-person shooter) es un género de videojuegos que simula el uso de armas de fuego desde una perspectiva de primera persona.
En un principio teníamos en mente basarnos en un conocido videojuego y precursor de este género el Wolfenstein 3D.
El videojuego cuenta con un nivel que tiene tres salas, en cual aparecemos en la del medio en el cual podemos activar o desactivar la iluminación de la sala y después nos encontraremos con dos puertas que tendremos que abrir, cada una en un extremo de la sala en el cual encontraremos un robot dando vueltas al cual podremos disparar y derrotar.
Si desea apuntar mejor el videojuego cuenta con un zoom.

2.Diseño

2.1. Mapa

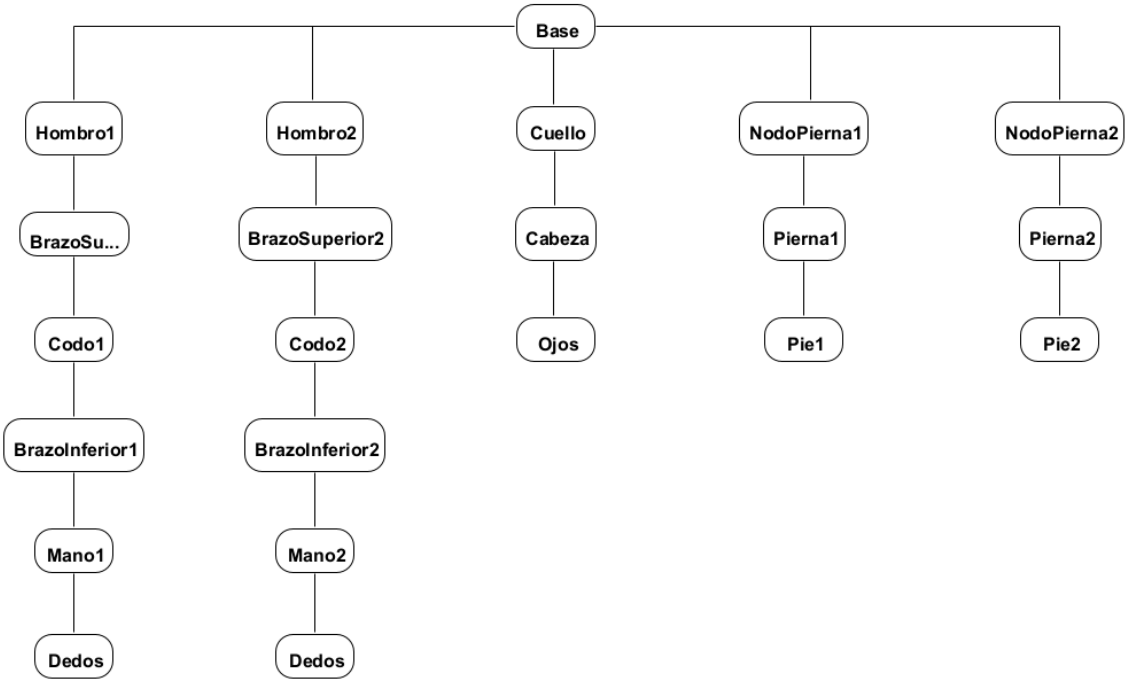
Para definir el diseño del mapa tenemos que separarlo en varias partes:

- Suelo y techo: están creados a partir de un cubo de la clase cubo, generado mediante una malla de QUADS al cual se le da las transformaciones necesarias como por ejemplo se escala para que sea un cuadrilátero de 50 x 50 y de alto 1, con su correspondiente traslación.
- Paredes exteriores: también están creados a partir de un cubo de la clase cubo, en el cual se generan 4 paredes exteriores para poder cerrar todo el mapa con unas dimensiones de largo 50, alto 4 y ancho 1, con su correspondiente traslación.
- Paredes interiores: también están creados a partir de un cubo de la clase cubo, en el cual se generan 4 paredes interiores para poder crear las salas del mapa con unas dimensiones de largo 25, alto 4 y ancho 1, con su correspondiente traslación.
- Puertas:

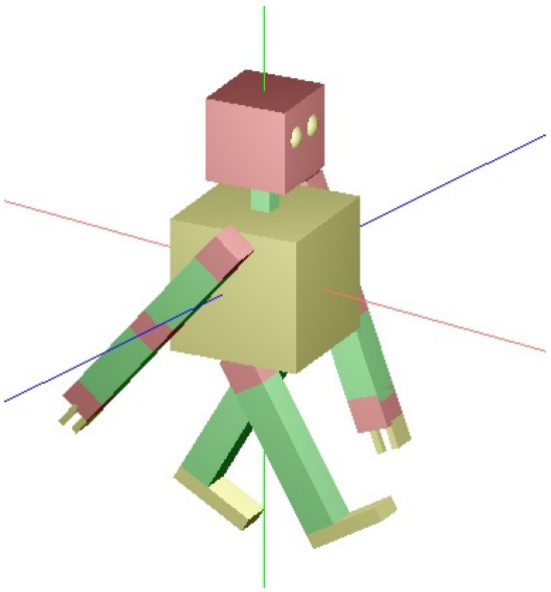
2.2. Robot

Diseño: el modelo se ha diseñado utilizando primitivas geométricas de OpenGL.
Hemos introducido el robot ya que se nos pedía que crearamos en el proyecto uno de los modelos hecho en la práctica 3B. El robot se crea en la clase robot se mueve de lugar, para tener la sensación de que se está moviendo le hemos añadido una animación de mover las piernas y las manos.

- Representación mediante arbol:


Base
Hombro1
Hombro2
Cuello
NodoPierna1
NodoPierna2
BrazoSu...
BrazoSuperior2
Cabeza
Pierna1
Pierna2
Codo1
Codo2
Ojos
Pie1
Pie2
BrazoInferior1
BrazoInferior2
Mano1
Mano2
Dedos
Dedos

- Resultado



2.3. Clases

2.3.1. Cubo

Especificación e implementación de la clase cubo, que contiene la funcionalidad de crearlo y darle textura.

2.3.2. igvCamara

Especificación e implementación de la clase igvCamara, que contiene la funcionalidad básica para crear y manipular cámaras de visión en la aplicación, también tiene otras funcionalidades añadidas por nosotros como el zoom o mirar.

2.3.3. igvColor

Especificación e implementación de la clase igvColor, que contiene la funcionalidad básica para definir y utilizar objetos de tipo color.

2.3.4. igvEscena3D

Especificación e implementación de la clase igvEscena3D, que contiene la funcionalidad básica para visualizar la escena, además tiene otras funciones como creación y eliminación de balas, dirección de las balas, creación de los robots, la iluminación, las puertas y los modelos.

2.3.5. igvFuenteLuz

Especificación e implementación de la clase igvFuenteLuz, que contiene la funcionalidad básica para definir y utilizar luces puntuales y tipo foco.

2.3.6. igvInterfaz

Especificación e implementación de la clase igvInterfaz, que contiene la funcionalidad básica para crear una ventana de visualización, su configuración y la gestión de los eventos del sistema. Cuando creamos la configuración del entorno, se muestra el programa en pantalla completa con una resolución de 1920 x 1080 y se activa el modo juego, además se cambia el cursor a una cruz.

2.3.7. igvPunto3D

Especificación e implementación de la clase igvPunto3D, que contiene la funcionalidad para declarar y utilizar objetos de tipo punto y vector.

2.3.8. puerta

Especificación e implementación de la clase puerta, que contiene la funcionalidad básica para crear una puerta y pintarla en la escena.

2.3.9. robot

Especificación e implementación de la clase robot, que contiene la funcionalidad para declararlo, moverse hacia delante, moverse hacia atrás y girar.

2.3.10. utils

Especificación e implementación de la clase utils, que contiene las funciones para cargar los ficheros .obj y cargar las imágenes de las texturas.

2.4. Teclas

Todas las teclas funcionan solo en minúscula.

- w: Se desplaza hacia adelante.
- a: Se desplaza hacia la izquierda.
- s: Se desplaza hacia atrás.
- d: Se desplaza hacia la derecha.
- z: Activa/Desactiva el zoom.
- q: Disparo.
- e: Enciende/Apaga la luz.
- Esc: Termina la ejecución.

3.Implementación

3.1. Sonido

Para implementar el sonido hacemos uso de la librería <mmsystem.h>, la cual tiene una función llamada “PlaySound” usada en el main para que inicie la música nada más empezar la ejecución del programa.

3.1.1 Problemas y Soluciones

- El primer problema que ocurre al usar “PlaySound()”, es que si no le específicas que quieres el sonido de forma asíncrona, el sistema detiene el programa y solo se escucha el audio. Para solucionar esto simplemente se añade como parámetro SND_ASYNC.

- El siguiente problema ocurre cuando queremos añadir un sonido más, al disparar el arma, al ser muy básica “PlaySound()” solo permite ejecutar un solo audio. Para solucionar este problema tendríamos que añadir una API que se encargase de mezclar varios audios al mismo tiempo. Por tanto, hemos decidido no añadir más audios.

- Por último “PlaySound()” no nos permite regular el volumen de forma sencilla y se escucha bastante alto

3.2. Texturas

Las texturas están almacenadas en formato .bmp, se utiliza la función definida en utils con el nombre de “LoadTexture()”. Dentro de la función se lee el fichero y se guardan en arrays para su posterior uso, así evitamos que se lea desde fichero cada vez que se aplican las texturas y por consiguiente optimizando el uso de texturas.

Las texturas se almacenan en la clase cubo, donde también se genera un cubo para posteriormente aplicar las texturas correspondientes.

3.2.1. Problemas y Soluciones

- Al crear el mapa nos encontramos con un problema visual, porque si escalamos el cubo para que ocupe 50x50 en el eje x,z, la textura también se escala. Al escalar la textura apenas se distingue de la original ya que se estira mucho y no es estética. Para solucionar esto se crean dos parámetros en el método “dibujar()” de la clase cubo, repetirX y repetirY, los cuales permiten repetir la imagen X veces en lo alto, Y

veces a lo ancho del cubo. Entonces a la hora de escalar el cubo se repite el patrón como se puede apreciar en el resultado final.

- Como se usa el mismo cubo para crear objetos con texturas, tenemos problemas para crear un robot con textura más detallada, por eso hemos optado por solo añadir textura en el cubo central del robot y dejar el resto de partes con colores de opengl.

- Como se ha comentado anteriormente, leer el fichero de textura cada vez que se aplica la textura mermaba el rendimiento del programa, por eso decidimos almacenar las texturas.

3.3. Iluminación

En la implementación de la iluminación hemos realizado lo siguiente, hemos quitado toda la iluminación ambiente y hemos dejado una única luz puntual en el centro de la sala del medio, el cual se puede activar y desactivar pulsando la tecla ‘e’.

3.3.1. Problemas y Soluciones

- Queríamos hacer un foco de luz que saliera del personaje y apuntar hacia donde estuviéramos mirando, pero al introducir el foco de luz no se veía, por lo tanto optamos por crear una luz puntual que sirviera como si simulara una luz en el techo.

3.4. Creación del Mapa

Para crear el mapa se usan en total 12 cubos con texturas (1 para el techo, 1 para suelo, 4 para las paredes exteriores, 4 para las paredes interiores y 2 para cada puerta).

3.4.1. Problemas y Soluciones

- Debido al problema de texturas comentado en el punto 3.2.1 pensamos en hacer varios cubos suelo, techo, paredes, etc. El mayor inconveniente de esta idea es el rendimiento, por eso la tuvimos que descartar y hacer lo mencionado en el punto 3.2.1.

- Existe un bug visual con las texturas de las puertas, la textura está un poco desplazada hacia un lado, no hemos conseguido solucionar el bug.

3.5. Robot

La implementación del robot se encuentra en la clase robot, en la clase igvEscena3D se implementa el diseño y en la clase igvInterfaz se gestiona los movimientos y animaciones. La animación y diseño son las mismas que en la práctica 3b.

3.5.1. Problemas y Soluciones

- Para implementar el movimiento del robot simplemente se cuenta el número de veces que se llama a la función “set_glutIdleFunc()” de la clase igvInterfaz, esta función tiene el código para el movimiento y animación del robot. Cuando llega a cierta cantidad de veces el robot gira y sigue desplazándose, así en bucle.

3.6. Cámara

Este punto es quizás el más difícil o tedioso ya que hay que realizar bastantes multiplicaciones de vectores, normalizar vectores, etc. Para poder conseguir una cámara en primera persona.

La mayor parte de información la he encontrado buscando en internet, por eso me ha parecido la más difícil de implementar.

En la clase igvCamara se gestionan estos cálculos, en el método ”onKeyBoard()” se calcula el movimiento de la cámara en el eje X y en el eje Z. Para el movimiento por ratón se usa el método “mirar()”

El zoom implementado es el de la práctica 2b, con el único cambio de que solo hace zoom una vez.

3.6.1. Problemas y Soluciones

- En la cámara queríamos limitar el movimiento en el eje Y, para no poder realizar un giro de 360 grados, para ello se comprueba si el ángulo de giro es mayor a cierto límite, si es mayor se mantiene la configuración de la cámara anterior, porque si no se mantiene la anterior los parámetros de la cámara seguirán cambiando, aunque en la pantalla no se notase el cambio.

- Como no queremos que se mueva en en eje Y al moverse con las teclas, se guarda la posición de la cámara en el eje Y antes de realizar los cálculos, para después de realizar los cálculos volver a los parámetros anteriores.

3.7. Interacción con los objetos

Para tener la interacción con los objetos hemos implementado dos formas

- Para las puertas lo hemos implementado mediant un buffer de color en el cual asignamos un color a un objeto por lo tanto cuando hacemos click derecho redibujamos los objetos del buffer y si coinde pues aplicamos las funciones correspondientes en este caso moverPuerta para que la puerta se abra.
- Para las colisiones de la bala contra el robot, el robot tiene una caja envolvente, por lo tanto se calcula si la bala está dentro de esa caja envolvente, si está llama a la función muerto del robot para que no se muestre.

3.8. Carga y dibujo de obj

Para cargar se usa la clase utils, el método “cargaOBJ()”.

3.8.1. Problemas y Soluciones

- Para cargar los modelos hay especificar a opengl que vas ha usar array de vértices, array de normales y array con la malla de triángulos. Esto se ve reflejado en la clase igvEscena3D en el método “visualizarPartes()”.

- El mayor problema de está técnica son las texturas, aplicar textura a un modelo complejo es altamente complejo y no hemos conseguido aplicar una textura. Por eso la pistola no tiene textura.

3.9. Pistola

El modelo de la pistola se carga desde el fichero obj, y se almacena para no tener que estar leyendo el fichero varias veces.

Después se dibuja en la escena y se ajusta hasta tener cierto tamaño y se rota para que mire hacia donde apunta la cámara, el mayor problema de la pistola viene al tener que ser “estática” respecto a la cámara, para ello se tiene que mover y rotar a la misma velocidad y sentido que la cámara.

El movimiento es fácil, ya que las coordenadas de la pistola son las mismas que la de la cámara, pero desplazada un poco hacia la derecha. Se complica un poco más a la hora de rotarla a la misma vez que la cámara.

Para rotar se obtiene la rotación que ha hecho la cámara en ese instante y se le pasa a la pistola y a la hora de dibujarla se desplaza al origen, después rota y por último vuelve a su posición anterior.

3.9.1.Problemas y Soluciones

- Solo hemos implementado la rotación de la pistola en el eje Y, ya que es la más fácil de implementar, por tanto solo rota al mover la la cámara hacia los lados.

3.10. Balas

Las balas son una esfera que recorre el espacio de forma lineal y con una velocidad constante. Se destruyen si recorren cierta distancia y están almacenadas en un vector de parejas, donde el primer valor es la posición en el espacio y el segundo valor la velocidad. También existen dos vectores más relacionados, que almacenan la dirección de la bala y la cantidad de espacio recorrido.

3.10.1.Problemas y Soluciones

- El principal problema a la hora de crear la bala es la dirección en la que tiene que ir. Se calcula en la clase igvInterfaz, cuando se presiona la tecla q. La dirección se calcula como vector posición de la cámara menos vector target de la cámara y el resultado normalizar (consultado en el enlace relacionado con la cámara de la bibliografía).

- Con la dirección obtenida simplemente se calcula el movimiento en el siguiente instante, para después moverla.

3.11. Fluidez

Para que el movimiento de cámara sea fluido se usa en la clase igvInterfaz el método “loop()”, en el cual se hacen varias cosas.

En primer lugar, se mantiene el cursor en el centro de la pantalla usando “glutWarpPointer()”.

Después se obtiene el tiempo que tarda opengl en llamar a la función “glutInit()”

En la tercera línea se calcula la diferencia de tiempo que ha tardado en llamar a “glutInit()”, entre el fotograma anterior y el actual y se almacena en dt para su posterior uso.

Y por último indicamos a opengl que vuelva a llamar a “loop()” en un tiempo determinado.

4.Conclusiones

Como conclusión, hemos estado de acuerdo en que gracias a las anteriores prácticas, muchas de las cosas que se nos pedían de requisitos se han hecho más amenas de hacer, por otro lado hemos tenido bastantes inconvenientes que la gran mayoría hemos podido solucionar o si no lo hemos abordado de otra manera como la iluminación, con un poco más de tiempo podríamos haberla creado como en nuestra idea principal.
Aparte de los requisitos hemos querido añadir más cosas como música a nuestro videojuego, cargar modelos de malla 3D (.obj) y hacer un ejecutable (.exe) para poder jugar directamente.

5.GitHub

Para la realización del proyecto hemos usado gitHub para que la administración del código sea más fluida entre nosotros.

Dentro de gitHub se puede descargar un fichero zip con el ejecutable, sin necesidad de tener que descargar y compilar el código. El archivo se encuentra en el apartado de releases y se llama proyectoFinal.zip. También se puede descargar donde estan los archivos .cpp y .h, con el mismo nombre.

Lo único que hay que hacer para poder ejecutar el programa es descomprimir el zip y dirigirse a la carpeta ejecutable y ejecutar el archivo proyectoFinal.exe

- https://github.com/jmv00037/proyecto_final

6.Video

https://drive.google.com/file/d/1ovq36bV9TvQ-rX6XCQ8nbhbc1JJiEPor/view?usp=sharing

7.Bibliografía

- https://prezi.com/view/JMduuB07dUqPEzirytSr/
- https://www.turbosquid.com/es/
- http://www.opengl-tutorial.org/
- https://www.opengl.org/resources/libraries/glut/spec3/node1.html
- https://stackoverflow.com/
- https://learnopengl.com/Getting-started/Camera