BENEMERITA UNIVERSIDAD
AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

Graficación

Parcial 2

Gálaga 3D (Proyecto 3D)

FUENTES GALLARDO CRISTIAN
GARCÍA GONGORA ALAN
PANTLE MACUIL FERNANDO

PROF. IVÁN OLMOS PINEDA

Puebla, Pue. Diciembre 2 del 2023

RESUMEN DEL PROBLEMA

En este caso se llevara a cabo una re-imaginación del clásico videojuego “Gálaga”, ahora pasado a un ambiente en tercera dimensión. En este videojuego, manejamos una nave espacial, la cual tiene la capacidad de volar por el plano y disparar láseres para eliminar las naves enemigas que tiene enfrente. Si esta nave choca 3 veces con las naves enemigas, se producirá un “game over” y el juego acabara. Como se menciona, ahora todo esta mecánica del juego pasara al 3D, es decir, se podrá manejar la nave en tres dimensiones y las naves enemigas vendrán igualmente en tres dimensiones.

El FPV no tiene implementado su respectiva colisión con los asteroides debido a la naturaleza del videojuego, pues la cámara siempre esta detrás de la nave.

No se logro hacer funcionar correctamente la iluminación, por lo que el renderizado de los modelos 3D se vera con un aspecto plasticoso.

INTRODUCCIÓN

Para poder llevar a cabo este proyecto, se requirieron de conocimientos en el manejo de python y en el uso de pygame y OpengGL. Ademas, se implementaron funcionalidades para:

- Renderización de modelos Wavefront
- Texturización de objetos gráficos
- Funcion de detección de colisiones
- Manejo del FPV (First Person View)
- Manejo de la nave y el FPV mediante el teclado

DESARROLLO

Implementación de objetos en escena (modelos 3D)

Para implementar nuestros modelos 3D, utilizamos una clase “*objloader*”, la cual lee la información tridimensional del objeto 3D en cuestión (datos de vértices, datos de caras, coordenadas de textura, normales, materiales y propiedades) y nos permite renderizarlos en nuestra escena por medio de dos funciones, la funcion *generate* (genera el objeto 3D) y *render* (renderiza el objeto en la escena).

Estos archivos se referencian por medio de la ruta relativa donde se encuentre ubicado el archivo.obj, al igual que sus materiales (archivo.mtl). Es importante que estos dos archivos estén en la carpeta del proyecto, de esta forma, el acceso a estos es mas cómodo, y sobre todo, no causara ningún problema en relación con la búsqueda de las rutas de los modelos.

Para pasarlos a escena, definimos una clase para cada uno de los modelos que se ocuparan (Nave, Escenario, Asteroide), cada una con sus respectivas variables de control. En el constructor de cada clase, se crean los objetos de tipo OBJ (clase de *objloader.py*) respectivos de cada modelo, donde en sus parámetros se referencian las rutas relativas que se mencionaron anteriormente.

Se define una función para generar los modelos 3D llamada "*Generate()" (exceptuando a la clase Asteroide, en esta no es necesario puesto que se llamo esta función en su constructor)*, y en otra función llamada "*Draw()"* se dibuja el objeto en cuestión.

Esta funciona de la siguiente manera:

1. Primero se hace un "*glPushMatrix()"* para guardar la matriz de modelado en un pila de matrices propia de OpenGL.
2. Después se hacen las trasformaciones respectivas según el objeto. Por ejemplo, en el caso de la nave, la trasladamos según sus propiedades tx, ty y tz ; la escalamos a un tamaño 1x1x1; y la rotamos según sus variables rx y rz.
3. Ahora, después de las transformaciones, renderizamos la nave por medio de la función "*render()".*
4. Finalmente, restauramos la matriz de modelado con un *"glPopMatrix()".*

Cabe destacar que las clases Nave y Asteroide funcionan de manera única a comparación del escenario (la cual funciona tal y como se dicto anteriormente):

La clase Nave genera dos modelos, uno para la nave y otro para sus láseres. Las variables que utiliza la nave son tx, ty, tz, rx y rz; las primeras tres manejan la posición en tres dimensiones en la cual estará ubicada este objeto (las cuales irán cambiando según el movimiento dado por el teclado), y las ultimas dos definen la rotación de este mismo tomando de pivotes los ejes X y Z (según los inputs del teclado la nave se inclinara un poco, tomando en cuenta el lado al que se este moviendo, dando una sensación mas realista de movimiento).

En el caso de los láseres, se hizo una variable "*lz*", la cual maneja la distancia a la cual se moverá el láser en el eje Z. No es necesario definir otras variables para los ejes X y Y, pues estos ocupan las mismas variables X y Y de la nave espacial. Ademas, se hizo una nueva función que dibuja estos proyectiles; funciona de manera idéntica al proceso de dibujado que se definió al principio, solo con el agregado que en la función "*Draw()*" al final del dibujado de la nave, en seguida se dibujan los láseres.

La clase Asteroide por otra parte, se define por las variables de dimensión del plano (DimBoardX Y DimBoardY); y por dos arreglos de tres valores (Position[] y Direction[]), los cuales tendrán un valor aleatorio.

Este arreglo "*Position*" ira actualizándose bajo ciertas condiciones: si el objeto esta dentro del escenario, este se moverá según la nueva dirección resultante de la suma de la posición en Z mas la dirección en Z; en caso contrario, a este se le reiniciara su posición, como si volviese a generarse.

A este ultimo arreglo "*Direction*" se le normalizara con respecto al eje Z, para así definir correctamente la dirección a la que ira.

Finalmente se utiliza la función "*Draw()*", la cual hace uso de las variables anteriormente mencionadas.

Todos estos procesos anteriormente mencionados se ejecutan dentro de una función "*display*" (ubicada dentro de la condicional de accionamiento de teclas), la cual llama las funciones *Draw()* de todos los objetos 3D.

El el caso de los asteroides, se llaman todos los *Draw()* de cada uno de los asteroides en conjunto con su función *Update()*. Otro detalle que vale la pena explicar es que debido al peso del escenario *moon_surface* (11 MB) y claro, combinado con todo el conjunto de modelos, el programa tardara un poco en iniciarse.

Implementación de texturas en objetos gráficos

Para cargar las texturas de nuestro fondo, hicimos una clase "*Espacio*" en la cual cargamos las texturas (imágenes bmp) en un arreglo llamado "*textures*" en el constructor.

Ahora, la función "*load_texture*" carga la textura de una imagen, la volteamos (debido a que Pygame y OpenGL tienen direcciones de textura invertidas) y convierte la imagen en una cadena de bytes. Se procede a darle una ID a la textura y se enlaza al objeto gráfico; se definen los parámetros de esta misma y se sube a OpenGL.

Para dibujar nuestro fondo, se hizo la función "*DrawSpace*", la cual inicializa las coordenadas de los vértices, los indices y las texturas para una cara del cubo que conformaran el fondo. Configuramos la escala y la posición de esta pared mediante transformaciones; se habilitan el uso de texturas, arreglos de vértices y coordenadas de textura; se configuran los punteros de los vértices y texturas para finalmente seleccionar el ID de la textura y dibujarla. Una vez dibujada, se deshabilitan las texturas y los arreglos de vértices y, coordenadas de textura.

Para cargar y dibujar las caras del cubo, primero se crean los objetos *espacio# dentro del Init()* del programa, donde a cada uno se le esta pasando la ruta de las imágenes bmp correspondientes. Después pasamos a dibujarlas dentro de la función "*display()*", donde cada cara tiene ciertas transformaciones para ubicarlas correctamente en la escena, formando así cinco paredes, o bien, un cubo (la cara inferior no es necesaria implementarla).

Implementación de detección de colisiones

Utilizamos una función "*checar_colisiones()*" (es llamada dentro de la condicional de lectura de inputs de teclado), la cual toma como parámetros al objeto *Nave,* al arreglo de asteroides y una variable booleana llamada *choque,* mas adelante se explicara para que sirve específicamente. Dentro de esta función, se iteraran entre todos los asteroides con la nave y los láseres. Para detectar las colisiones entre nuestros objetos, seguimos el siguiente proceso:

1. Debemos calcular la distancia entre nuestros objetos, la cual se saca mediante la formula de la distancia euclidiana (esta se deduce a partir del teorema de Pitágoras), la cual es:

   $$distancia = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

   donde [x1,y1,z1] es el arreglo de coordenadas del primer objeto, y [x2,y2,z2] del segundo objeto.

2. Ahora, se toma en cuenta el radio de cada uno de estos mismos, desde su centro hasta el punto mas lejano de su contorno; ahora sumaremos los dos radios, dándonos así la distancia en la cual estos dos objetos se tocan.

3. Hacemos la condicional de detección: Si la distancia entre estos objetos es menor a la suma de sus radios, entonces se ejecutara cierta acción.

   En el caso de la nave, el asteroide que se estrella en cuestión, se le reiniciara su posición (como se explico en **La clase Asteroide**), y devolverá la variable *choque* como True. Así, mediante un procedimiento fuera de esta función "*checar_colisiones()*" se leerá esta variable booleana, el cual si resulta ser True, se le restara 1 al contador de vidas. Si este contador llega a cero, la variable *game_over* se volverá True, y por ende detendrá la ejecución del programa (mas adelante se explica donde se utiliza esta variable)

4. **Respecto a los láseres**, cuando ocurre una colisión, el asteroide determinado se reiniciara y el láser también se reiniciara en su posición original, la cual es dentro de la nave.

Implementación del FPV y el control por teclado

El First Person View y el movimiento de la nave se modifican mediante los inputs de teclado que se reciban. Hay dos tipos de inputs: inputs de un solo tecleo, e inputs de tecleo presionado.

Para habilitar la lectura de inputs de un solo tecleo, generamos un ciclo de *e* a *pygame.event.get()*, y después generamos la condicional *e.type == KEYDOWN*, la cual lee los inputs. Ahora solo queda hacer condicionales extra para cada tecla que se desee que tenga una funcionalidad. En este caso, se implemento una tecla "*p*" que devuelva el valor de *pausado* como *not pausado*, la cual si esta activa, la ejecución se pausara mediante la condicional que esta antes de entrar a la lectura de teclas presionadas, donde también se toma en cuenta el valor booleano de la variable *game_over*, la cual se menciono su funcionamiento en el sub-tema anterior.

Para habilitar la lectura de inputs de tecleo presionado, se genera un objeto *keys* del tipo pygame.key.get_pressed(), la cual permite que si se mantienen presionadas las teclas determinadas, la acción que ejecutan no se detendrá, a mas que se deje de presionar la tecla, claro esta.

En este caso, se implementaron cuatro teclas:W, A, S, y D que manejaran el movimiento hacia arriba, izquierda, abajo y derecha respectivamente. La estructura de cada función es la siguiente:

- Tecla W.- si la nave en su coordenada *Y* está dentro de la dimensión del escenario (dimy), entonces se le sumara 0.8 a la coordenada en *Y* de la nave, 0.45 al eje de rotación con pivote en *X*, y se sumaran 0.7 y 0.8 a las coordenadas en *Y* de la posición y la dirección de apuntado de la cámara respectivamente.

- Tecla A.- si la nave en su coordenada *X* está dentro de la dimensión del escenario (-dimx), entonces se le restara 0.8 a la coordenada en *X* de la nave, se sumara 0.45 al eje de rotación con pivote en *Z*, y se restaran 0.7 y 0.8 a las coordenadas en *X* de la posición y la dirección de apuntado de la cámara respectivamente.

- Tecla S.- si la nave en su coordenada *Y* está dentro de la dimensión del escenario (-dimy), entonces se le restara 0.8 a la coordenada en *Y* de la nave, 0.45 al eje de rotación con pivote en *X*, y se restaran 0.7 y 0.8 a las coordenadas en *Y* de la posición y la dirección de apuntado de la cámara respectivamente.

- Tecla D.- si la nave en su coordenada *X* está dentro de la dimensión del escenario (dimx), entonces se le sumara 0.8 a la coordenada en *X* de la nave, se restara 0.45 al eje de rotación con pivote en *Z*, y se sumaran 0.7 y 0.8 a las coordenadas en *X* de la posición y la dirección de apuntado de la cámara respectivamente.

Una vez hechas las modificaciones, se carga la matriz identidad y se carga la función gluLookAt() con las nuevos parámetros de la cámara en cada una de las condicionales de las teclas.

Funciones y procesos extra

No se implemento una tecla para accionar los láseres (debido a ciertas dificultades), en cambio, estos se disparan automáticamente. Lo único que se hace para accionarlos es: dentro de la condicional de el accionamiento de teclas (*if not pausado and game_over == False:*), ir restando a la coordenada "*lz*" de los láseres de la nave 4, así este ira en linea recta hasta llegar al limite del escenario.

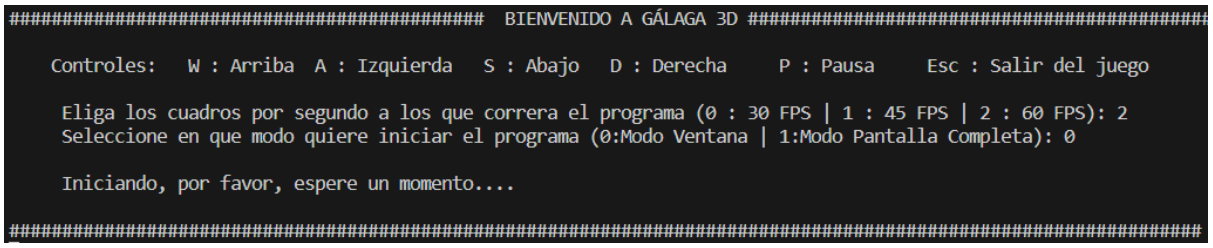
Este programa tiene dos funcionalidades extra:

- Este puede ejecutar en pantalla completa o en modo ventana; esto se configura dentro de la variable *pantalla*. Para el modo ventana, se llama a la primitiva "*pygame.display.set_mode()"*, en la cual se le pasa por parámetros las dimensiones de la ventana (*ventana*) y se inicializa OPENGL y el buffer doble. Para la pantalla completa, se cambia el parámetro ventana por pygame.FULLSCREEN y se centra la pantalla en (0,0).
- También se pueden cambiar los cuadros por segundo a la que se ejecuta el programa. Para ello, se crea el objeto clock de tipo "*pygame.time.Clock()*", y una variable *framerate.* Se hace una cadena de condicionales que permite cambiarlo entre 30 FPS, 45 FPS, y 60 FPS. Esta variable la lee la función "*clock.tick(framerate)*", creado a partir del objeto *clock* (la cual se encuentra dentro del ciclo *while* de la ejecución).

Estas funciones anteriores las puede cambiar el usuario al principio de la ejecución del programa.

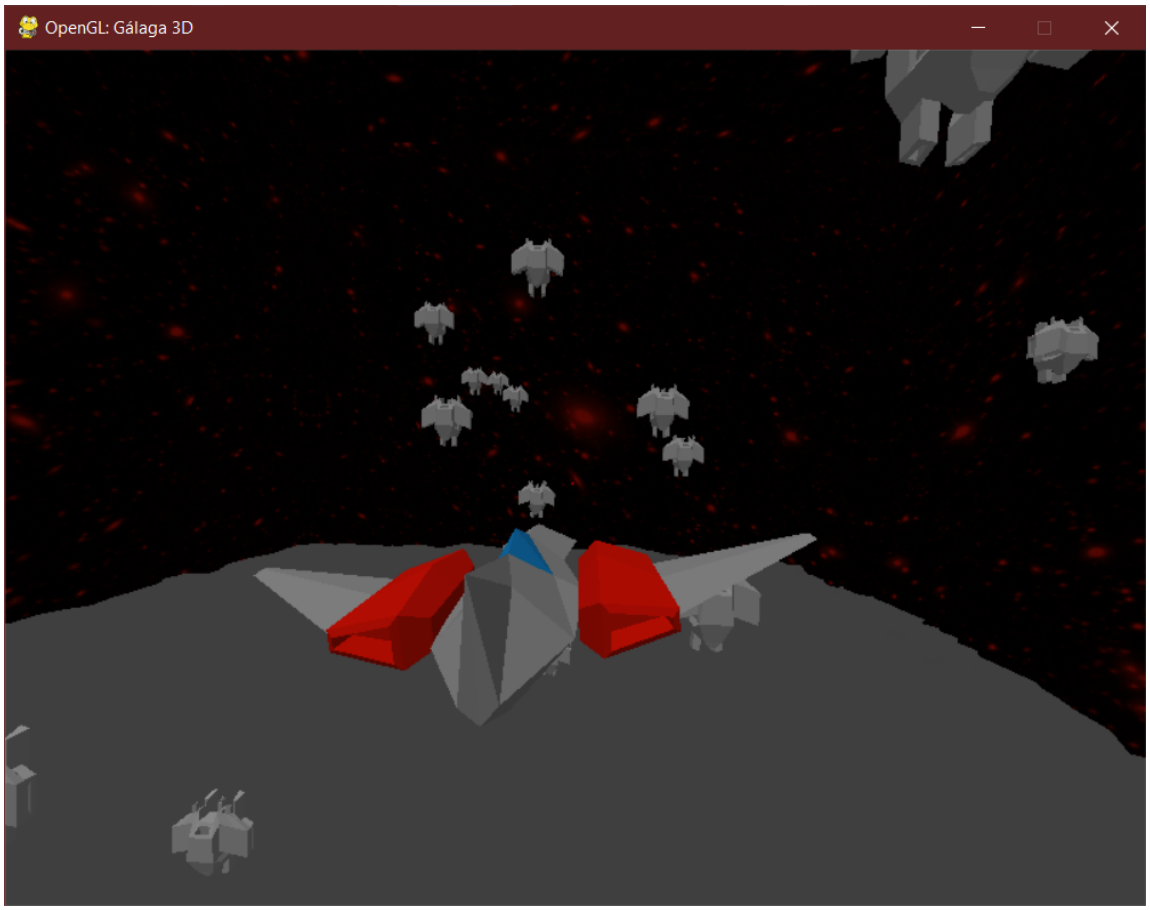
Pruebas del funcionamiento del programa

Se mostraran algunas capturas de pantalla sobre el funcionamiento del proyecto:



Menú contextual de Galaga 3D

Videojuego en acción



Pantalla de Game Over

Para visualizar correctamente el funcionamiento del programa, se dejara un enlace a un video en Google Drive aquí abajo:

- https://drive.google.com/file/d/1WZAGShWST93LgODq0Aeht1JxsyQ14_07/view?usp=sharing

CONCLUSIONES

Este proyecto resulto ser bastante entretenido de crear. Se hizo uso de todo lo que se vio en clase con respecto al manejo de escenas 3D en OpengGL, exceptuando la iluminación, el cual termino siendo confuso de implementar.

Ahora con estos conocimientos adquiridos y llevados a la practica, se puede seguir avanzando y creando proyectos mas sofisticados, sean escenas, animaciones o videojuegos. El uso de las primitivas de pygame y OpenGL facilitan bastante la implementación de movimientos, posicionamientos y rotaciones a nuestros objetos Wavefront. De hecho, hasta se aprendio un poco sobre como usar Blender, pues se requirió de modificar los modelos Wavefront que se encontraron para que quedasen correctamente implementados en la escena.