Clave única: 291423

Francisco Jacob Flores Rodríguez

GRAFICACIÓN POR COMPUTADORA

REPORTE DEL LECTOR OBJ

OMAR RODRIGUEZ GONZÁLEZ

Martes, 17 de noviembre de 2020

INTRODUCCIÓN

A lo largo de nuestra vida muy probablemente hemos visto diversas animaciones en forma de cortometrajes, películas, videos, entre otros, pero tal vez nunca nos hemos puesto a pensar en cómo se hace ese proceso. Uno de los pasos para estos procesos es el cargar archivos que contengan los objetos que se manejarán durante el proyecto. Es aquí en donde entran los archivos OBJ.

Es muy seguro que se utilicen técnicas más avanzadas en las grandes producciones, pero este trabajo nos sirve para entender los elementos de un lector y cómo es que se manejan los dichosos archivos OBJ. Como su nombre lo indica, el lector de OBJ leerá y guardará en memoria archivos con dicha extensión que manejen al menos vértices y caras. Las caras de un polígono están formadas por aristas, los cuales se infieren tomando en cuenta la información dada por la cara, y las aristas están formados por vértices. Además, puede haber vértices normales, vértices de textura, y más datos que son relevantes, pero en el caso del lector no tomaremos en cuenta todos estos elementos.

Todos estos datos que he mencionado están incluidos en los archivos con dicha extensión, delimitados por espacios, diagonales, o algún otro delimitador. Esto define la forma en que funciona el lector, ya que se toman en cuenta los delimitadores para cargar los datos a las instancias que a cada uno le corresponde. Es así como al tener la información cargada en memoria, se podrá dibujar al objeto en pantalla para hacer las respectivas transformaciones y desplazamientos esperados. Es por esto por lo que el lector es parte esencial del manejo de los archivos, ya que permitirán trabajar con los objetos deseados.

DISEÑO DEL LECTOR

EXPLICACIÓN DE UN ARCHIVO OBJ

Un archivo OBJ está organizado de diversas maneras, pero todas siguiendo una nomenclatura ya establecida, aunque a nosotros solo nos importan algunas para el proyecto. Estas van indicadas en cada línea por letras que definen cada tipo de datos:

- **o:** Nombre del objeto

```
o BasketballBall
```

- **g:** Nombre del grupo (también lo tomamos como nombre del objeto en el proyecto).

```
g cube
```

- **f:** Una cara del objeto.

```
f 1//2 7//2 5//2
f 1//2 3//2 7//2
f 1//6 4//6 3//6
f 1//6 2//6 4//6
f 3//3 8//3 7//3
f 3//3 4//3 8//3
f 5//5 7//5 8//5
f 5//5 8//5 6//5
f 1//4 5//4 6//4
f 1//4 6//4 2//4
f 2//1 6//1 8//1
f 2//1 8//1 4//1
```

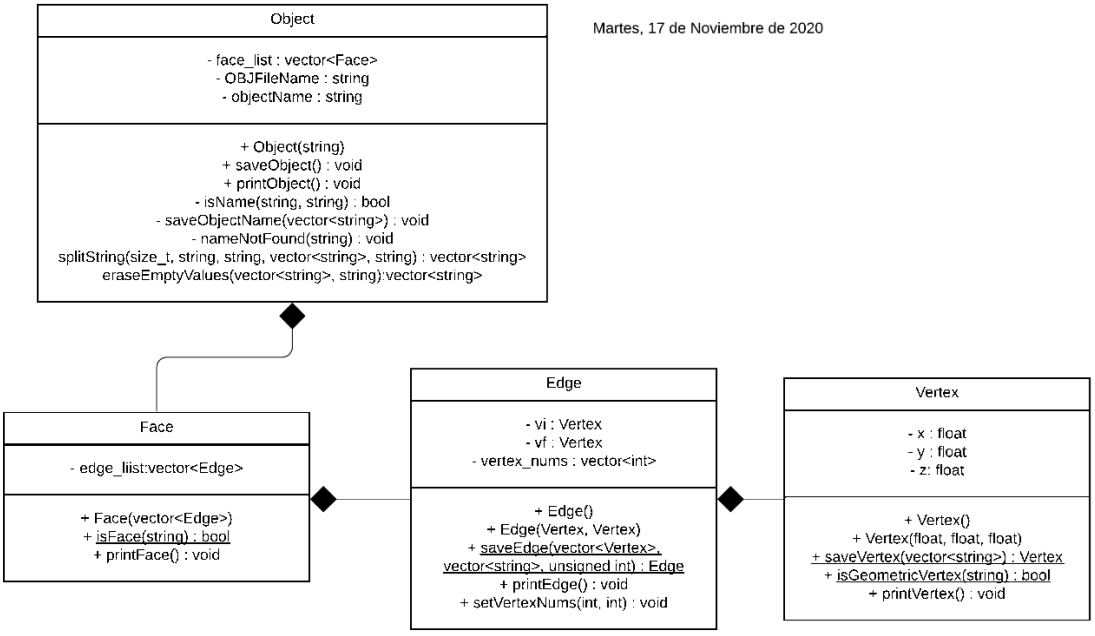
- **v:** Un vértice con sus coordenadas (x, y, z, w (opcional)).

```
v 0.0 0.0 0.0
v 0.0 0.0 1.0
v 0.0 1.0 0.0
v 0.0 1.0 1.0
v 1.0 0.0 0.0
v 1.0 0.0 1.0
v 1.0 1.0 0.0
v 1.0 1.0 1.0
```

- **vn:** vértice normal.

```
vn 0.0 0.0 1.0
vn 0.0 0.0 -1.0
vn 0.0 1.0 0.0
vn 0.0 -1.0 0.0
vn 1.0 0.0 0.0
vn -1.0 0.0 0.0
```

DIAGRAMA DE CLASES



*** DIAGRAMA HECHO EN: lucidchart (lucid.app)**

En el diagrama de clases se muestran las 4 clases que hasta ahora se manejarán para la realización del programa. Las flechas que se ven en el diagrama son un tipo de Aggregation, específicamente Composition. Esta indica que el hijo no puede existir independientemente del padre. En este caso la generalización es el Objecto, pero como no manejé herencia, esta es la forma en la que supe representar la relación que hay entre las clases, pero probablemente esté incorrecto.

CLASES:

- **Object:** Es la clase que manejará al objeto con todos sus elementos, es decir, sus caras, aristas, y vértices.
- **Face:** Esta clase manejará las caras del objeto y guardará las aristas que forman las caras.
- **Edge:** Esta clase manejará las aristas del objeto. Guardará los números de vértice de las aristas y sus vértices como tal.
- **Vertex:** Esta clase manejará los vértices, es decir, sus coordenadas.

Estructuras de datos usadas:

- **Vector:** Un vector es una estructura de datos similar a los arreglos. A diferencia de los arreglos, tienen la capacidad de crecer o decrecer dinámicamente. También es algo parecida a una lista por esta última característica mencionada. Se utiliza para guardar la lista de caras, de vectores y de vértices.

PRUEBAS

```
int main(){
    // Utilizar el constructor con el nombre del archivo.
    Object obj = Object("modelsOBJ/OrangeCartoon_basketball_ball_OBJ.obj");
    obj.saveObject();
    // std::cout << "\n Terminó de guardar el objeto.\n";
    obj.printObject();
}
```

Para comenzar con las pruebas primero debemos indicar la carpeta en donde se encuentra el archivo OBJ y el nombre del archivo. Es así como se guardará en memoria y se podrá manipular.

```
STRE-Basquetbol_y_lector-OBJ/LECTOR OBJ$ make
g++ -I./   -c -o include/object.o include/object.cpp
g++ main.o include/face.o include/edge.o include/vertex.o include/object.o -o main -l glfw -l
GLEW -l GL -l GLU -l armadillo
jake@Jake-XUbuntu20:/media/sf_GRAFICACION_POR_COMPUTADORA_[9-10AM]/PROYECTO_SEMESTRE-Basquetbo
l_y_lector-OBJ/LECTOR OBJ$ ./main
```

En las siguientes imágenes se encuentra la prueba que hice con el archivo OBJ que utilizaré para realizar el proyecto. Después de guardar la información con Object::saveObject(), la muestra con el método de impresión. De esta forma sabemos que se guardaron los datos de forma correcta. Específicamente se muestran el número de cara, el total de aristas, y los vértices que forman a cada arista.

```
- CARA 644

-> NÚMERO de ARISTAS en esta cara: 4

        -> ARISTA: 1

- VÉRTICES: 618, 620

Vertice inicial:
 -> x = -0.010843, y = -0.346438, z = -0.14129

Vertice final:
 -> x = -0.010843, y = -0.311769, z = -0.206161


        -> ARISTA: 2

- VÉRTICES: 620, 621

Vertice inicial:
 -> x = -0.010843, y = -0.311769, z = -0.206161

Vertice final:
 -> x = 0.010835, y = -0.311769, z = -0.206161


        -> ARISTA: 3

- VÉRTICES: 621, 619

Vertice inicial:
 -> x = 0.010835, y = -0.311769, z = -0.206161

Vertice final:
 -> x = 0.010835, y = -0.346438, z = -0.14129


        -> ARISTA: 4
```

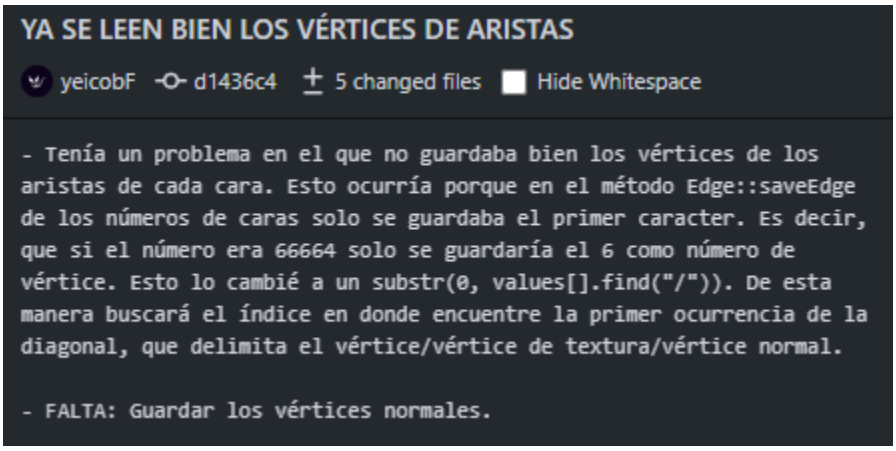
```
 -> x = 0.010835, y = -0.208317, z = -0.309613

Vertice final:
 -> x = -0.010843, y = -0.208317, z = -0.309613

-----------------------------


- CARA 648

-> NÚMERO de ARISTAS en esta cara: 4

        -> ARISTA: 1

- VÉRTICES: 626, 628

Vertice inicial:
 -> x = -0.010843, y = -0.143491, z = -0.344263

Vertice final:
 -> x = -0.010843, y = -0.073151, z = -0.3656


        -> ARISTA: 2

- VÉRTICES: 628, 629

Vertice inicial:
 -> x = -0.010843, y = -0.073151, z = -0.3656

Vertice final:
 -> x = 0.010835, y = -0.073151, z = -0.3656


        -> ARISTA: 3

- VÉRTICES: 629, 627

Vertice inicial:
 -> x = 0.010835, y = -0.073151, z = -0.3656
```

```
 -> x = -0.010829, y = -0.361721, z = -0.098207


        -> ARISTA: 2

- VÉRTICES: 617, 616

Vertice inicial:
 -> x = -0.010829, y = -0.361721, z = -0.098207

Vertice final:
 -> x = 0.010821, y = -0.361721, z = -0.098207


        -> ARISTA: 3

- VÉRTICES: 616, 663

Vertice inicial:
 -> x = 0.010821, y = -0.361721, z = -0.098207

Vertice final:
 -> x = 0.010804, y = -0.368339, z = -0.070605


        -> ARISTA: 4

- VÉRTICES: 663, 662

Vertice inicial:
 -> x = 0.010804, y = -0.368339, z = -0.070605

Vertice final:
 -> x = -0.010809, y = -0.368473, z = -0.070009

-----------------------------

- ESTE OBJETO TIENE 662 CARAS -
```

PROBLEMAS ENCONTRADOS

A lo largo de la realización del lector me encontré con diversos problemas, entre ellos el que no funciona con todos los archivos OBJ. Esto me hizo optar a hacer un repositorio en Git para mantener un control de los cambios que he realizado y los problemas con los que me he encontrado.

- **No guardaba correctamente los números de vértices de las aristas. Esto ocurría porque en el método que realizaba estas operaciones solamente se guardaba el primer carácter de la subcadena del vértice. Lo solucioné cambiando la implementación a cuando encontrara la ocurrencia de la diagonal.**



- **Si al final de una fila de datos hay un espacio, este genera problemas.**
- **Problemas con la impresión, ya que se guarda la "f" de las caras en el vector de values (el cual es de tipo string). Esto lo solucioné poniendo el valor que no se debía ingresar en cada caso (en este caso la f), así no la agregaría al vector y solo agregaría los valores.**
- **Al correr el programa, cuando entra a hacer el setVertexNums(); lanza este error:**

malloc(): corrupted top size
Aborted (core dumped)

El problema era que utilizaba la palabra reservada this -> en la clase Edge para el edge_list, así que lo sustituí por esto:

```
Face::Face(vector <Edge> _edge_list){
// Como el this.atributo en Java.
// this -> edge_list = edge_list;
edge_list = _edge_list; }
```

- Como el this -> es un apuntador, necesitaría tener un espacio de memoria reservada, y aún más siendo un vector. Después de que lo cambié, ya no tuve ese problema.

- **Adjunto una captura de pantalla de algunos errores con los que me encontré (no son muy específicos en el título, pero sí en la descripción):**