UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN

Facultad de Ingeniería

Escuela Profesional de Ingeniería en Informática y Sistemas

Laboratorio

"Creación de una cámara para representar escenas en OpenGL"

Docente : Ing. Omar la Torre Vilca

Curso : Diseño Asistido por Computadora

Nombres y apellidos :

- Juan Alberto Flores Pari 2018-119048
- Cristhian Santos Castillo Sanca 2018-119044

Ciclo : 8vo Ciclo

Año de estudios : 4to año

TACNA - PERÚ

2022

Creación de una función que simula una cámara para la representación de escenas en OpenGL usando transformaciones elementales

Introducción

En el presente trabajo desarrollaremos una función que simule una cámara para representar y/o movernos en las escenas de OpenGL, todo esto usando transformaciones elementales.

La función posicionara la "cámara" en un determinado punto de coordenadas y desde ahí podrá representar la escena que hayamos generado, bien sea objetos tridimensionales.

1. Objetivos

1.1. Crear una función que simule una cámara para la representación de objetos.
1.2. Con la cámara implementada necesitamos realizar rotaciones, acercamientos, alejamientos y una visión desde distintos ángulos de la escena.
1.3. Para realizar dichos movimientos se implementará la captura del teclado del usuario.

1. Desarrollo

1.1. Antecedentes

1.1.1. Función gluLookAt:

Esta función implementada en OpenGL determina dónde y cómo está dispuesta la cámara, se representa de la siguiente manera:

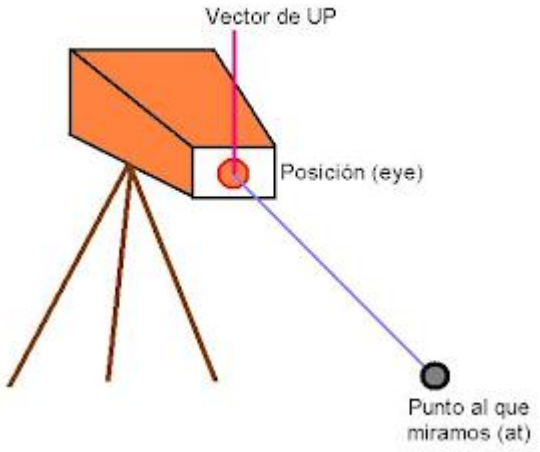
```
gluLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
```

Donde:

Las coordenadas "eye" es la posición de nuestra cámara.

Las coordenadas "at" es el punto de apunta nuestra cámara.

Las coordenadas del vector “up” nos permitirá regular la orientación de la cámara.



1.1.2. C++:

Es un lenguaje de programación desarrollado por Bjarne Stroustrup en 1979 basándose en el lenguaje de programación C para extender la manipulación de objetos dentro de la programación.

1.1.3. OpenGL:

Es una especificación que describe un conjunto de funciones y su comportamiento, que permite al desarrollador dibujar escenas tridimensionales a partir de primitivas como el punto, línea o simples polígonos. Fue desarrollado por Silicon Graphics Inc. en 1992.

1.2. Desarrollo de la función.

En esta sección explicaremos el código paso a paso. Iniciamos declarando las librerías que serán necesarias para la compilación del programa, incluyendo las librerías de OpenGL, las cuales nos permitirán usar funciones predeterminadas para confeccionar la nueva función.

```
#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glut.h>

#include <fstream>
#include <stdio.h>
#include "math.h"

using namespace std;
```

Imagen 2: Inserción de las librerías para la ejecución del programa [Datos propios]

En seguida declaramos una clase denominada CVector, donde declararemos las coordenadas y las funciones de nuestro código.

```
class CVector {

    private:

        float nOrm_x,nOrm_y,nOrm_z;  // Estas tres coordenadas forman el vector normal.

    public:

        // Constructores:
        // Por defecto con todas las coordenadas a 0, a las que se le indiquen y a las que se
        // le indiquen mas la normal.

        CVector ();
        CVector (float cx, float cy, float cz);
        CVector (float cx, float cy, float cz, CVector vNorm);

        ~CVector();

        void set(CVector v); // Da un valor al vector
        void setNormals(CVector vn); // Da valores al vector normal  ----SIN USO AUN----
        CVector getNormal(); // Devuelve los valores de las normales  ---- SIN USO AUN----


        float x,y,z;  // coordenadas x,y,z
        int indice;   // Indice para leer el formato .ASE ----SIN USO AUN----


        CVector operator +(CVector v2);   // Sobrecarga de los operadores para que sepa cómo actuar.
        CVector operator -(CVector v2);
};
```

Imagen 3: Se muestra la clase creada [Datos propios]

```
CVector::CVector() {

    x = 0;
    y = 0;
    z = 0;

    nOrm_x = 0;
    nOrm_y = 0;
    nOrm_z = 0;

    indice = 0;
}
```

Imagen 4: [Datos propios]

El código que se muestra es la función constructor de nuestra clase, donde inicializamos nuestras variables en 0.

```
CVector::CVector(float cx, float cy, float cz) {

    x = cx;
    y = cy;
    z = cz;

    indice = 0;

    nOrm_x = 0;
    nOrm_y = 0;
    nOrm_z = 0;


}

CVector::CVector(float cx, float cy, float cz, CVector vNorm) {

    x = cx;
    y = cy;
    z = cz;

    indice = 0;

    nOrm_x = vNorm.x;
    nOrm_y = vNorm.y;
    nOrm_z = vNorm.z;


}
```

Imagen 5: [Datos propios]

Las funciones que se muestran en la parte superior, asignamos las coordenadas en los vectores que usaremos.

```
void CVector::set(CVector v) {

    x = v.x;
    y = v.y;
    z = v.z;

}
```

Imagen 6: [Datos propios]

Asignamos valor del vector v hacia las coordenadas en x, y y z.

```
CVector CVector::operator +(CVector v2) {
    
    CVector v;

    v.x = x + v2.x;
    v.y = y + v2.y;
    v.z = z + v2.z;

    return v;
}

CVector CVector::operator -(CVector v2) {

    CVector v;

    v.x = x - v2.x;
    v.y = y - v2.y;
    v.z = z - v2.z;

    return v;
}
```

Imagen 7: [Datos propios]

Realizamos operaciones de suma y resta de lo vectores con estas funciones.

```
class CCamera {

    private:

        CVector cPos;
        CVector cView;
        CVector cUp;

        float cSpeed,rSpeed;

    public:

        CCamera();
        CCamera(CVector camera_pos,CVector camera_view ,CVector camera_up);
        ~CCamera();
        void SetPos(CVector camera_pos);  // Indica la posicion de la camara.

        // Estas 4 funciones  realizan el movimiento de la camara
        void Move(float speed);
        void rotate(float speed);
        void rotateUp(float droot);
        void StrafeUp(float droot);

        CVector getPos();    // Obtiene las coordenadas de posicion de la camara
        CVector getView();   // Obtiene las coordenadas del vector de dirección de la camara
        CVector getUp();     // Obtiene las coordenadas del vector que apunta hacia arriba de la camara
        void Update();  // actualiza glutLookAt(..) Con los parametros de la camara
        void setSpeed(float speed);
};
```

Imagen 8: [Datos propios]

Declaramos una nueva clase con el nombre “Ccamera”, aquí declaramos las variables en forma vectorial de la cámara como son la posición, la dirección donde apunta la cámara y un vector que nos permitirá el movimiento de arriba o abajo. También

declaramos variables de velocidad. De la misma forma se declara las funciones de la clase, su constructor y adicionalmente su destructor. Por ejemplo, la función “void Move” nos ayudara en el movimiento de la escena y la función “CVector getPos()” nos servirá para capturar la posición de la cámara.

```
CCamera::CCamera(){

    cPos.x = 0;
    cPos.y = 0;
    cPos.z = 0;

    cView.x = 0;
    cView.y = 0;
    cView.z = 0;

    cUp.x = 0;
    cUp.y = 0;
    cUp.z = 0;

}
```

Imagen 9: [Datos propios]

Realizamos el código del constructor de nuestra clase, inicializando los valores en 0.

```
CCamera::CCamera(CVector camera_pos,CVector camera_view ,CVector camera_up){

    cPos.x =    camera_pos.x;
    cPos.y =    camera_pos.y;
    cPos.z =    camera_pos.z;

    cView.x =  camera_view.x;
    cView.y =  camera_view.y;
    cView.z =  camera_view.z;

    cUp.x = camera_up.x;
    cUp.y = camera_up.y;
    cUp.z = camera_up.z;

    cSpeed = 0.5;
    rSpeed = 0.1;

}
```

Imagen 10: [Datos propios]

En esta función es donde asignaremos los valores de los vectores que habíamos obtenido en las funciones de la clase anterior. Así como también la asignación de un valor a la variable de velocidad.

```
CCamera::~CCamera(){

}
```

Imagen 11: [Datos propios]

Implementamos el destructor de la clase.

```
void CCamera::SetPos(CVector camera_pos) {

    cPos.x =    camera_pos.x;
    cPos.y =    camera_pos.y;
    cPos.z =    camera_pos.z;


}
```

Imagen 12: [Datos propios]

En esta función es donde indicamos la posición de la cámara con ayuda de los valores del vector posición.

```
void CCamera::Move(float dir) {

    CVector vDir;

    vDir = cView - cPos;

    cPos.x  = cPos.x  + vDir.x * (dir * cSpeed);
    cPos.z  = cPos.z  + vDir.z * (dir * cSpeed);
    cView.x = cView.x + vDir.x * (dir * cSpeed);
    cView.z = cView.z + vDir.z * (dir * cSpeed);

}
```

Imagen 13: [Datos propios]

Con esta función es donde obtenemos el vector de dirección de la cámara.

```
void CCamera::rotate(float droot) {

    CVector vDir = cView - cPos;

    cView.z = (float)(cPos.z + sin(rSpeed*droot )*vDir.x + cos(rSpeed*droot )*vDir.z);
    cView.x = (float)(cPos.x + cos(rSpeed*droot )*vDir.x - sin(rSpeed*droot )*vDir.z);

}
```

Imagen 14: [Datos propios]

En esta función es donde conseguiremos que la cámara haga el movimiento de rotación alterando los valores del vector de dirección.

```
void CCamera::rotateUp(float droot) {

    cView.y = cView.y + rSpeed*3*droot;


}
```

Imagen 15: [Datos propios]

Al igual que la función anterior, aquí lograremos rotar la cámara, pero ahora en él eje y.

```
CVector CCamera::getPos() {

    CVector v;

        v.x = cPos.x;
        v.y = cPos.y;
        v.z = cPos.z;

    return v;
}
```

Imagen 16: [Datos propios]

Con esta función obtenemos la posición de la cámara y lo almacenamos en el vector v y retornamos su valor.

```
CVector CCamera::getView(){

    CVector v;

        v.x = cView.x;
        v.y = cView.y;
        v.z = cView.z;

    return v;
}
```

Imagen 17: [Datos propios]

Con esta función obtenemos la dirección donde apunta de la cámara y lo almacenamos en el vector v y retornamos su valor.

```
CVector CCamera::getUp(){

    CVector v;

        v.x = cUp.x;
        v.y = cUp.y;
        v.z = cUp.z;

    return v;
}
```

Imagen 18: [Datos propios]

Con esta función el vector en el Y y luego lo almacenamos en el vector v y retornamos su valor.

```
void lookAtView(float eyex, float eyey, float eyez,    float centerx, float centery, float centerz,    float  upx, float  upy, float  upz)
{
    // centerx es el punto donde la camara apunta
    // esta parte hace referencia al vector f o matris f del doc

    float vfnx = centerx - eyex; // Se calcula el vector del "eye"
    float vfny = centery - eyey;
    float vfnz = centerz - eyez;

    // normalizar el vector posición  , para tener un vector ortogonal
    float len = sqrt(vfnx * vfnx + vfny * vfny + vfnz * vfnz); // Se normaliza

    vfnx = vfnx / len;      // Se encuentra el vector "f"
    vfny = vfny / len;
    vfnz = vfnz / len;

    // Productos Vectoriales , producto cruz

    float vlx = vfny * upz - vfnz * upy;      // se calcula el vector "l"
    float vly = vfnz * upx - vfnx * upz;
    float vlz = vfnx * upy - vfny * upx;

    float vux = vly * vfnz - vlz * vfny;      // Se calcula el nuevo vector "u" (up)
    float vuy = vlz * vfnx - vlx * vfnz;
    float vuz = vlx * vfny - vly * vfnx;

    // Matrix
    float mat[16] = {
        vlx, vux, -vfnx, 0,
```

Imagen 19: [Datos propios]

Esta es la función más importante de nuestra codificación, realizamos las operaciones necesarias para preparar nuestra matriz.

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_{ve} \\ 0 & 1 & 0 & -y_{ve} \\ 0 & 0 & 1 & -z_{ve} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Una vez obtenido dicha matriz, obtenemos un producto matricial de la siguiente manera:

```
//Apply the matrix and
glMultMatrixf(mat);
```

Imagen 20: [Datos propios]

```
void CCamera::Update() {

    glLoadIdentity();
    lookAtView(cPos.x,cPos.y,cPos.z,cView.x,cView.y,cView.z,cUp.x,cUp.y,cUp.z);
    //gluLookAt(cPos.x,cPos.y,cPos.z,cView.x,cView.y,cView.z,cUp.x,cUp.y,cUp.z);

    glRotatef(angulo_y,0.0,0.1,0.0);
    glRotatef(angulo_x,1.0,0.0,0.0);
    //glTranslatef(translate_x,translate_y,translate_z);

}
```

Imagen 21: [Datos propios]

Es aquí donde llamamos la función que habíamos creado con anterioridad, enviando algunos parámetros para que dicha función realice sus procedimientos.

```
void Draw3dCubeSpace() {

    int max_lines = 100;
    int i = -max_lines ;

    glBegin(GL_LINES);

        for (i=-max_lines ; i < max_lines ;i++) {

            // suelo

            glColor3f(1,0,0);
                glVertex3f(-max_lines ,0,i);
                glVertex3f(max_lines ,0,i);

            glColor3f(0,1,0);
                glVertex3f(i,0,max_lines );
                glVertex3f(i,0,-max_lines );

            // techo

            glColor3f(1,0,0);
                glVertex3f(-max_lines ,max_lines,i);
                glVertex3f(max_lines ,max_lines,i);

            glColor3f(0,1,0);
                glVertex3f(i,0,max_lines );
                glVertex3f(i,0,-max_lines );

            //paredes

            glColor3f(0,0,1);
                glVertex3f(i,0,-max_lines );
                glVertex3f(i,max_lines ,-max_lines );
```

Imagen 22: [Datos propios]

En esta captura de pantalla presentamos parte del código para dibujar líneas como parte de la escena que queremos representar.

```
void Draw3dCube() {

    CVector v;
    CVector vNorm1(-1, 1, 1);
    CVector vNorm2(-1, -1, 1);
    CVector vNorm3(1, -1, 1);

    glColor3f(0.0f,1.0f,0.0f);


 glBegin(GL_QUADS);
      // front
      glNormal3f(1, 0, 0);
      glColor3f(1, 0,0 );
      glVertex3f(-1, 1, 1);
      glVertex3f(-1, -1, 1);
      glVertex3f(1, -1, 1);
      glVertex3f(1, 1, 1);

      // back
      glNormal3f(0, 0, -1);
      glColor3f(0, 1, 0);
      glVertex3f(-1, 1, -1);
      glVertex3f(1, 1, -1);
      glVertex3f(1, -1, -1);
      glVertex3f(-1, -1, -1);

      // top
      glNormal3f(0, 1, 0);
      glColor3f(0, 0, 1);
      glVertex3f(-1, 1, -1);
      glVertex3f(-1, 1, 1);
      glVertex3f(1, 1, 1);
```

Imagen 23: [Datos propios]

Esto es parte del código para dibujar un objeto tridimensional, en este caso un cubo 3D y formar parte de la escena que queremos representar.

```
CCamera camara(CVector(0, 2,10),CVector(0,2,0),CVector(0.0,1.0,0.0));
```

Imagen 24: [Datos propios]

Instanciamos nuestra clase y mandamos los valores de los vectores que tenemos para trabajarlos.

Una vez instanciada nuestra clase, podemos hacer uso de las funciones que contiene nuestra clase tal como lo veremos a continuación.

```
camara.Update();
```

Imagen 25: [Datos propios]

Actualizamos la posición de la cámara con los valores del objeto “camara”.

```
void  specialfunc(int key, int x, int y) {

    switch(key) {
        case GLUT_KEY_UP :
            camara.Move(1);  // MOVE hace qu
                             // el valor 1 (
            render();
            break;
        case GLUT_KEY_DOWN :
            camara.Move(-1);
            render();
            break;
        case GLUT_KEY_PAGE_UP:
            camara.rotateUp(1); // RotateUp
            render();
            break;
        case GLUT_KEY_PAGE_DOWN:
            camara.rotateUp(-1);
            render();
            break;
        case GLUT_KEY_LEFT:
            camara.rotate(-1); // Rotate es
            render();
            break;
        case GLUT_KEY_RIGHT:
            camara.rotate(1);
            render();
            break;
        case GLUT_KEY_F1:
            render();
            break;
        case GLUT_KEY_F2:
            camara.StrafeUp(1);  // Para ir
            render();
            break;
        case GLUT_KEY_F3:
```

Imagen 26: [Datos propios]

Esta es la función donde implementamos el código para que pueda ser posible la captura del teclado y ejecutar las funciones de la clase “CCamera” a través del objeto “camara”.

```
int main(int argc, char* argv[])
{

    // Estas funciones inician glut. Indicamos que es doble buffer,
    // el tamaño, la posicion inicial y el nombre de la ventana.

    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE );
    glutInitWindowPosition(100,100);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Tutorial 0 - Espacio de trabajo 3D");

    glutShowWindow();

    initopengl();

    // Estas funciones son las que se repiten continuamente y serán
    // de tomar datos, visualizar...

    glutSpecialFunc(specialfunc);
    glutKeyboardFunc(keyboard);
    glutDisplayFunc(render);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);

    glutMainLoop();  // que no pare el circo, bucle infinito hasta c
                     // que llama a las funciones anteriores.



    return 0;
}
```

Imagen 27: [Datos propios]

Esta es la función principal donde podemos iniciar la ventana de trabajo y desde aquí podemos llamar las funciones para mostrar la escena, la cámara y poder hacer los movimientos de rotación, traslación, acercamiento y/o alejamiento de la escena.

Conclusiones

Con el trabajo realizado hemos alcanzado algunos objetivos planteados en la primera parte del documento. Hemos ido conociendo funciones nuevas de OpenGL, en especial el gluLookAt (). Para este último, lo que hicimos fue entender que es lo que hacía y poder implementarlo con nuestro propio razonamiento. De la misma forma hemos potenciado los conocimientos que teníamos en el lenguaje de programación de C++ como el manejo de clases o las funciones que ofrece OpenGL para el dibujado de objetos o escenas y así obtener la ayuda de un computador en el diseño gráfico.