UNIVERSIDAD CENTRAL DEL ECUADOR

FACULTAD DE INGENIERIA CIENCIAS FISICAS Y MATEMATICAS

INGENIERIA EN CONPUTACION GRAFICA

PROGRAMACION GRAFICA II

Nombre: Fabian Portero

Tema: Glu y Glut

Fecha: 17/04/19

Objetivo

Graficar objetos geométricos a partir de estas librerías haciendo uso de las primitivas predefinidas en las mismas.

Introducción

La animación generada por computadora en el cine y la televisión, así como los videojuegos de última generación, presenta agua, fuego y otros efectos naturales realistas. Muchas personas nuevas en gráficos de computadora se asombran al saber que estos modelos realistas y complejos son simples triángulos y píxeles en lo que respecta al hardware de gráficos de computadora.

GLU es el acrónimo de OpenGL Utility Library (se podría traducir como Biblioteca de utilidades para OpenGL). Esta biblioteca está compuesta por una serie de funciones de dibujo de alto nivel que, a su vez, se basan en las rutinas primitivas de OpenGL y se suele distribuir normalmente junto a él.

GLUT Mecanismos (del inglés OpenGL Utility Toolkit) es una biblioteca de utilidades para programas OpenGL que principalmente proporciona diversas funciones de entrada/salidacon el sistema operativo. Entre las funciones que ofrece se incluyen declaración y manejo de ventanas y la interacción por medio de teclado y ratón. También posee rutinas para el dibujado de diversas primitivas geométricas (tanto sólidas como en modo wireframe) que incluyen cubos, esferas y teteras. También tiene soporte para creación de menús emergentes.

Desarrollo

El glBegin () / glEnd () paradigma OpenGL ® Distilled cubre el glBegin / () glEnd () paradigma para propósitos ilustrativos solamente. La mayoría de las implementaciones de OpenGL evitan el uso de glBegin () / glEnd () para especificar la geometría debido a sus problemas de rendimiento inherentes.

Datos de vértices: este capítulo cubre los datos de coordenadas normales y de textura y omite otros datos de vértices, como los atributos de vértice (utilizados en los sombreadores de vértices), las marcas de borde y las coordenadas de niebla.

Asignación y desasignación de objetos de búfer: este capítulo no trata la interfaz para alterar dinámicamente partes de datos de objetos de búfer.

Evaluadores: OpenGL permite a los programadores representar curvas y superficies implícitas desde puntos de control.

Rectángulos: como puede especificar vértices para representar cualquier forma deseada, esta interfaz abreviada para dibujar rectángulos en el plano rara vez se usa.

Funcionalidad de matriz de vértice completa: este libro presenta un subconjunto de la interfaz de matriz de vértice y no cubre matrices intercaladas; tipos de datos de matrices de vértices distintos de GL_FLOAT y GL_DOUBLE; y algunos comandos de representación de matrices de vértices, como glDrawArrays () .

Este libro no cubre todas las características que afectan el color final y la apariencia de la geometría renderizada, como sombreadores de niebla, plantilla, vértice y fragmento, y otras características relacionadas.

Aunque son útiles en muchas circunstancias de representación, estas características no son esenciales para la programación OpenGL. Si su aplicación requiere esta funcionalidad, consulte OpenGL ® Guía de programación , OpenGL ® Manual de referencia , y OpenGL ® lenguaje de sombreado .

Primitivas de OpenGL

En OpenGL, las aplicaciones procesan primitivas especificando un tipo primitivo y una secuencia de vértices con datos asociados. El tipo primitivo determina cómo OpenGL interpreta y representa la secuencia de vértices.

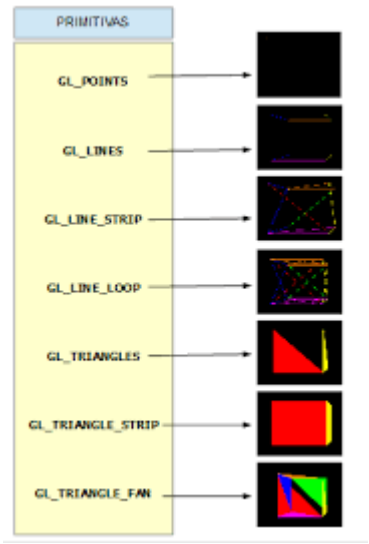

Gráfico (1)

En el Grafico (1) se muestran todas las primitivas de Open GL

OpenGL interpreta los vértices y representa cada primitiva usando las siguientes reglas:

GL_POINTS – Usa este tipo primitivo para representar puntos matemáticos. OpenGL representa un punto para cada vértice especificado.

GL_LINES : use esta primitiva para dibujar segmentos de línea no conectados. OpenGL dibuja un segmento de línea para cada grupo de dos vértices. Si la aplicación especifica n vértices, OpenGL procesa n / 2 segmentos de línea. Si n es impar, OpenGL ignora el vértice final.

GL_LINE_STRIP : use esta primitiva para dibujar una secuencia de segmentos de línea conectados. OpenGL representa un segmento de línea entre los vértices primero y segundo, entre el segundo y el tercero, entre el tercero y el cuarto, y así sucesivamente. Si la aplicación especifica n vértices, OpenGL procesa n –1 segmentos de línea.

GL_LINE_LOOP : use esta primitiva para cerrar una franja de línea. OpenGL presenta esta primitiva como un GL_LINE_STRIP con la adición de un segmento de línea de cierre entre los vértices final y primero.

GL_TRIANGLES – Usa esta primitiva para dibujar triángulos individuales. OpenGL representa un triángulo para cada grupo de tres vértices. Si su aplicación especifica n vértices, OpenGL procesa n / 3 triángulos. Si n no es un múltiplo de 3, OpenGL ignora los vértices en exceso.

GL_TRIANGLE_STRIP : usa esta primitiva para dibujar una secuencia de triángulos que comparten bordes. OpenGL representa un triángulo utilizando los vértices primero, segundo y tercero, y luego otro utilizando los vértices segundo, tercero y cuarto, y así sucesivamente. Si la aplicación especifica n vértices, OpenGL procesa n –2 triángulos conectados. Si n es menor que 3, OpenGL no procesa nada.

GL_TRIANGLE_FAN : usa esta primitiva para dibujar un abanico de triángulos que comparten bordes y también comparten un vértice. Cada triángulo comparte el primer vértice especificado. Si la aplicación especifica una secuencia de vértices v, OpenGL representa un triángulo utilizando v 0 , v 1 y v 2 ; otro triángulo usando v 0 , v 2 y v 3 ; otro triángulo usando v 0 , v 3 y v 4 ; y así. Si la aplicación especifica n vértices, OpenGL procesa n–2 triángulos conectados. Si n es menor que 3, OpenGL no procesa nada.

GL_QUADS : use esta primitiva para dibujar cuadriláteros convexos individuales. OpenGL representa un cuadrilátero para cada grupo de cuatro vértices. Si la aplicación

especifica n vértices, OpenGL procesa n / 4 cuadriláteros. Si n no es un múltiplo de 4, OpenGL ignora los vértices en exceso.

GL_QUAD_STRIP : usa esta primitiva para dibujar una secuencia de cuadriláteros que comparten aristas. Si la aplicación especifica una secuencia de vértices v, OpenGL representa un cuadrilátero usando v 0 , v 1 , v 3 y v 2 ; otro cuadrilátero que usa v 2 , v 3 , v 5 y v 4 ; y así. Si la aplicación especifica n vértices, OpenGL procesa (n -2) / 2 cuadriláteros. Si n es menor que 4, OpenGL no procesa nada.

GL_POLYGON – Use GL_POLYGON para dibujar una primitiva n -gon convexa rellena única . OpenGL representa un polígono de n lados, donde n es el número de vértices especificado por la aplicación. Si n es menor que 3, OpenGL no procesa nada.

Para GL_QUADS, GL_QUAD_STRIP y GL_POLYGON , todas las primitivas deben ser planas y convexas. De lo contrario, el comportamiento de OpenGL no está definido. La biblioteca de GLU es compatible con la teselación de polígonos, que permite a las aplicaciones generar primitivas rellenas que son no convexas o con intersección propia, o que contienen orificios. Consulte el conjunto de funciones “gluTess” en el Manual de referencia de OpenGL ® para obtener más información.

Compartir Vertices

Tenga en cuenta que GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN y GL_QUAD_STRIP comparten vértices entre sus segmentos de línea, triángulos y cuadriláteros. En general, debe usar estos primitivos cuando sea posible y práctico para reducir el cálculo redundante por vértice.

Podría representar una primitiva GL_QUAD_STRIP de dos cuadriláteros utilizando GL_QUADS , por ejemplo. Representado como un GL_QUAD_STRIP , su aplicación necesitaría enviar solo seis vértices únicos. Sin embargo, la versión GL_QUADS de esta primitiva requeriría ocho vértices, dos de los cuales son redundantes. Pasar vértices idénticos a OpenGL aumenta el número de operaciones por vértice y podría crear un cuello de botella en el rendimiento en la tubería de representación.

Adicionalmente podemos hacer uso de la librería GLUT que contiene otras primitivas adicionales. A continuación se muestra el Grafico (2) con ejemplos de todas las primitivas posibles a dibujar con GLUT.

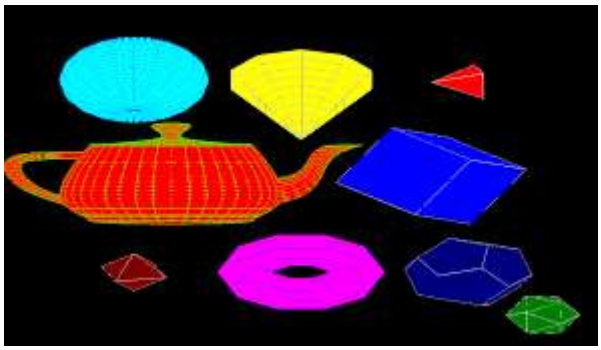

Gráfico (2)

Resultados

Haciendo uso de GLU y GLUT se construyó un sistema solar, también se implementó un sistema de cámaras mediante el uso de LookAT como se muestra en el Grafico (3)

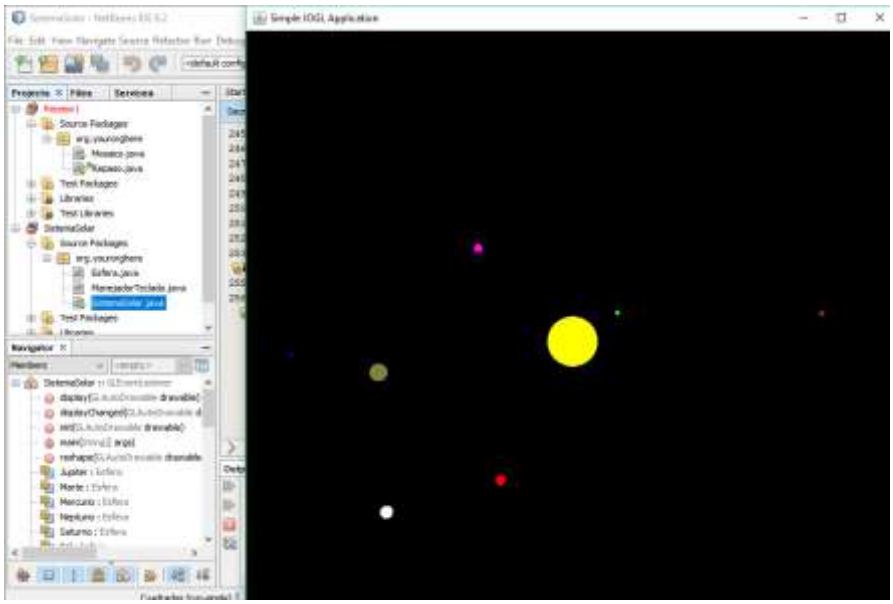

Gráfico (3)

Conclusiones

A partir de estas figuras (primitivas) se pueden construir objetos mas complejos e incluso representar entornos como el caso de esta práctica. Además las funciones que incorporan estas librerías nos permite mayor versatilidad y mejor control sobre las primitivas.

Anexos

Aquí se adjunta el código usado para la creación del sistema solar a partir de las librerías GLU y GLUT, desarrollado en Java.

package org.yourorghere;

```
import com.sun.opengl.util.Animator;
import java.awt.Frame;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.ArrayList;
import javax.media.opengl.GL;
import javax.media.opengl.GLAutoDrawable;
import javax.media.opengl.GLCanvas;
import javax.media.opengl.GLEventListener;
import javax.media.opengl.glu.GLU;
import java.lang.Math.* ;

public class SistemaSolar implements GLEventListener {

    Esfera Sol,Mercurio,Venus,Tierra,Marte,Jupiter,Saturno,Urano,Neptuno;

    float rme,rve,rti,rma,rju,rsa,rur,rne,x,y,z;



     public static  float trasladaX=0;
   public static float trasladaY=0;

   public static    float escalaX=1;
   public static   float escalaY=1;

   public static   float rotarX=0;
   public static    float rotarY=0;
    public static    float rotarZ=0;

    int a = 5;
    public static int cam = 1;
    public ManejadorTeclado mt;
    float i=0,b1=-40,b2=30;
    int j=0;
```

```
int [] posx = {1,2,3,4};
 int [] posy= {1,2,3,4};
public static void main(String[] args) {
    Frame frame = new Frame("Simple JOGL Application");
    GLCanvas canvas = new GLCanvas();

    canvas.addGLEventListener(new SistemaSolar());
    frame.add(canvas);
    frame.setSize(800, 800);
    final Animator animator = new Animator(canvas);
    frame.addWindowListener(new WindowAdapter() {

        @Override
        public void windowClosing(WindowEvent e) {
            // Run this on another thread than the AWT event queue to
            // make sure the call to Animator.stop() completes before
            // exiting
            new Thread(new Runnable() {

                public void run() {
                    animator.stop();
                    System.exit(0);
                }
            }).start();
        }
    });
    // Center frame
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
    animator.start();
}

public void init(GLAutoDrawable drawable) {
    // Use debug pipeline
```

```
    // drawable.setGL(new DebugGL(drawable.getGL()));



    GL gl = drawable.getGL();
    System.err.println("INIT GL IS: " + gl.getClass().getName());

    // Enable VSync
    gl.setSwapInterval(1);

    // Setup the drawing area and shading mode
    gl.glClearColor(0, 0, 0,0);
   // gl.glClearColor(0, 0, 0,0);
    gl.glShadeModel(GL.GL_SMOOTH); // try setting this to GL_FLAT and see what happens.

    gl.glEnable(GL.GL_DEPTH_TEST);

      mt = new ManejadorTeclado();
     drawable.addKeyListener(mt);

    //sol
     Sol = new Esfera(gl, 1,50,50, 0,0,0,20,20,20 ,0,0,0,0,0,0);
    //Planetas
      Mercurio = new Esfera(gl, 1,50,50, 0,0,0,2,2,2 ,0,0,0,0,0,0);
       Venus = new Esfera(gl, 1,50,50, 0,0,0,2.5f,2.5f,2.5f ,0,0,0,0,0,0);
        Tierra = new Esfera(gl, 1,50,50, 0,0,0,3.5f,3.5f,3.5f ,0,0,0,0,0,0);
         Marte = new Esfera(gl, 1,50,50, 0,0,0,4,4,4 ,0,0,0,0,0,0);
          Jupiter = new Esfera(gl, 1,50,50, 0,0,0,7,7,7 ,0,0,0,0,0,0);
           Saturno = new Esfera(gl, 1,50,50, 0,0,0,5,5,5 ,0,0,0,0,0,0);
            Urano = new Esfera(gl, 1,50,50, 0,0,0,2,2,2 ,0,0,0,0,0,0);
             Neptuno = new Esfera(gl, 1,50,50, 0,0,0,1.5f,1.5f,1.5f ,0,0,0,0,0,0);

}

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
```

```
    GL gl = drawable.getGL();
    GLU glu = new GLU();

    if (height <= 0) { // avoid a divide by zero error!

        height = 1;
    }
    final float h = (float) width / (float) height;
    gl.glViewport(0, 0, width, height);
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluPerspective(45.0f, h, 1.0, 1000.0);
    gl.glMatrixMode(GL.GL_MODELVIEW);
    gl.glLoadIdentity();
}

public void display(GLAutoDrawable drawable) {
    GL gl = drawable.getGL();

    // Clear the drawing area
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
    // Reset the current matrix to the "identity"



    gl.glLoadIdentity();
    GLU glu= new GLU ();



   if(cam ==1){
        glu.gluLookAt(0,600,0, 0, -1, 0, 0, 0, 1);

   }
   if(cam ==2){
        glu.gluLookAt(0,400,200, 0, 0, 0, 1, 0, 1);
```

```
}
if(cam ==3){
     glu.gluLookAt(0,300,-600, 0, -1, 0, 0, 1, 0);

}



    gl.glTranslatef(trasladaX, trasladaY, 0);
      gl.glScalef(escalaX, escalaY, 1);
      gl.glRotatef(rotarX, 1, 0, 0);
      gl.glRotatef(rotarY, 0, 1, 0);
      gl.glRotatef(rotarZ, 0, 0, 1);




       //Sol


          gl.glPushMatrix();
          gl.glColor3f(1,1,0);
          gl.glTranslatef(0,0,0);
          gl.glRotatef(180,0,0,0);
          gl.glScalef(1,1,1);
              Sol.display();
          gl.glPopMatrix();

      //Planetas
          //Mercurio
          gl.glPushMatrix();
          gl.glColor3f(0,1,0);
          gl.glRotatef(rme+=1f,0,1,0);
```

```
            gl.glScalef(1,1,1);
            gl.glTranslatef(30,0,30);
                Mercurio.display();
            gl.glPopMatrix();


            //Venus
            gl.glPushMatrix();
            gl.glColor3f(0,0,0);
            gl.glRotatef(rve+=0.5,0,1,0);
            gl.glScalef(1,1,1);
            gl.glTranslatef(-65,0,-65);
               Venus.display();
            gl.glPopMatrix();




            //Tierra
            gl.glPushMatrix();
            gl.glColor3f(1,0,1);
//              gl.glRotatef(ry+=0.01f,0,1,0);
            gl.glRotatef(rti+=0.25f,0,1,0);
            gl.glScalef(1,1,1);
            gl.glTranslatef(-35,0,100);
//                gl.glTranslatef(2*(float) Math.cos(20),0,45*(float )Math.sin(20));
               Tierra.display();
            gl.glPopMatrix();

             //Marte
            gl.glPushMatrix();
            gl.glColor3f(1,0,0);
            gl.glRotatef(rma+=0.10f,0,1,0);
            gl.glScalef(1,1,1);
```

```
gl.glTranslatef(100,0,-75);
   Marte.display();
gl.glPopMatrix();

 //Jupiter
gl.glPushMatrix();
gl.glColor3f(0.5f,0.5f,0.25f);
gl.glRotatef(rju+=0.1f,0,1,0);
gl.glScalef(1,1,1);
gl.glTranslatef(150,0,45);
   Jupiter.display();
gl.glPopMatrix();

 //Saturno
gl.glPushMatrix();
gl.glColor3f(1,1,1);
gl.glRotatef(rsa+=0.05f,0,1,0);
gl.glScalef(1,1,1);
gl.glTranslatef(175,0,-100);
   Saturno.display();
gl.glPopMatrix();

 //Urano
gl.glPushMatrix();
gl.glColor3f(0.8f,0.125f,0.3f);
gl.glRotatef(rur+=0.025f,0,1,0);
gl.glScalef(1,1,1);
gl.glTranslatef(-200,0,0);
   Urano.display();
gl.glPopMatrix();

 //Neptuno
gl.glPushMatrix();
gl.glColor3f(0,0,1);
```

```
            gl.glRotatef(rne+=0.01f,0,1,0);
            gl.glScalef(1,1,1);
            gl.glTranslatef(225,0,0);
              Neptuno.display();
            gl.glPopMatrix();



        gl.glFlush();




    }


    public void displayChanged(GLAutoDrawable drawable, boolean modeChanged, boolean deviceChanged) {
    }
}
```

Bibliografia

Group, K. (2019). OpenGL - The Industry Standard for High Performance Graphics. Retrieved from http://www.opengl.org/