Universidad Nacional Mayor de
SAN MARCOS
Universidad del Perú. Decana de América

Facultad de Ingeniería de Sistemas e Informática

Computación Gráfica y Visual
Profesor: Paucar Curasma Herminio

Proyecto Final

https://github.com/r3gor/CG-Proyecto

Documentación

https://darkblood202.github.io/CG-Proyecto-doc/

Integrantes:

Alumno	Participación
GALARZA AREVALO JONATHAN	3
HIDALGO DÍAZ SEBASTIAN EDUARDO	3
HIDALGO DÍAZ SEBASTIAN EDUARDO	3
MOQUILLAZA ALCARRAZ SANTIAGO YOVANY	3
RAMOS PAREDES ROGER ANTHONY	3
RIOS JAIMES JHONEL	3
VILLARREAL DOROTEO OMAR	3

CONTENIDO

1. Introducción

El ser humano se ha caracterizado sobre otras especies por su gran habilidad creativa y capaz de reflejar toda su imaginación; sin embargo, también es posible un proceso análogo para reflejar su realidad.

La presente aplicación consiste en la tan idealizada generación de figuras sólidas realistas, que se puede definir como cuerpos geométricos formados por muchos puntos unidos en tres dimensiones que han formado capas o mallas entre ellos. Estas piezas son la base para formar distintas figuras, siendo la suma de ellos, intersección, escalamiento, etc. En la aplicación su formación empieza de la generación de puntos, a luego unirlos por distintas primitivas GL_TRIANGLE_STRIP, GL_TRIANGLES, GL_TRIANGLE_FAN y también con el fin de mostrar los puntos y sus conexiones usamos GL_LINES y sus distintas variaciones.

Como características principales tenemos que se podrá rotar la figura, mover con las teclas A-W-S-D , hacer zoom con la rueda del mouse, poder ver los puntos y uniones que lo conforman, darle smooth, mostrar las normales y muchas funcionalidades más.

2. Marco Teórico.

a. Algoritmos:

TORUS:

El primer paso consiste en instanciar un objeto de la clase Torus, que hereda de la clase Shape. Este objeto cuenta con un método constructor de la forma:

```
Torus(int num_x, int num_y, float Rad, float rad) : Shape(num_x, num_y), R(Rad), r(rad) {}
```

Instanciamos un objeto Torus que llamaremos *torusShape*, el cual inicializamos con los parámetros:

- num_x: 10
- num_y: 10
- Rad: 0.2
- rad: 0.05

Luego, llamamos al método *createVertexObjects()* para generar los VAO, VBOs y EBOs necesarios para renderizar la figura seguido del método *initData()* que calculará los vértices, puntos, normales e índices con los que se llenarán el VAO, VBOs y EBOs.

Lo siguiente es definido por la interacción de la aplicación con el usuario a través de la interfaz gráfica y es ejecutado dentro del bucle principal de la aplicación (es decir, mientras no se haya enviado una señal de cierre de ventana de la aplicación).

- Color de fondo y de figura.
- Figura.
- Shaders.
- Relleno.
- Normales.
- Wires.
- Segments.
- Propiedades de figura.

Empieza con checkbocks que nos ayudarán a controlar el color de fondo de la pantalla y de la figura.

Lo siguiente que se muestra es la lista de *checkboxes* con las diferentes figuras disponibles que podemos renderizar y visualizar en el programa. Podemos seleccionar una o varias (incluso todas, si así se desea), cada una seguirá las mismas propiedades que establezcamos por medio de la GUI. A su vez, al momento de seleccionarla, se nos presentará una pequeña ventana mediante la cual podemos modificar las propiedades de cada figura (dependiendo de cuál sea esta). Al ser un torus o figura especial , podemos modificar 2 parámetros:

- Radio mayor del torus
- Radio interior del torus

Al modificar cualquiera de estos parámetros, el listener de las propiedades de la figura detecta la modificación y procede a procesarla mediante el método *setProp()*, el cual accede al setter de la figura y actualiza sus puntos, vértices, etc mediante el método *initData()* coincidiendo con los nuevos valores introducidos por el usuario a través de las modificaciones en la GUI. Para evitar un bucle infinito, se establece un booleano bandera *prop_listener* a falso, que se tornará verdadero cuando se realice otra modificación en las propiedades de la figura.

Si seleccionamos la casilla de Shaders, se nos presentará una pequeña ventana que nos permitirá elegir entre tipos de shaders que deseamos ver en la figura renderizada. Estos son:

- Ordinario.
- BlingPong.
- Goraud.
- Phong.

Lo siguiente es el Relleno. La *checkbox* correspondiente nombra a esta opción como *Fill*. Si la marcamos, el programa llamará al método *renderFill()* de la figura (o figuras) que estemos renderizando en ese momento, pintándola con el color elegido en el selector de color.

El programa también nos ofrece la posibilidad de renderizar las normales por cada figura. Si se ha seleccionado la casilla *Normals*, se llamará al método *renderNormals()*.

Otra opción disponible por medio de la GUI es Render wire, la cual (si es que ha sido seleccionada), renderizará las líneas asociadas a los vértices de la figura sin el pintado entre ellas. Esto lo realiza mediante la llamada al método *renderWire()*.

Entre los últimos apartados, se tienen dos barras de sliders con los cuales puede modificarse los segmentos y stacks de las figuras, que afecta con cuántos triángulos (lo que se traduce a nivel de detalle) la figura va a ser renderizada. Cuando se detecta alguna acción en uno de los sliders, esta se procesa mediante el método *segmentsUpdate()* el cual envía los nuevos valores (seleccionados desde el slider en el GUI) a la figura. Inmediatamente se vuelve a llamar al método *initData()* de la figura para actualizar los vértices para que coincidan con los segment y stack definidos por el usuario. Finalmente, para evitar un bucle, se establece un booleano bandera *segments_event_listener* en falso, y que solo se tornará verdadero cuando vuelva a recibirse una acción en alguno de los sliders.

ALGORITMO UTILIZADO:
Este es el algoritmo utilizado para iniciar los datos del Torus(vértices y normales).

```
void Torus::initData()
{
   num_vertices = (seg_x + 1) * (seg_y + 1);
   restart_index = num_vertices;
   num_indices = (seg_x * 2 * (seg_y + 1)) + seg_x - 1;

   // build vertices and normals
   std::vector<float> vertices, normales;

   float dTheta = 2 * PI / float(seg_x);
   float dPhi = 2 * PI / float(seg_y);

   float theta = 0.0f;

   for (int i = 0; i <= seg_x; i++)
   {
      float phi = 0.0f;

      float sinTheta = sin(theta);
      float cosTheta = cos(theta);

      for (int j = 0; j <= seg_y; j++)
      {
         float sinPhi = sin(phi);
         float cosPhi = cos(phi);

         vertices.push_back((R + r * cosPhi)* cosTheta); // x
         vertices.push_back((R + r * cosPhi)* sinTheta); // y
         vertices.push_back(r * sinPhi); // z

         normales.push_back(cosTheta * cosPhi); // x
         normales.push_back(sinTheta * cosPhi); // y
         normales.push_back(sinPhi); // z

         phi += dPhi;
      }

      theta += dTheta;
   }
      float sinTheta = sin(theta);
      float cosTheta = cos(theta);

      for (int j = 0; j <= seg_y; j++)
      {
         float sinPhi = sin(phi);
         float cosPhi = cos(phi);
```

```
            vertices.push_back((R + r * cosPhi)* cosTheta); // x
            vertices.push_back((R + r * cosPhi)* sinTheta); // y
            vertices.push_back(r * sinPhi); // z

            normales.push_back(cosTheta * cosPhi); // x
            normales.push_back(sinTheta * cosPhi); // y
            normales.push_back(sinPhi); // z

            phi += dPhi;
        }
        theta += dTheta;
    }
```

Esto es para renderizar la figura, por medio de la primitiva GL_TRIANGLE_STRIP:

```
void Torus::renderFill() const{

    // draw torus fill

    vao.bind();
    ibo.bind();

    glEnable(GL_PRIMITIVE_RESTART);
    glPrimitiveRestartIndex(restart_index);
    glDrawElements(GL_TRIANGLE_STRIP, num_indices, GL_UNSIGNED_INT, (void *)(0));
    glDisable(GL_PRIMITIVE_RESTART);
}
```

CUBO:

CILINDRO:

Método set

```
void Cylinder::set(float baseRadius, float topRadius, float height, int sectors, int stacks, bool smooth)
{
    this->baseRadius = baseRadius;
    this->topRadius = topRadius;
    this->height = height;
    this->sectorCount = sectors;
    if(sectors < MIN_SECTOR_COUNT)
        this->sectorCount = MIN_SECTOR_COUNT;
    this->stackCount = stacks;
    if(stacks < MIN_STACK_COUNT)
        this->stackCount = MIN_STACK_COUNT;
    this->smooth = smooth;
```

El método *set()* se encarga de asignar los argumentos dados a los campos de datos de la clase, accesándolos mediante el puntero *this* que apunta al objeto. También delimita el mínimo número de segments y stacks que puede tener una figura; es decir, la cantidad mínima de polígonos de la figura que siempre van a ser renderizados a través de *MIN_SECTOR_COUNT* y *MIN_STACK_COUNT*.

Método renderFill()

```
void Cylinder::renderFill() const
{
    // interleaved array

    glBindVertexArray(vao);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_vert);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_norm);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indi);

    glDrawElements(GL_TRIANGLES,  count: (unsigned int)indices.size(), GL_UNSIGNED_INT,  indices: 0);
}
```

El método *renderFill()* se encarga del pintado de la figura. Se empieza enlazando el respectivo VAO de la figura y posteriormente el VBO para los vertices (*vbo_vert)*. Mediante el método *glVertexAttribPointer* pasamos esta data por el VertexShader

localizado en el atributo en locación 0 el cual indicaremos mediante *glEnableVertexAttribArray(0)* para que active específicamente ese atributo. y hacemos lo mismo con el VBO para las normales de la figura (*vbo_norm)*, pero con el atributo en locación 1.

Método buildUnitCircleVertices()

```
void Cylinder::buildUnitCircleVertices()
{
    const float PI = acos( x: -1);
    float sectorStep = 2 * PI / sectorCount;
    float sectorAngle;  // radian

    std::vector<float>().swap( & unitCircleVertices);
    for(int i = 0; i <= sectorCount; ++i)
    {
        sectorAngle = i * sectorStep;
        unitCircleVertices.push_back(cos(sectorAngle)); // x
        unitCircleVertices.push_back(sin(sectorAngle)); // y
        unitCircleVertices.push_back(0);                // z
    }
}
```

Este algoritmo genera los puntos de la base del cilindro y los guarda para utilizarlos nuevamente tanto para la base superior como inferior (o si se desea hacer escalamiento) sin necesidad de computar datos nuevos. Este método se llama nuevamente para generar el resto de puntos que conforman el resto de caras del sólido. Se guardan los puntos en un vector *unitCircleVertices* de tipo float, basado en el número de secciones de la figura.

Método generador de puntos initData()

```
 float x, y, z;                  // vertex position
//float s, t;                   // texCoord
float radius;                   // radius for each stack

// get normals for cylinder sides
std::vector<float> sideNormals = getSideNormals();

// put vertices of side cylinder to array by scaling unit circle
```

```
        for(int i = 0; i <= stackCount; ++i)
        {
        z = -(height * 0.5f) + (float)i / stackCount * height;    // vertex position z
        radius = baseRadius + (float)i / stackCount * (topRadius - baseRadius);
// lerp
        float t = 1.0f - (float)i / stackCount;   // top-to-bottom

        for(int j = 0, k = 0; j <= sectorCount; ++j, k += 3)
        {
        x = unitCircleVertices[k];
        y = unitCircleVertices[k+1];
        addVertex(x * radius, y * radius, z);   // position
        addNormal(sideNormals[k], sideNormals[k+1], sideNormals[k+2]); //
normal
        addTexCoord((float)j / sectorCount, t); // tex coord
        }
        }

        // remember where the base.top vertices start
        unsigned int baseVertexIndex = (unsigned int)vertices.size() / 3;

        // put vertices of base of cylinder
        z = -height * 0.5f;
        addVertex(0, 0, z);
        addNormal(0, 0, -1);
        addTexCoord(0.5f, 0.5f);
        for(int i = 0, j = 0; i < sectorCount; ++i, j += 3)
        {
        x = unitCircleVertices[j];
        y = unitCircleVertices[j+1];
        addVertex(x * baseRadius, y * baseRadius, z);
        addNormal(0, 0, -1);
        addTexCoord(-x * 0.5f + 0.5f, -y * 0.5f + 0.5f);         // flip horizontal
        }

        // remember where the base vertices start
        unsigned int topVertexIndex = (unsigned int)vertices.size() / 3;

        // put vertices of top of cylinder
        z = height * 0.5f;
        addVertex(0, 0, z);
        addNormal(0, 0, 1);
        addTexCoord(0.5f, 0.5f);
        for(int i = 0, j = 0; i < sectorCount; ++i, j += 3)
        {
        x = unitCircleVertices[j];
        y = unitCircleVertices[j+1];
        addVertex(x * topRadius, y * topRadius, z);
```

```
addNormal(0, 0, 1);
addTexCoord(x * 0.5f + 0.5f, -y * 0.5f + 0.5f);
}

// put indices for sides
unsigned int k1, k2;
for(int i = 0; i < stackCount; ++i)
{
k1 = i * (sectorCount + 1);  // bebinning of current stack
k2 = k1 + sectorCount + 1;          // beginning of next stack

for(int j = 0; j < sectorCount; ++j, ++k1, ++k2)
{
// 2 trianles per sector
addIndices(k1, k1 + 1, k2);
addIndices(k2, k1 + 1, k2 + 1);

// vertical lines for all stacks
lineIndices.push_back(k1);
lineIndices.push_back(k2);
// horizontal lines
lineIndices.push_back(k2);
lineIndices.push_back(k2 + 1);
if(i == 0)
{
        lineIndices.push_back(k1);
        lineIndices.push_back(k1 + 1);
}
}
}


// remember where the base indices start
baseIndex = (unsigned int)indices.size();

// put indices for base
for(int i = 0, k = baseVertexIndex + 1; i < sectorCount; ++i, ++k)
{
if(i < (sectorCount - 1))
addIndices(baseVertexIndex, k + 1, k);
else    // last triangle
addIndices(baseVertexIndex, baseVertexIndex + 1, k);
}

// remember where the base indices start
topIndex = (unsigned int)indices.size();

for(int i = 0, k = topVertexIndex + 1; i < sectorCount; ++i, ++k)
```

```
{
if(i < (sectorCount - 1))
addIndices(topVertexIndex, k, k + 1);
else
addIndices(topVertexIndex, k, topVertexIndex + 1);
}

// build vertices to paint normals
float long_linea = 0.012f;
for (int i = 0; i < normals.size(); i+=3) {
norm_lines.push_back(vertices[i]);
norm_lines.push_back(vertices[i+1]);
norm_lines.push_back(vertices[i+2]);

norm_lines.push_back(vertices[i] + normals[i] * long_linea);
norm_lines.push_back(vertices[i+1] + normals[i+1] * long_linea);
norm_lines.push_back(vertices[i+2] + normals[i+2] * long_linea);
}
```

CUBO

I. DESCRIPCIÓN DEL FUNCIONAMIENTO

Primero debemos instanciar un objeto de la clase Cubesphere, que hereda de la clase Shape. Este objeto cuenta con un método constructor de la forma:

```
Cubesphere(float radius, int sub, bool smooth) : radius(radius),
subdivision(sub), smooth(smooth), interleavedStride(32)
```

Instanciamos un objeto Cubesphere que llamaremos *cubeShape*, el cual inicializaremos con los parámetros:

- radius: 0.1
- sub: 0
- smooth: true

Luego, llamamos al método *createVertexObjects()* para generar los VAO, VBOs y EBOs necesarios para renderizar la figura seguido del método *initData()* que calculará los vértices, puntos, normales e índices con los que se llenarán el VAO, VBOs y EBOs.

Lo siguiente es definido por la interacción de la aplicación con el usuario a través de la interfaz gráfica y es ejecutado dentro del bucle principal de la aplicación (es decir, mientras no se haya enviado una señal de cierre de ventana de la aplicación).

- Color de fondo y de figura.
- Figura.
- Shaders.
- Relleno.
- Normales.
- Wires.
- Segments.
- Propiedades de figura.

Lo primero que se nos presenta en la interfaz gráfica es un selector de color para cambiar el color de nuestra figura (si es que hemos seleccionado la *checkbox* correspondiente a cualquiera de ellas) y el fondo. Los cambios que realicemos serán renderizados inmediatamente y a tiempo real en el programa.

Lo siguiente que se muestra es la lista de *checkboxes* con las diferentes figuras disponibles que podemos renderizar y visualizar en el programa. Podemos seleccionar una o varias (incluso todas, si así se desea), cada una seguirá las mismas propiedades que establezcamos por medio de la GUI. A su vez, al momento de seleccionarla, se

nos presentará una pequeña ventana mediante la cual podemos modificar las propiedades de cada figura (dependiendo de cuál sea esta). Al ser un cubo, podemos modificar dos parámetros:

- Radio de cubósfera.
- Arista

Al modificar cualquiera de estos parámetros, el listener de las propiedades de la figura detecta la modificación y procede a procesarla mediante el método *setProp()*, el cual accede al setter de la figura y actualiza sus puntos, vértices, etc mediante el método *initData()* coincidiendo con los nuevos valores introducidos por el usuario a través de las modificaciones en la GUI. Para evitar un bucle infinito, se establece un booleano bandera *prop_listener* a falso, que se tornará verdadero cuando se realice otra modificación en las propiedades de la figura.

Si seleccionamos la casilla de Shaders, se nos presentará una pequeña ventana que nos permitirá elegir entre tipos de shaders que deseamos ver en la figura renderizada. Estos son:

- Ordinario.
- BlingPong.
- Goraud.
- Phong.

Lo siguiente es el Relleno. La *checkbox* correspondiente nombra a esta opción como *Fill*. Si la marcamos, el programa llamará al método *renderFill()* de la figura (o figuras) que estemos renderizando en ese momento, pintándola con el color elegido en el selector de color.

El programa también nos ofrece la posibilidad de renderizar las normales por cada figura. Si se ha seleccionado la casilla *Normals*, se llamará al método *renderNormals()*.

Otra opción disponible por medio de la GUI es Render wire, la cual (si es que ha sido seleccionada), renderizará las líneas asociadas a los vértices de la figura sin el pintado entre ellas. Esto lo realiza mediante la llamada al método *renderWire()*.

Entre los últimos apartados, se tienen dos barras de sliders con los cuales puede modificarse los segmentos y stacks de las figuras, que afecta con cuántos triángulos (lo que se traduce a nivel de detalle) la figura va a ser renderizada. Cuando se detecta alguna acción en uno de los sliders, esta se procesa mediante el método *segmentsUpdate()* el cual envía los nuevos valores (seleccionados desde el slider en el

GUI) a la figura. Inmediatamente se vuelve a llamar al método *initData()* de la figura para actualizar los vértices para que coincidan con los segment y stack definidos por el usuario. Finalmente, para evitar un bucle, se establece un booleano bandera *segments_event_listener* en falso, y que solo se tornará verdadero cuando vuelva a recibirse una acción en alguno de los sliders.

ALGORITMO

```
void Cubesphere::initData()
{
   // generate unit-length verties in +X face
   std::vector<float> unitVertices =
Cubesphere::getUnitPositiveX(vertexCountPerRow);

   // clear memory of prev arrays
   clearArrays();

   float x, y, z, s, t;
   int k = 0, k1, k2;

   // build +X face
   for(unsigned int i = 0; i < vertexCountPerRow; ++i)
   {
      k1 = i * vertexCountPerRow;     // index for curr row
      k2 = k1 + vertexCountPerRow;    // index for next row
      t = (float)i / (vertexCountPerRow - 1);

      for(unsigned int j = 0; j < vertexCountPerRow; ++j, k += 3, ++k1, ++k2)
      {
         x = unitVertices[k];
         y = unitVertices[k+1];
         z = unitVertices[k+2];
         s = (float)j / (vertexCountPerRow - 1);
         addVertex(x*radius, y*radius, z*radius);
         addNormal(x, y, z);
         addTexCoord(s, t);

         // add indices
         if(i < (vertexCountPerRow-1) && j < (vertexCountPerRow-1))
         {
            addIndices(k1, k2, k1+1);
            addIndices(k1+1, k2, k2+1);
            // lines: left and top
            lineIndices.push_back(k1);  // left
            lineIndices.push_back(k2);
```

```
            lineIndices.push_back(k1);  // top
            lineIndices.push_back(k1+1);
        }
    }

}

// array size and index for building next face
unsigned int startIndex;                // starting index for next face
int vertexSize = (int)vertices.size();      // vertex array size of +X face
int indexSize = (int)indices.size();        // index array size of +X face
int lineIndexSize = (int)lineIndices.size(); // line index size of +X face

// build -X face by negating x and z
startIndex = vertices.size() / 3;
for(int i = 0, j = 0; i < vertexSize; i += 3, j += 2)
{
   addVertex(-vertices[i], vertices[i+1], -vertices[i+2]);
   addTexCoord(texCoords[j], texCoords[j+1]);
   addNormal(-normals[i], normals[i+1], -normals[i+2]);
}
for(int i = 0; i < indexSize; ++i)
{
   indices.push_back(startIndex + indices[i]);
}
for(int i = 0; i < lineIndexSize; i += 4)
{
   // left and bottom lines
   lineIndices.push_back(startIndex + lineIndices[i]);     // left
   lineIndices.push_back(startIndex + lineIndices[i+1]);
   lineIndices.push_back(startIndex + lineIndices[i+1]);  // bottom
   lineIndices.push_back(startIndex + lineIndices[i+1] + 1);
}

// build +Y face by swapping x=>y, y=>-z, z=>-x
startIndex = vertices.size() / 3;
for(int i = 0, j = 0; i < vertexSize; i += 3, j += 2)
{
   addVertex(-vertices[i+2], vertices[i], -vertices[i+1]);
   addTexCoord(texCoords[j], texCoords[j+1]);
   addNormal(-normals[i+2], normals[i], -normals[i+1]);
}
for(int i = 0; i < indexSize; ++i)
{
   indices.push_back(startIndex + indices[i]);
}
for(int i = 0; i < lineIndexSize; ++i)
{
```

```
    // top and left lines (same as +X)
    lineIndices.push_back(startIndex + lineIndices[i]);
}

// build -Y face by swapping x=>-y, y=>z, z=>-x
startIndex = vertices.size() / 3;
for(int i = 0, j = 0; i < vertexSize; i += 3, j += 2)
{
    addVertex(-vertices[i+2], -vertices[i], vertices[i+1]);
    addTexCoord(texCoords[j], texCoords[j+1]);
    addNormal(-normals[i+2], -normals[i], normals[i+1]);
}
for(int i = 0; i < indexSize; ++i)
{
    indices.push_back(startIndex + indices[i]);
}
for(int i = 0; i < lineIndexSize; i += 4)
{
    // top and right lines
    lineIndices.push_back(startIndex + lineIndices[i]); // top
    lineIndices.push_back(startIndex + lineIndices[i+3]);
    lineIndices.push_back(startIndex + lineIndices[i] + 1); // right
    lineIndices.push_back(startIndex + lineIndices[i+1] + 1);
}

// build +Z face by swapping x=>z, z=>-x
startIndex = vertices.size() / 3;
for(int i = 0, j = 0; i < vertexSize; i += 3, j += 2)
{
    addVertex(-vertices[i+2], vertices[i+1], vertices[i]);
    addTexCoord(texCoords[j], texCoords[j+1]);
    addNormal(-normals[i+2], normals[i+1], normals[i]);
}
for(int i = 0; i < indexSize; ++i)
{
    indices.push_back(startIndex + indices[i]);
}
for(int i = 0; i < lineIndexSize; ++i)
{
    // top and left lines (same as +X)
    lineIndices.push_back(startIndex + lineIndices[i]);
}

// build -Z face by swapping x=>-z, z=>x
startIndex = vertices.size() / 3;
for(int i = 0, j = 0; i < vertexSize; i += 3, j += 2)
{
    addVertex(vertices[i+2], vertices[i+1], -vertices[i]);
```

```
        addTexCoord(texCoords[j], texCoords[j+1]);
        addNormal(normals[i+2], normals[i+1], -normals[i]);
    }
    for(int i = 0; i < indexSize; ++i)
    {
        indices.push_back(startIndex + indices[i]);
    }
    for(int i = 0; i < lineIndexSize; i += 4)
    {
        // left and bottom lines
        lineIndices.push_back(startIndex + lineIndices[i]);     // left
        lineIndices.push_back(startIndex + lineIndices[i+1]);
        lineIndices.push_back(startIndex + lineIndices[i+1]);   // bottom
        lineIndices.push_back(startIndex + lineIndices[i+1] + 1);
    }

    // build vertices to paint normals
    float long_linea = 0.02f;
    for (int i = 0; i < normals.size(); i+=3) {
        norm_lines.push_back(vertices[i]);
        norm_lines.push_back(vertices[i+1]);
        norm_lines.push_back(vertices[i+2]);

        norm_lines.push_back(vertices[i] + normals[i] * long_linea);
        norm_lines.push_back(vertices[i+1] + normals[i+1] * long_linea);
        norm_lines.push_back(vertices[i+2] + normals[i+2] * long_linea);
    }

    glBindVertexArray(vao);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_vert);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(float),
&vertices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_norm);
    glBufferData(GL_ARRAY_BUFFER, normals.size() * sizeof(float),
&normals[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indi);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indi_lines);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, lineIndices.size() *
sizeof(unsigned int), &lineIndices[0], GL_STATIC_DRAW);

    glGenVertexArrays(1, &vao_2);
    glBindVertexArray(vao_2);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, vbo_norm_lines);
    glBufferData(GL_ARRAY_BUFFER, norm_lines.size() * sizeof(float),
&norm_lines[0], GL_STATIC_DRAW);
}
```

CONO:

```
void Cylinder::buildUnitCircleVertices()
{
    const float PI = acos(-1);
    float sectorStep = 2 * PI / sectorCount;
    float sectorAngle;  // radian

    std::vector<float>().swap(unitCircleVertices);
    for(int i = 0; i <= sectorCount; ++i)
    {
        sectorAngle = i * sectorStep;
        unitCircleVertices.push_back(cos(sectorAngle)); // x
        unitCircleVertices.push_back(sin(sectorAngle)); // y
        unitCircleVertices.push_back(0);                // z
    }
}
```

- Este método nos calcula la cantidad de vértices que tendrá el círculo que será la base de la figura. Estos se generan mediante la cantidad de segments que están definidos en la interfaz gráfica.

```
for(int i = 0; i <= stackCount; ++i)
{
    z = -(height * 0.5f) + (float)i / stackCount * height;      // vertex positic
    radius = baseRadius + (float)i / stackCount * (topRadius - baseRadius);     /
    float t = 1.0f - (float)i / stackCount;   // top-to-bottom

    for(int j = 0, k = 0; j <= sectorCount; ++j, k += 3)
    {
        x = unitCircleVertices[k];
        y = unitCircleVertices[k+1];
        addVertex(x * radius, y * radius, z);   // position
        addNormal(sideNormals[k], sideNormals[k+1], sideNormals[k+2]); // normal
        addTexCoord((float)j / sectorCount, t); // tex coord
    }
}
```

- Este doble for se encarga de añadir las coordenadas de los vértices del lado del cilindro al arreglo. Estas coordenadas se calculan en base a la altura y el radio que el usuario define en la interfaz gráfica.

```
z = -height * 0.5f;
addVertex(0, 0, z);
addNormal(0, 0, -1);
addTexCoord(0.5f, 0.5f);
for(int i = 0, j = 0; i < sectorCount; ++i, j += 3)
{
    x = unitCircleVertices[j];
    y = unitCircleVertices[j+1];
    addVertex(x * baseRadius, y * baseRadius, z);
    addNormal(0, 0, -1);
    addTexCoord(-x * 0.5f + 0.5f, -y * 0.5f + 0.5f);
}
```

- Este algoritmo permite agregar las coordenadas de la base de nuestra figura, las cuales se calculan en base a la cantidad de vértices que se calcularon con anterioridad.

```
z = height * 0.5f;
addVertex(0, 0, z);
addNormal(0, 0, 1);
addTexCoord(0.5f, 0.5f);
for(int i = 0, j = 0; i < sectorCount; ++i, j += 3)
{
    x = unitCircleVertices[j];
    y = unitCircleVertices[j+1];
    addVertex(x * topRadius, y * topRadius, z);
    addNormal(0, 0, 1);
    addTexCoord(x * 0.5f + 0.5f, -y * 0.5f + 0.5f);
}
```

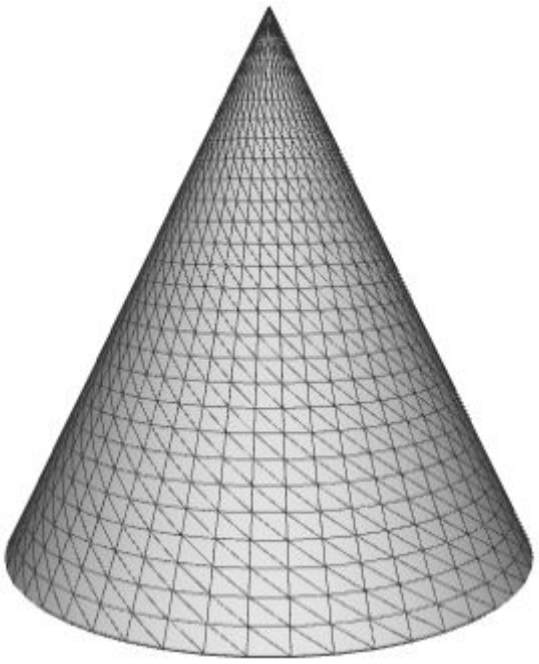
- Tiene el mismo funcionamiento que el algoritmo anterior, solo que este se encarga de agregar las coordenadas de la base superior.

```
float long_linea = 0.012f;
for (int i = 0; i < normals.size(); i+=3) {
    norm_lines.push_back(vertices[i]);
    norm_lines.push_back(vertices[i+1]);
    norm_lines.push_back(vertices[i+2]);

    norm_lines.push_back(vertices[i] + normals[i] * long_linea);
    norm_lines.push_back(vertices[i+1] + normals[i+1] * long_linea);
    norm_lines.push_back(vertices[i+2] + normals[i+2] * long_linea);
}
```

- Se encarga de construir los vértices para pintar las normales en pantalla. Estos se calculan en base a los vértices del cilindro que previamente fueron calculados.

El cono sigue la misma lógica que el cilindro, con la única diferencia de que una de sus dos bases tiene un radio igual a 0, esto nos genera un único punto en el plano XY. Por este motivo todos los puntos de la otra base circular se unen con el punto anteriormente generado.



ESFERA:

Primero debemos instanciar un objeto de la clase Sphere, que hereda de la clase Shape. Este objeto cuenta con un método constructor de la forma:

Sphere (float radius, int sectors, int stacks, bool smooth)

Instanciamos un objeto, al cual llamaremos sphereShape, el cual iniciaremos con los parámetros:

- radius = 0.1
- sectors = 10
- stacks = 10
- smooth = true

Luego, llamamos al método *createVertexObjects()* para generar los VAO, VBOs y EBOs necesarios para renderizar la figura seguido del método *initData()* que calculará los vértices, puntos, normales e índices con los que se llenarán el VAO, VBOs y EBOs.

Lo siguiente es definido por la interacción de la aplicación con el usuario a través de la interfaz gráfica y es ejecutado dentro del bucle principal de la aplicación (es

decir, mientras no se haya enviado una señal de cierre de ventana de la aplicación).

- Color de fondo y de figura.
- Figura.
- Shaders.
- Relleno.
- Normales.
- Wires.
- Segments.
- Propiedades de figura.

Lo primero que se nos presenta en la interfaz gráfica es un selector de color para cambiar el color de nuestra figura (si es que hemos seleccionado la *checkbox* correspondiente a cualquiera de ellas) y el fondo. Los cambios que realicemos serán renderizados inmediatamente y a tiempo real en el programa.

Lo siguiente que se muestra es la lista de *checkboxes* con las diferentes figuras disponibles que podemos renderizar y visualizar en el programa. Podemos seleccionar una o varias (incluso todas, si así se desea), cada una seguirá las mismas propiedades que establezcamos por medio de la GUI. A su vez, al momento de seleccionarla, se nos presentará una pequeña ventana mediante la cual podemos modificar las propiedades de cada figura (dependiendo de cuál sea esta). Al ser una esfera, podemos modificar solo un parámetro:

- radio

Al modificar cualquiera de estos parámetros, el listener de las propiedades de la figura detecta la modificación y procede a procesarla mediante el método *setProp()*, el cual accede al setter de la figura y actualiza sus puntos, vértices, etc mediante el método *initData()* coincidiendo con los nuevos valores introducidos por el usuario a través de las modificaciones en la GUI. Para evitar un bucle infinito, se establece un booleano bandera *prop_listener* a falso, que se tornará verdadero cuando se realice otra modificación en las propiedades de la figura.

Si seleccionamos la casilla de Shaders, se nos presentará una pequeña ventana que nos permitirá elegir entre tipos de shaders que deseamos ver en la figura renderizada. Estos son:

- Ordinario.
- BlingPong.
- Goraud.
- Phong.

Lo siguiente es el Relleno. La *checkbox* correspondiente nombra a esta opción como *Fill*. Si la marcamos, el programa llamará al método *renderFill()* de la figura (o figuras) que estemos renderizando en ese momento, pintándola con el color elegido en el selector de color.

El programa también nos ofrece la posibilidad de renderizar las normales por cada figura. Si se ha seleccionado la casilla *Normals*, se llamará al método *renderNormals()*.

Otra opción disponible por medio de la GUI es Render wire, la cual (si es que ha sido seleccionada), renderizará las líneas asociadas a los vértices de la figura sin el pintado entre ellas. Esto lo realiza mediante la llamada al método *renderWire()*.
Entre los últimos apartados, se tienen dos barras de sliders con los cuales puede modificarse los segmentos y stacks de las figuras, que afecta con cuántos triángulos (lo que se traduce a nivel de detalle) la figura va a ser renderizada. Cuando se detecta alguna acción en uno de los sliders, esta se procesa mediante el método segmentsUpdate() el cual envía los nuevos valores (seleccionados desde el slider en el GUI) a la figura. Inmediatamente se vuelve a llamar al método initData() de la figura para actualizar los vértices para que coincidan con los segment y stack definidos por el usuario. Finalmente, para evitar un bucle, se establece un booleano bandera segments_event_listener en falso, y que solo se tornará verdadero cuando vuelva a recibirse una acción en alguno de los sliders.

ALGORITMO UTILIZADO

```
void Sphere::initData() {

	// x = r * cos(u) * cos(v)
	// y = r * cos(u) * sin(v)
	// z = r * sin(u)
	// where u: stack(latitude) angle (-90 <= u <= 90)
	//       v: sector(longitude) angle (0 <= v <= 360)

	const float PI = acos(-1);

	// clear memory of prev arrays
	clearArrays();

	float x, y, z, xy;                              // vertex position
	float nx, ny, nz, lengthInv = 1.0f / radius;    // normal
	float s, t;                                     // texCoord

	float sectorStep = 2 * PI / sectorCount;
	float stackStep = PI / stackCount;
```

```
        float sectorAngle, stackAngle;

        for(int i = 0; i <= stackCount; ++i)
        {
        stackAngle = PI / 2 - i * stackStep;        // starting from pi/2 to -pi/2
        xy = radius * cosf(stackAngle);             // r * cos(u)
        z = radius * sinf(stackAngle);              // r * sin(u)

        // add (sectorCount+1) vertices per stack
        // the first and last vertices have same position and normal, but different
tex coords
        for(int j = 0; j <= sectorCount; ++j)
        {
        sectorAngle = j * sectorStep;       // starting from 0 to 2pi

        // vertex position
        x = xy * cosf(sectorAngle);         // r * cos(u) * cos(v)
        y = xy * sinf(sectorAngle);         // r * cos(u) * sin(v)
        addVertex(x, y, z);

        // normalized vertex normal
        nx = x * lengthInv;
        ny = y * lengthInv;
        nz = z * lengthInv;
        addNormal(nx, ny, nz);

        // vertex tex coord between [0, 1]
        s = (float)j / sectorCount;
        t = (float)i / stackCount;
        addTexCoord(s, t);
        }
        }

        // indices
        //  k1--k1+1
        //  |  / |
        //  | /  |
        //  k2--k2+1
        unsigned int k1, k2;
        for(int i = 0; i < stackCount; ++i) {
        k1 = i * (sectorCount + 1);  // beginning of current stack
        k2 = k1 + sectorCount + 1;          // beginning of next stack

        for(int j = 0; j < sectorCount; ++j, ++k1, ++k2) {
        // 2 triangles per sector excluding 1st and last stacks
        if(i != 0)
        {
                addIndices(k1, k2, k1+1);   // k1---k2---k1+1
```

```
        }

        if(i != (stackCount-1))
        {
                addIndices(k1+1, k2, k2+1); // k1+1---k2---k2+1
        }

        // vertical lines for all stacks
        lineIndices.push_back(k1);
        lineIndices.push_back(k2);
        if(i != 0)  // horizontal lines except 1st stack
        {
                lineIndices.push_back(k1);
                lineIndices.push_back(k1 + 1);
        }
        }
        }

        // build vertices to paint normals
        float long_linea = 0.012f;
        for (int i = 0; i < normals.size(); i+=3) {
        norm_lines.push_back(vertices[i]);
        norm_lines.push_back(vertices[i+1]);
        norm_lines.push_back(vertices[i+2]);

        norm_lines.push_back(vertices[i] + normals[i] * long_linea);
        norm_lines.push_back(vertices[i+1] + normals[i+1] * long_linea);
        norm_lines.push_back(vertices[i+2] + normals[i+2] * long_linea);
        }

        glBindVertexArray(vao);

        glBindBuffer(GL_ARRAY_BUFFER, vbo_vert);
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(float),
&vertices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, vbo_norm);
        glBufferData(GL_ARRAY_BUFFER, normals.size() * sizeof(float),
&normals[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indi);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_indi_lines);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, lineIndices.size() *
sizeof(unsigned int), &lineIndices[0], GL_STATIC_DRAW);
```

```
        glBindVertexArray(vao_2);

        glBindBuffer(GL_ARRAY_BUFFER, vbo_norm_lines);
        glBufferData(GL_ARRAY_BUFFER, norm_lines.size() * sizeof(float),
&norm_lines[0], GL_STATIC_DRAW);
}
```

b. Librerías:

i. **Dear ImGUI:** Es una librería de interfaz gráfica de usuario para C++. Genera buffers de vértices optimizados que se pueden renderizar en cualquier momento en la aplicación. Es rápida. portable y autocontenido (sin dependencias externas).

ii. **OpenGL Mathematics (GLM):** Es una librería matemática para C++ usada en software gráfico bajo la especificación GLSL. Nos provee de clases y funciones implementadas con la misma nomenclatura que GLSL; sin embargo, no se basa solo en esta convención, también provee de funcionalidades extras como: transformación de matrices, empaquetamiento de datos, etc.

iii. **IrrKlang:** Para la reproducción de files en .mp3 en c + +.

c. Herramientas:

i. **Eclipse:** Es IDE de código abierto multiplataforma con soporte a múltiples lenguajes de programación como Java, C/C++, etc. Además, cuenta con plugins que nos permiten expandir las características y funcionalidades de la plataforma.

ii. **C + +**: Es un lenguaje de programación multiparadigma de alto nivel que nos permite un mayor control sobre la memoria de la computadora en comparación a otros. Los programas escritos en este lenguaje tiene una mayor velocidad de ejecución ya que el código es compilado directamente a binarios.

iii. **StarUML:** Es una herramienta de modelado de software basado en los estándares UML y MDA. Nos permite realizar diagramas de casos de uso, de clases, de secuencias, etc.

iv. **Doxygen:** Para la generación de los diagramas de clases y colaboración de manera automática.

v. **Github y git:** Para trabajar de manera colaborativamente y tener un control de versiones en remoto y local.

d. Técnicas:

i. Primitivas: GL_TRIANGLE_STRIP, GL_LINES,etc. Para el renderizado de las figuras.

ii. Gestión de vaos, vbos y ebos.

iii. Uso de cámaras.

iv. Sombreado de Phong: Utiliza 3 componentes básicos:

1. Ambiente: Luz que llega rebotada de algún objeto y se refleja en todas las direcciones simultáneamente.
2. Difusa: Luz que llega desde la fuente directamente pero luego va hacia todas las direcciones, sumándole la luz ambiental nos dará el color del objeto.
3. Especular: Llega directamente desde la fuente de luz y rebota en una dirección, de acuerdo a la normal de la superficie. Afectará al brillo de esa superficie.

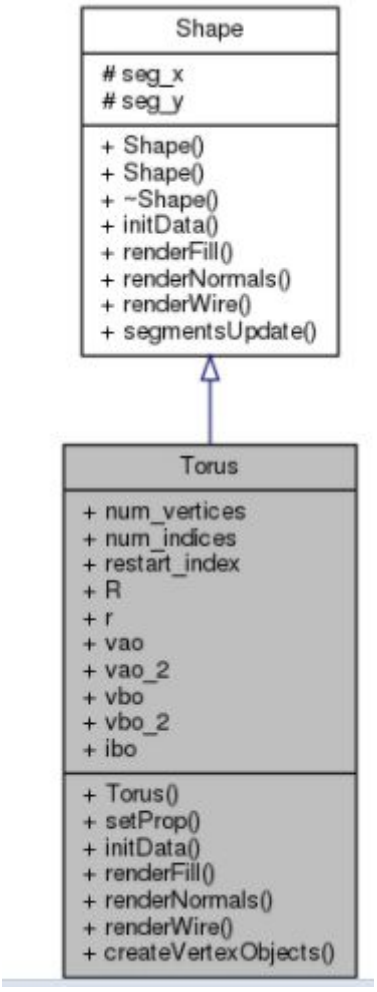
e. Documentación con Doxygen:

Toda la documentación se encuentra en este link. Cuenta con diagramas de clases y diagramas de cooperación:
https://darkblood202.github.io/CG-Proyecto-doc.

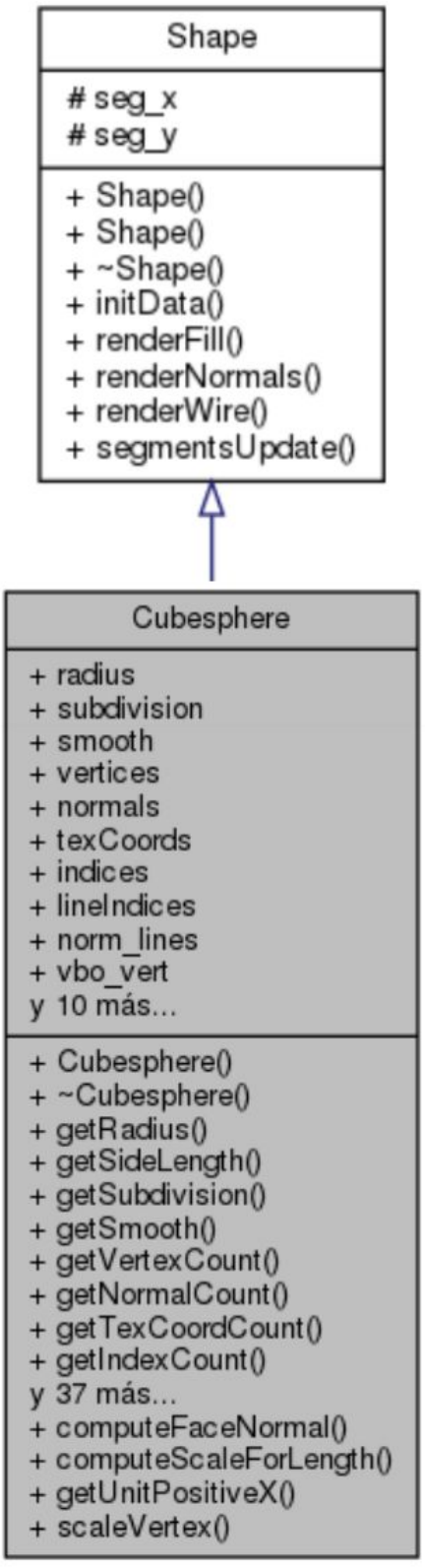
3.Metodología de Desarrollo.

a. Diagrama de Clases.

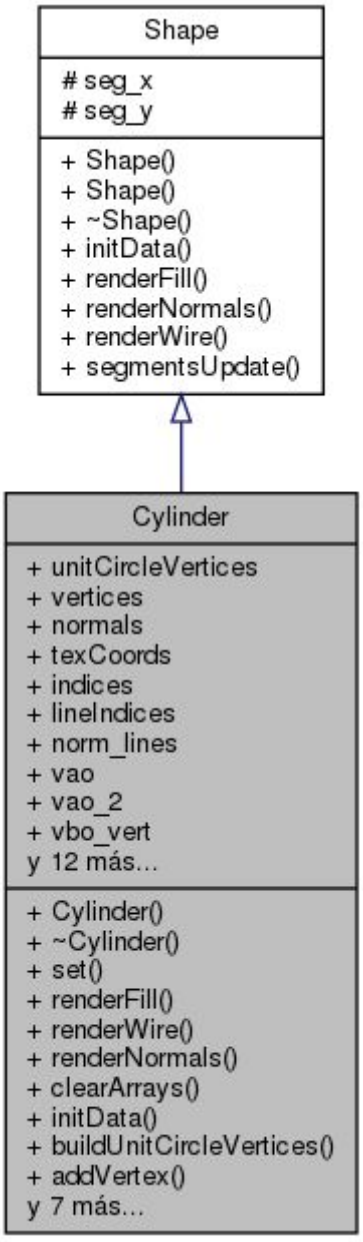
TORUS:



CUBO:


Shape
seg_x
seg_y
+ Shape()
+ Shape()
+ ~Shape()
+ initData()
+ renderFill()
+ renderNormals()
+ renderWire()
+ segmentsUpdate()
Cubesphere
+ radius
+ subdivision
+ smooth
+ vertices
+ normals
+ texCoords
+ indices
+ lineIndices
+ norm_lines
+ vbo_vert
y 10 más...
+ Cubesphere()
+ ~Cubesphere()
+ getRadius()
+ getSideLength()
+ getSubdivision()
+ getSmooth()
+ getVertexCount()
+ getNormalCount()
+ getTexCoordCount()
+ getIndexCount()
y 37 más...
+ computeFaceNormal()
+ computeScaleForLength()
+ getUnitPositiveX()
+ scaleVertex()

CILINDRO Y CONO:



ESFERA:

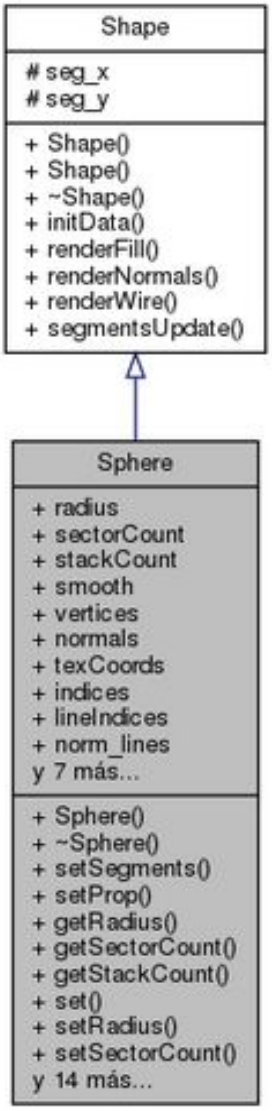
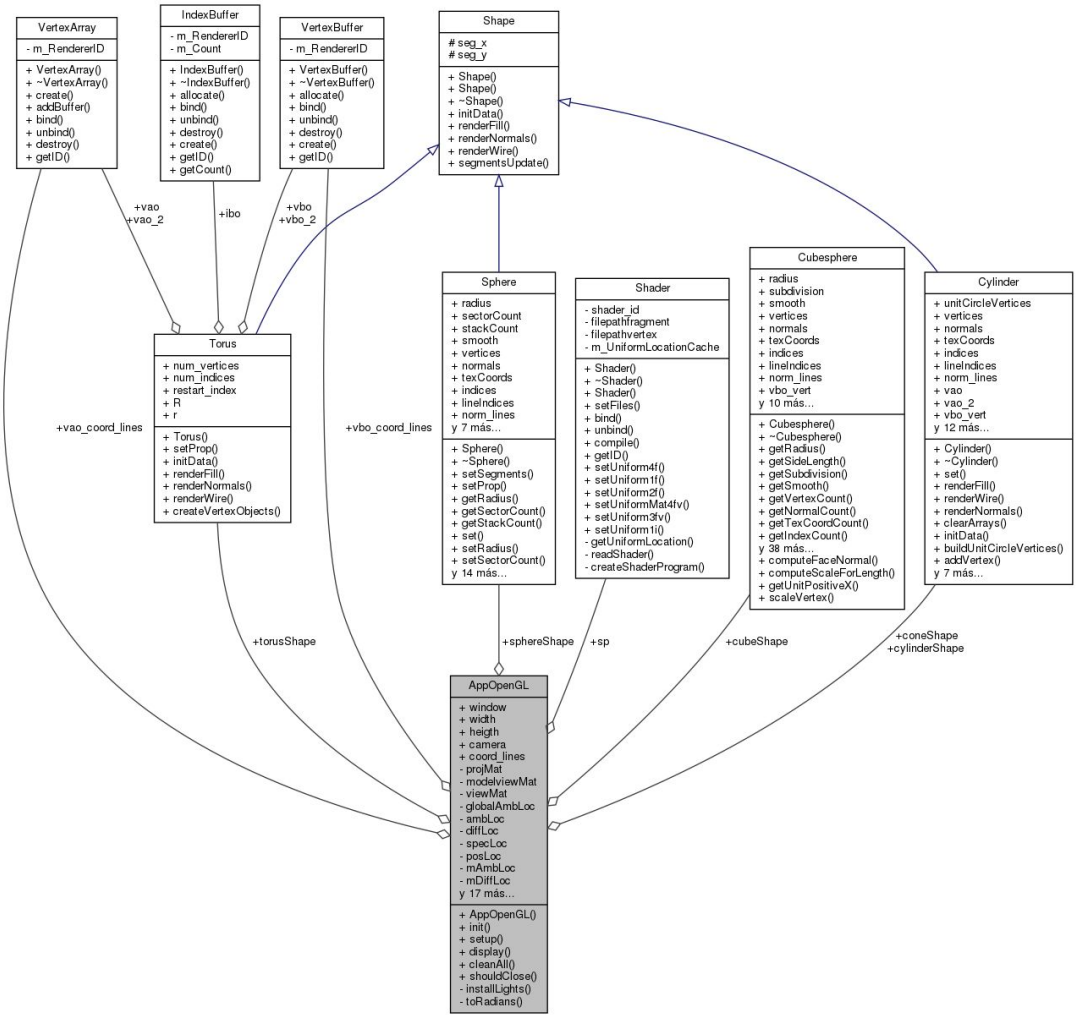
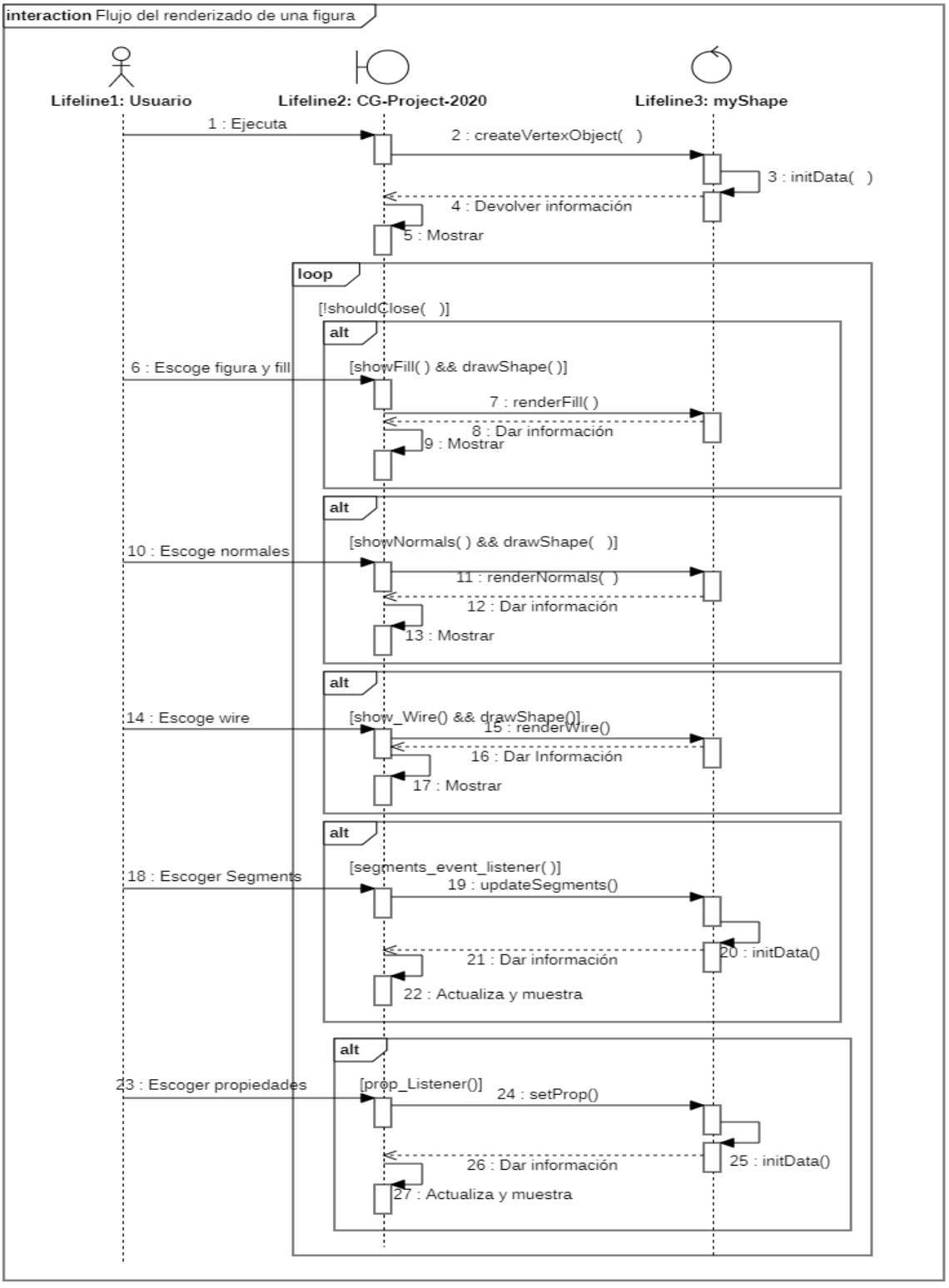

Shape
seg_x
seg_y
+ Shape()
+ Shape()
+ ~Shape()
+ initData()
+ renderFill()
+ renderNormals()
+ renderWire()
+ segmentsUpdate()
Sphere
+ radius
+ sectorCount
+ stackCount
+ smooth
+ vertices
+ normals
+ texCoords
+ indices
+ lineIndices
+ norm_lines
y 7 más...
+ Sphere()
+ ~Sphere()
+ setSegments()
+ setProp()
+ getRadius()
+ getSectorCount()
+ getStackCount()
+ set()
+ setRadius()
+ setSectorCount()
y 14 más...

DIAGRAMA GENERAL:

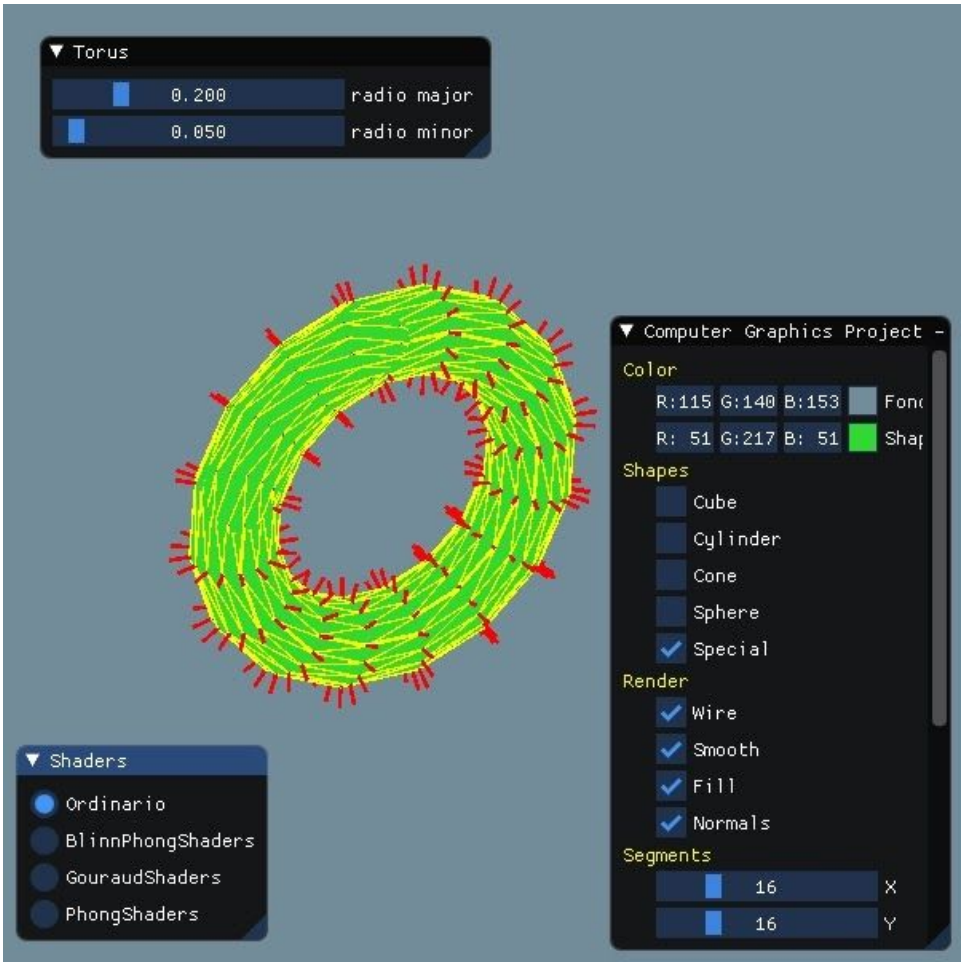


b. Diagrama de Secuencia General de una figura.

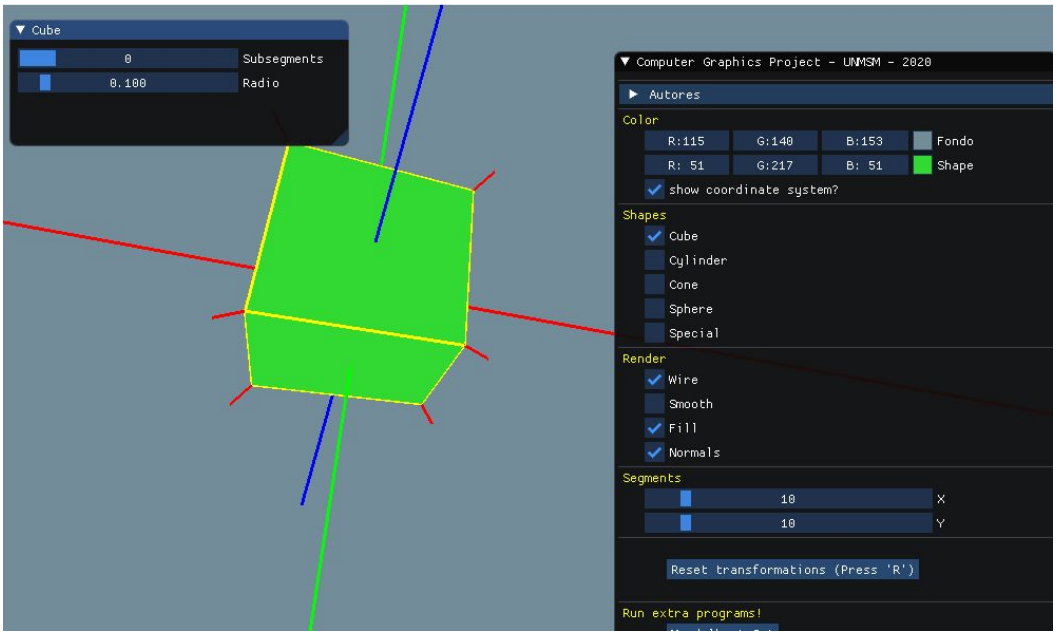


c. Prototipos Mockups de la aplicación.

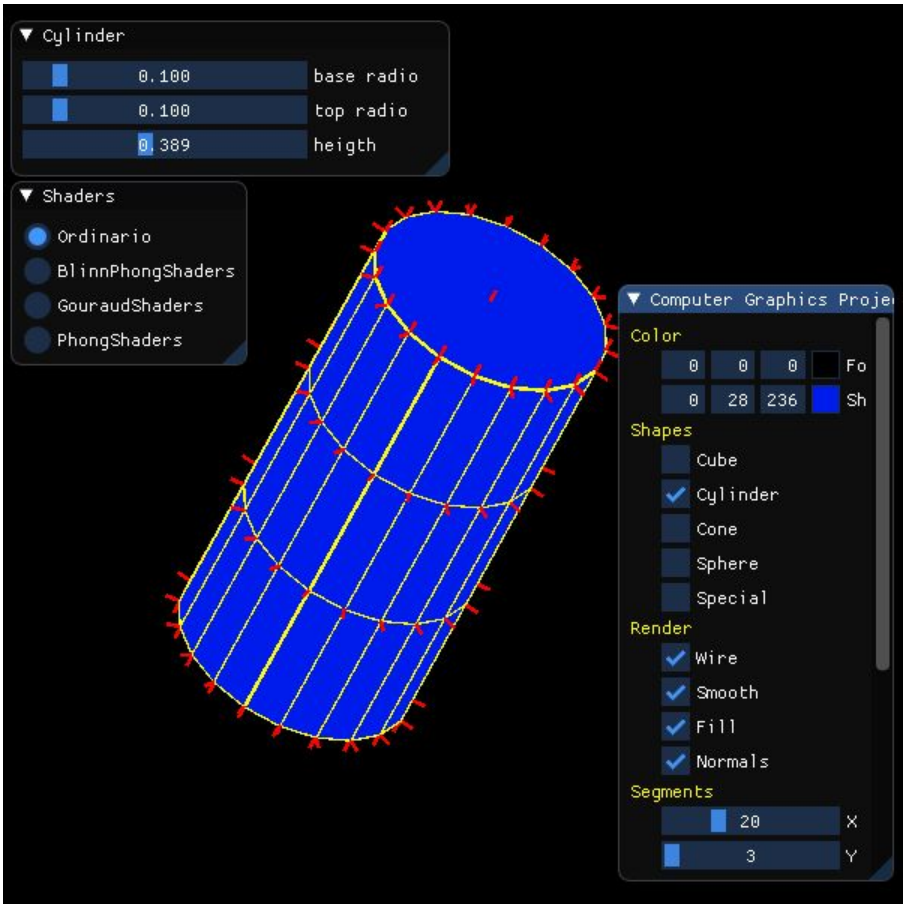
TORUS:



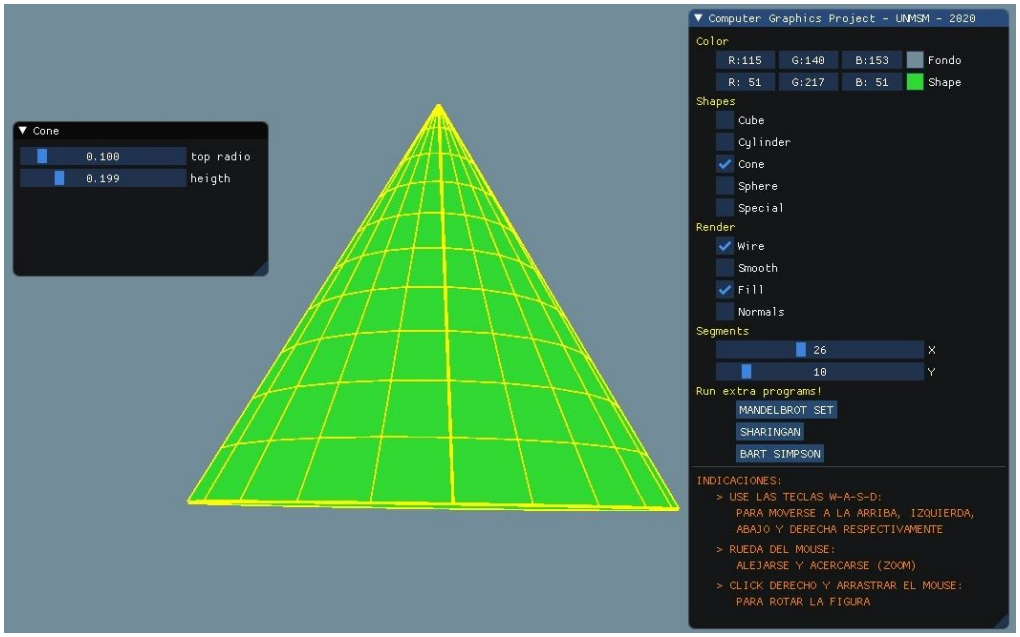
CUBO:



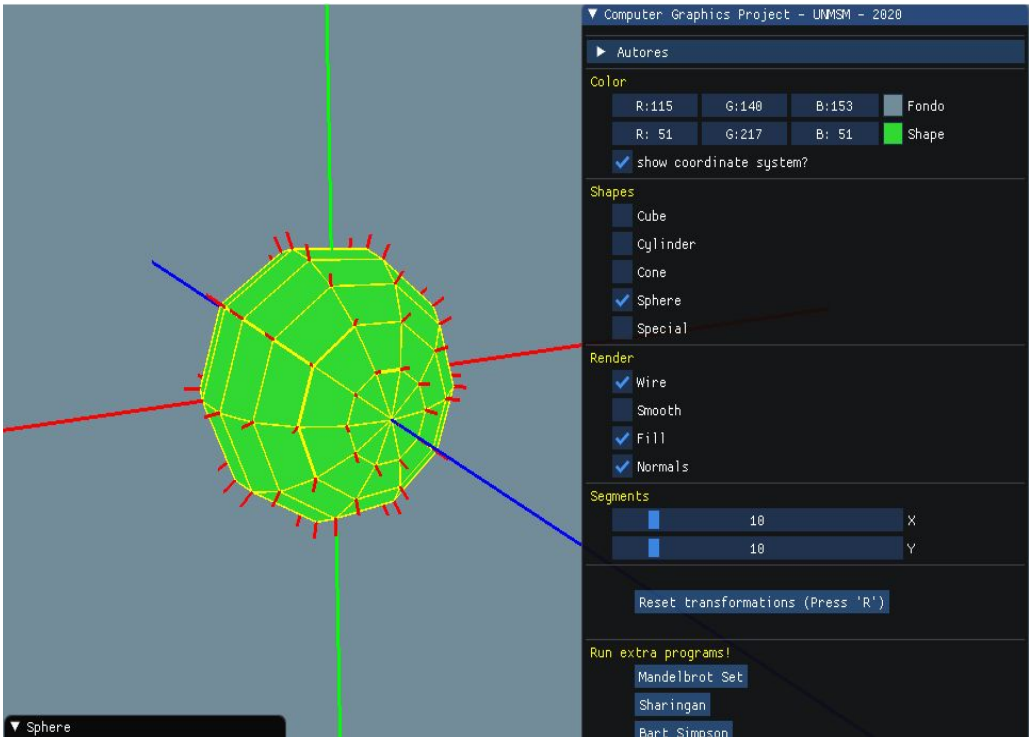
CILINDRO:



CONO:


Cone
0.100
top radio
0.199
heigth
Computer Graphics Project - UNMSM - 2020
Color
R:115
G:140
B:153
Fondo
R: 51
G:217
B: 51
Shape
Shapes
Cube
Cylinder
Cone
Sphere
Special
Render
Wire
Smooth
Fill
Normals
Segments
26
X
10
Y
Run extra programs!
MANDELBROT SET
SHARINGAN
BART SIMPSON
INDICACIONES:
> USE LAS TECLAS W-A-S-D:
PARA MOVERSE A LA ARRIBA, IZQUIERDA,
ABAJO Y DERECHA RESPECTIVAMENTE
> RUEDA DEL MOUSE:
ALEJARSE Y ACERCARSE (ZOOM)
> CLICK DERECHO Y ARRASTRAR EL MOUSE:
PARA ROTAR LA FIGURA

ESFERA:


Computer Graphics Project - UNMSM - 2020
Autores
Color
R:115
G:140
B:153
Fondo
R: 51
G:217
B: 51
Shape
show coordinate system?
Shapes
Cube
Cylinder
Cone
Sphere
Special
Render
Wire
Smooth
Fill
Normals
Segments
10
X
10
Y
Reset transformations (Press 'R')
Run extra programs!
Mandelbrot Set
Sharingan
Bart Simpson
Sphere

4.Aporte de los integrantes.

ROGER:

- Generación de vértices del Torus y el método renderFill(): Ambos son para poder generar la figura.
- Smooth de vértices del Torus
- Iluminación de vértices del Torus: Ordinario, BlingFound ,Phong,etc.
- Aplicaciones extra (Mandelbrot Set, Sharingan)

SANTIAGO:

- Wiring de vértices del Torus: Puntos que conforman a la figura.
- Normales de vértices del Torus: Líneas rojas de la aplicación(normales).
- Irrklang: Para la reproducción de música, etc.

SEBASTIAN:

- Implementación de la clase Cilindro heredada de la clase Shapes.
- Algoritmo de generación de puntos de la clase Cilindro.
- Renderizado de una instancia de la clase Cilindro en el main loop (wire, rellenado, smooth y normales).

JONATHAN

- Implementación de la clase Cubesphere heredada de la clase Shapes.
- Renderizado de una instancia de la clase Cubesphere junto a sus métodos respectivos para las normales, el rellenado y el wire.
- Smooth de vértices de la Esfera.
- Trabajo conjunto para la adaptabilidad al modelo de trabajo con clases y módulos.

JHONEL:

- Implementación de la instancia Cono que hereda de Cilindro.
- Widget de la GUI para poder modificar el radio de la base y la altura del Cono.
- Renderizado de la instancia del Cono junto con sus respectivos métodos para el wire, rellenado y normales.

OMAR

- Implementación de la clase Esfera heredada de la clase Shapes.
- Renderizado de una instancia de la clase Esfera junto a sus métodos respectivos para las normales, el rellenado y el wire.

TODOS:

- Manejo de clases.
- Manejo de Imgui
- Generación de documentación vía Doxygen.
- Aplicaciones extra (Bart Simpson)

5. Conclusiones y Recomendaciones.

- La programación orientada a objetos permite organizar mucho mejor el código, junto a la programación modular permiten un mejor manejo y seguimiento del mismo, pues lo mantiene simple, conciso y claro.
- La herramienta imGUI utilizada en esta aplicación demuestra lo útil que es manipular el código en marcha, lo cual brinda un sinfín de posibilidades, sobre todo en un mundo que palpa más cercana la realidad virtual.
- Reutilizar código es una muy buena práctica, un claro ejemplo lo fue al notar que el modificar un solo parámetro de la clase "cilindro" genera un cono.

6. Referencias.

a. Ahn, S. H. (s. f.-a). OpenGL Cylinder, Prism & Pipe. Recuperado 5 de septiembre de 2020, de http://www.songho.ca/opengl/gl_cylinder.html
b. Ahn, S. H. (s. f.-b). OpenGL Sphere. Recuperado 5 de septiembre de 2020, de http://www.songho.ca/opengl/gl_sphere.html
c. Herramientas case Star UML. (s. f.). Recuperado de https://sites.google.com/site/herramientascasestaruml/
d. Indexed Rendering Torus. (s. f.). Recuperado 5 de septiembre de 2020, de http://www.mbsoftworks.sk/tutorials/opengl4/011-indexed-rendering-torus/
e. Cornut O. (s. f.). Dear ImGUI. Recuperado de https://github.com/ocornut/imgui
f. OpenGL Mathematics. (s. f.). Recuperado de https://glm.g-truc.net/0.9.9/index.html
g. Van Heesch D. (2018). Doxygen. Recuperado de https://www.doxygen.nl/index.html