Iluminación y mallas

Auxiliar N°6

CC3501 – Modelación y Computación Gráfica para Ingenieros

Pablo Pizarro R. @ppizarror.com

Contenidos de hoy

- Generación de superficies con curvas
- Repaso concepto de cámara
- Iluminación
- Mallas

1. Generación de superficies con curvas

Superficies con curvas

- Existen varios métodos para poder generar superficies curvas:
  - Usando splines
  - Sólidos de revolución
  - A mano ☹, ingresando vértice a vértice los puntos

- Todas estas metodologías se basan en el mismo principio:
  - **1)** Generar los vértices $v_i = (x_i, y_i, z_i)$
  - **2)** Conectar los vértices generados de manera ordenada usando triángulos y agrupándolos en *Shapes*

Superficies con splines

- Ejemplo con curvas de Catmull-Rom:
  - Primero se genera la lista de vértices $P_i = (x_i, y_i, z_i)$

$$\begin{bmatrix} P_i & P_{i+1} & T_i & T_{i+1} \end{bmatrix} = \begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \begin{bmatrix} 0 & 0 & -\frac{1}{2} & 0 \\ 1 & 0 & 0 & -\frac{1}{2} \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

  - Por último se puede generar la superficies de dos maneras:
    - 1) Generando la superficie triangulando el plano que generen estos puntos
    - 2) Conectando todos los puntos de la superficie con otro externo, denominado centro

Superficies con splines

- Ejemplo con curvas de Catmull-Rom:
  - Primero se genera la lista de vértices $P_i = (x_i, y_i, z_i)$

$$\begin{bmatrix} P_i & P_{i+1} & T_i & T_{i+1} \end{bmatrix} = \begin{bmatrix} P_{i-1} & P_i & P_{i+1} & P_{i+2} \end{bmatrix} \begin{bmatrix} 0 & 0 & -\frac{1}{2} & 0 \\ 1 & 0 & 0 & -\frac{1}{2} \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

  - Por último se puede generar la superficies de dos maneras:
    - 1) Generando la superficie triangulando el plano que generen estos puntos
    - 2) Conectando todos los puntos de la superficie con otro externo, denominado centro

  - A manera de ayuda se programó la función *createColorPlaneFromCurve(curve, triangulate, r, g, b, center)*

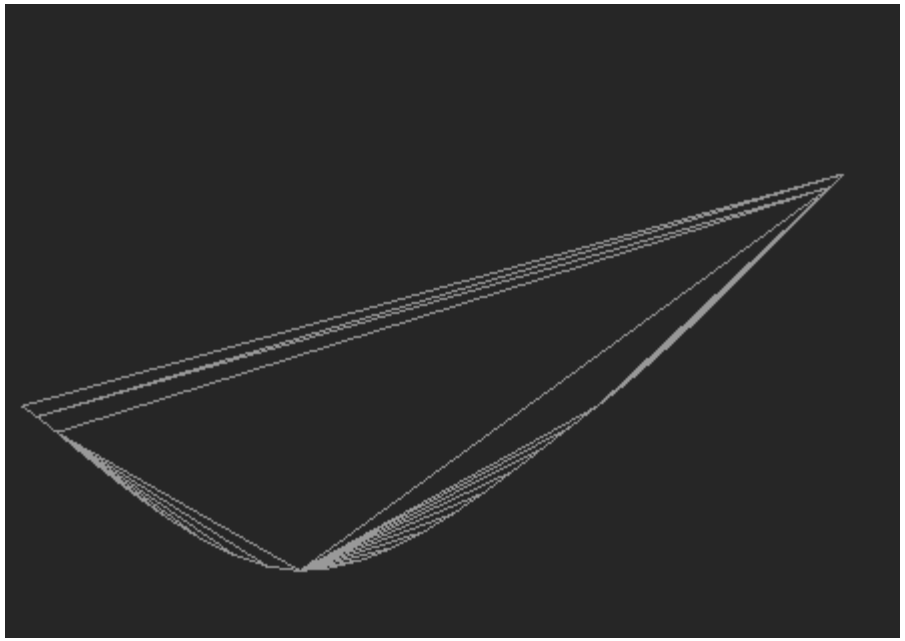
Superficies con splines

- Ejemplo con curvas de Catmull-Rom: Generando la superficie triangulando el plano que generen estos puntos. Se puede usar triangulación de Delaunay u otros.

```
# Create plane using triangulation
obj_planeT = bs_ext.createColorPlaneFromCurve(curve, True, 0.6, 0.6, 0.6)
obj_planeT.setShader(colorShaderProgram)
obj_planeT.translate(ty=0.25)
```

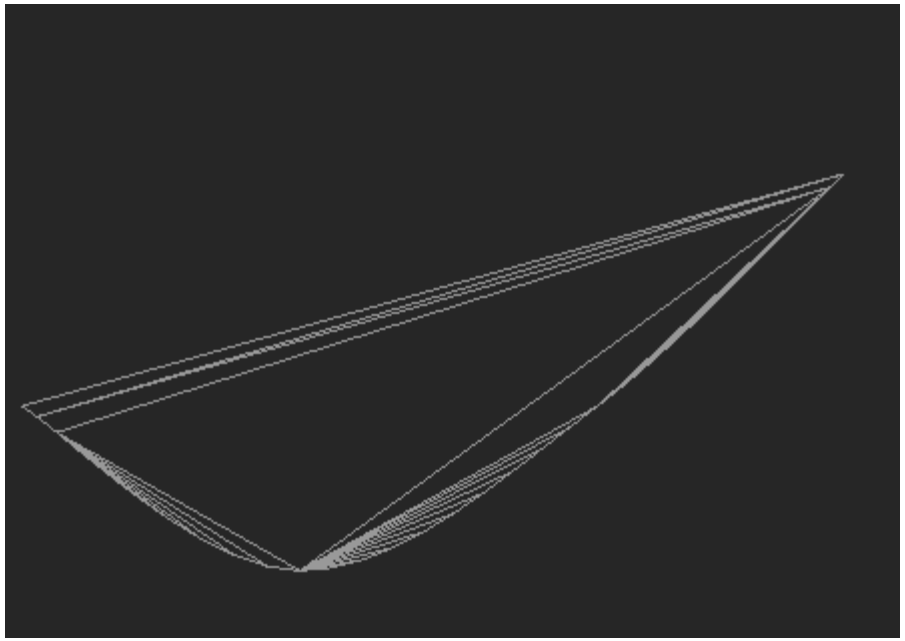
Superficies con splines

- Ejemplo con curvas de Catmull-Rom: Generando la superficie triangulando el plano que generen estos puntos. Se puede usar triangulación de Delaunay u otros.



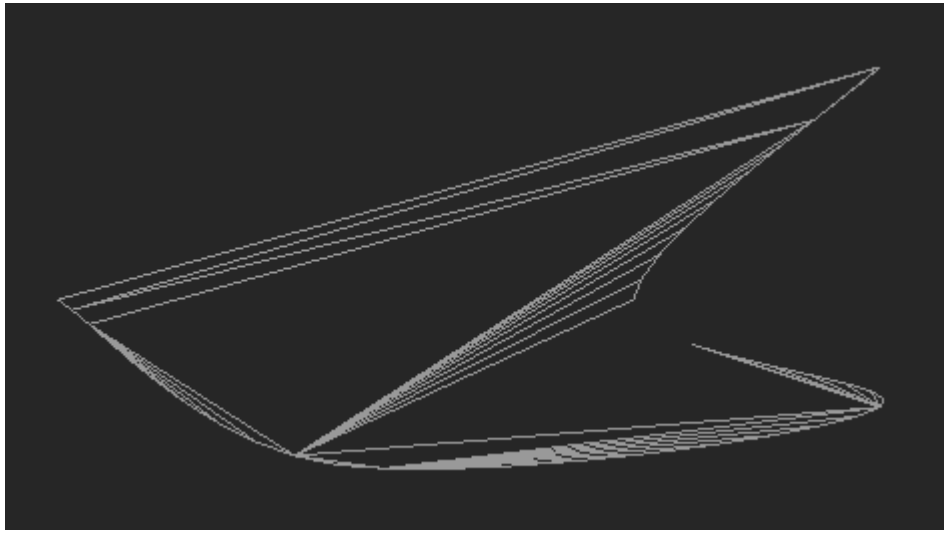
Superficies con splines

- Ejemplo con curvas de Catmull-Rom: Generando la superficie triangulando el plano que generen estos puntos. Se puede usar triangulación de Delaunay u otros.



Superficies con splines

- Ejemplo con curvas de Catmull-Rom: Generando la superficie triangulando el plano que generen estos puntos. ¿Qué pasa en el caso de figuras no convexas?



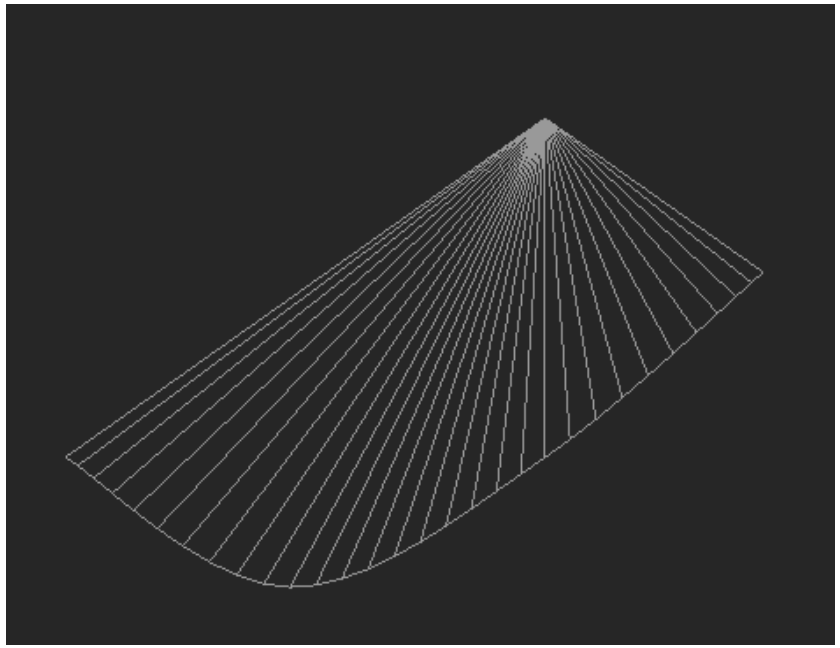
Superficies con splines

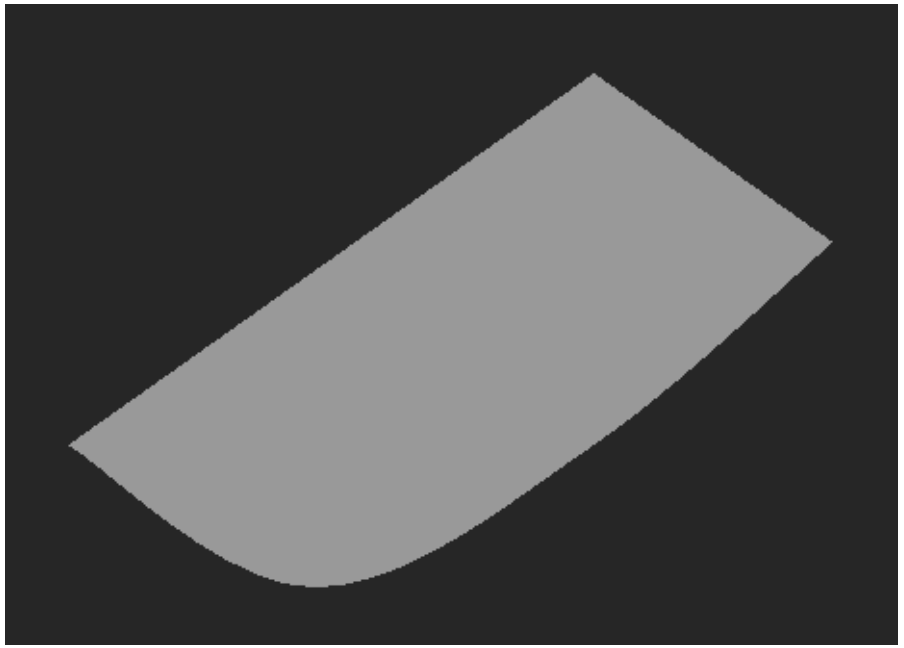
- Ejemplo con curvas de Catmull-Rom: Generando la superficie conectando cada par de puntos P_i, P_{i+1} con un centro C_i, así cada triángulo queda definido por (P_i, P_{i+1}, C_i)

```
# Create plane
vertices = [[1, 0], [0.9, 0.4], [0.5, 0.5], [0, 0.5]]
curve = catrom.getSplineFixed(vertices, 10)

obj_planeL = bs_ext.createColorPlaneFromCurve(curve, False, 0.6, 0.6, 0.6, center=(0, 0))
obj_planeL.setShader(colorShaderProgram)
```

Superficies con splines

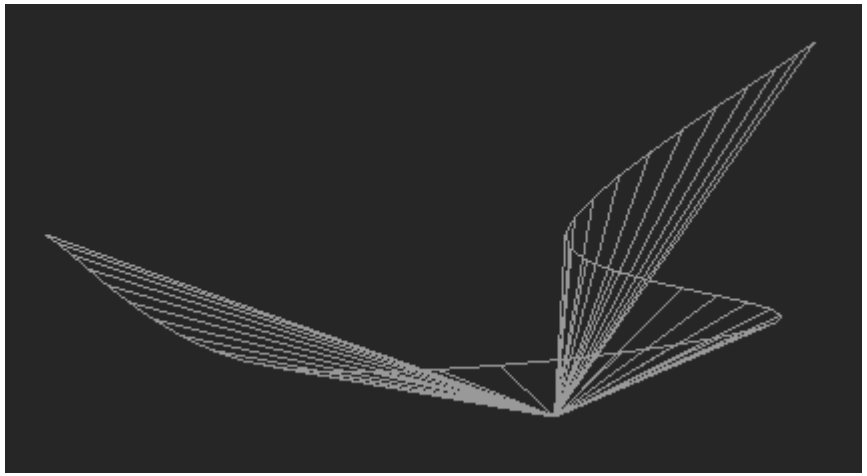
- Ejemplo con curvas de Catmull-Rom: Generando la superficie conectando cada par de puntos P_i, P_{i+1} con un centro C_i, así cada triángulo queda definido por (P_i, P_{i+1}, C_i)





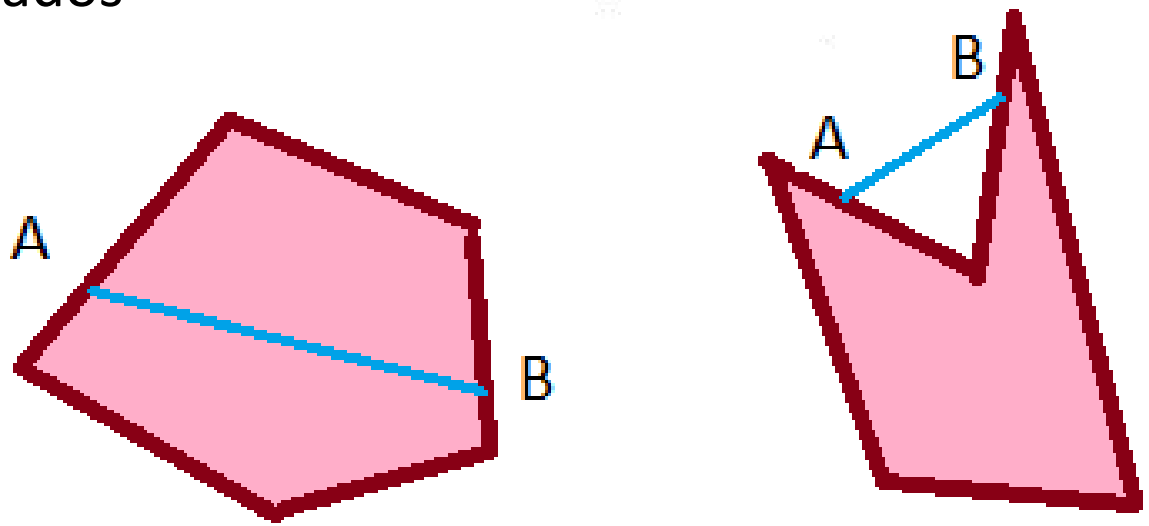
Superficies con splines

- Ejemplo con curvas de Catmull-Rom: Generando la superficie conectando cada par de puntos P_i, P_{i+1} con un centro C_i, así cada triángulo queda definido por (P_i, P_{i+1}, C_i). ¿Qué pasa en el caso de figuras no convexas?



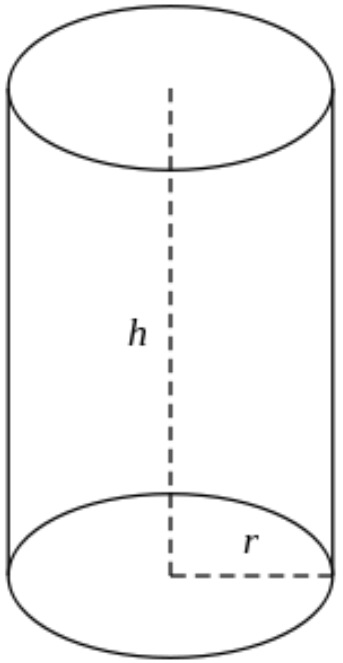
Superficies con splines

- Ejemplo con curvas de Catmull-Rom
  - Pueden ser una buena alternativa para generar superficies
  - Es complicado conocer los vértices que definen la curva
  - Para el caso en que la curva no genera un área convexa no se obtienen buenos resultados



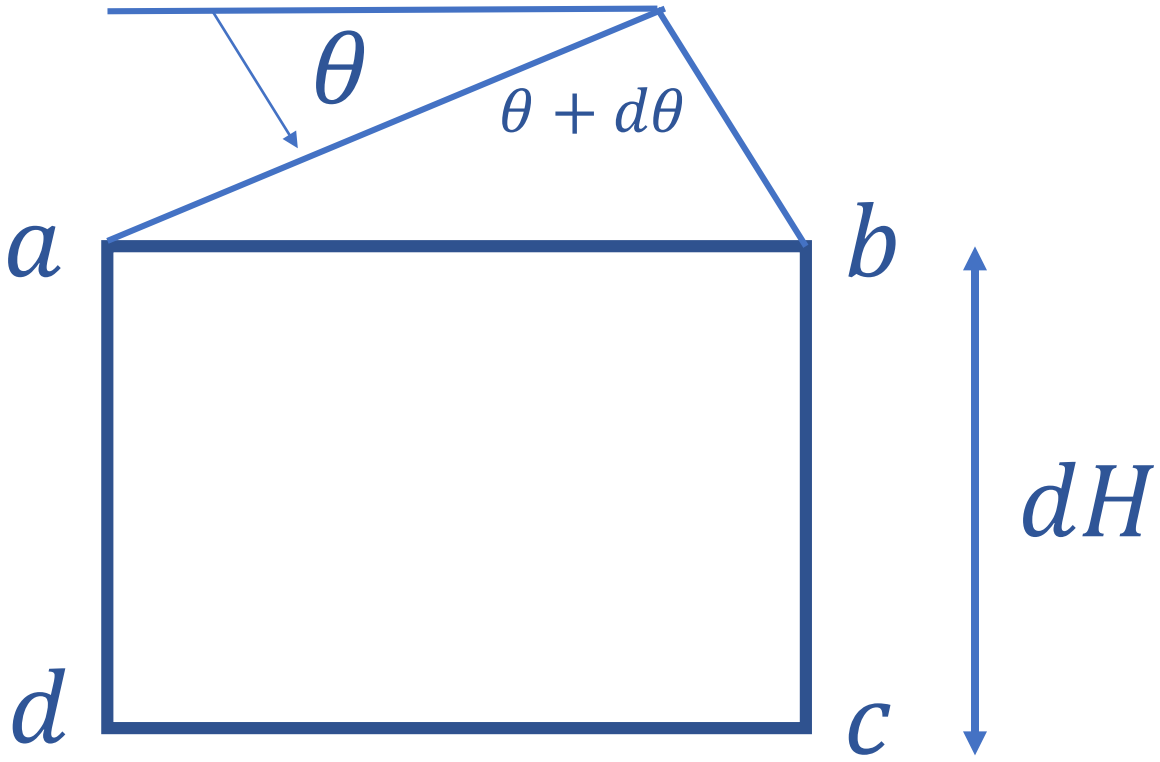
Superficies con sólidos de revolución

- Misma metodología:
  - **1)** Generar los vértices $v_i = (x_i, y_i, z_i)$
  - **2)** Conectar los vértices generados de manera ordenada usando triángulos y agrupándolos en *Shapes*
- Ejemplo práctico: Crear un cilindro



Superficies con sólidos de revolución

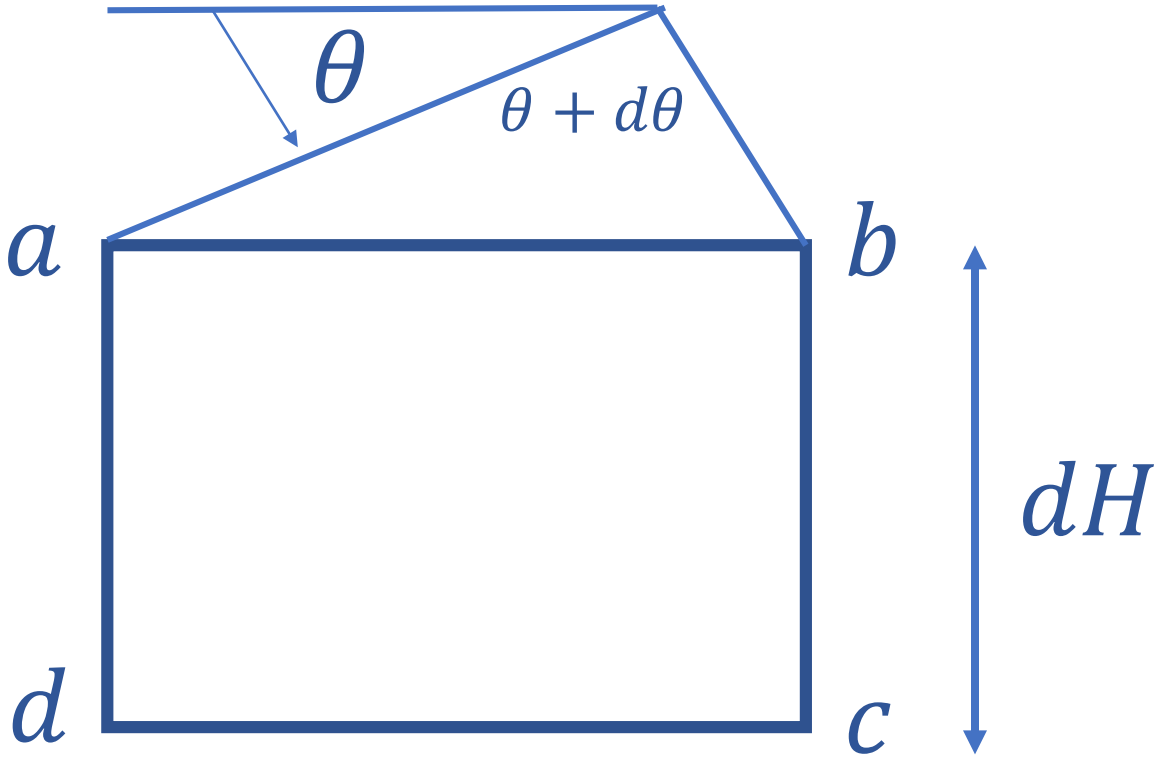
- Crear un cilindro: Primero se parametriza la circunferencia usando coordenadas polares (Radio, Theta), el objetivo luego es conectar cuatro vértices formando una cara.



¿Cómo calculamos a,b,c,d?

Superficies con sólidos de revolución

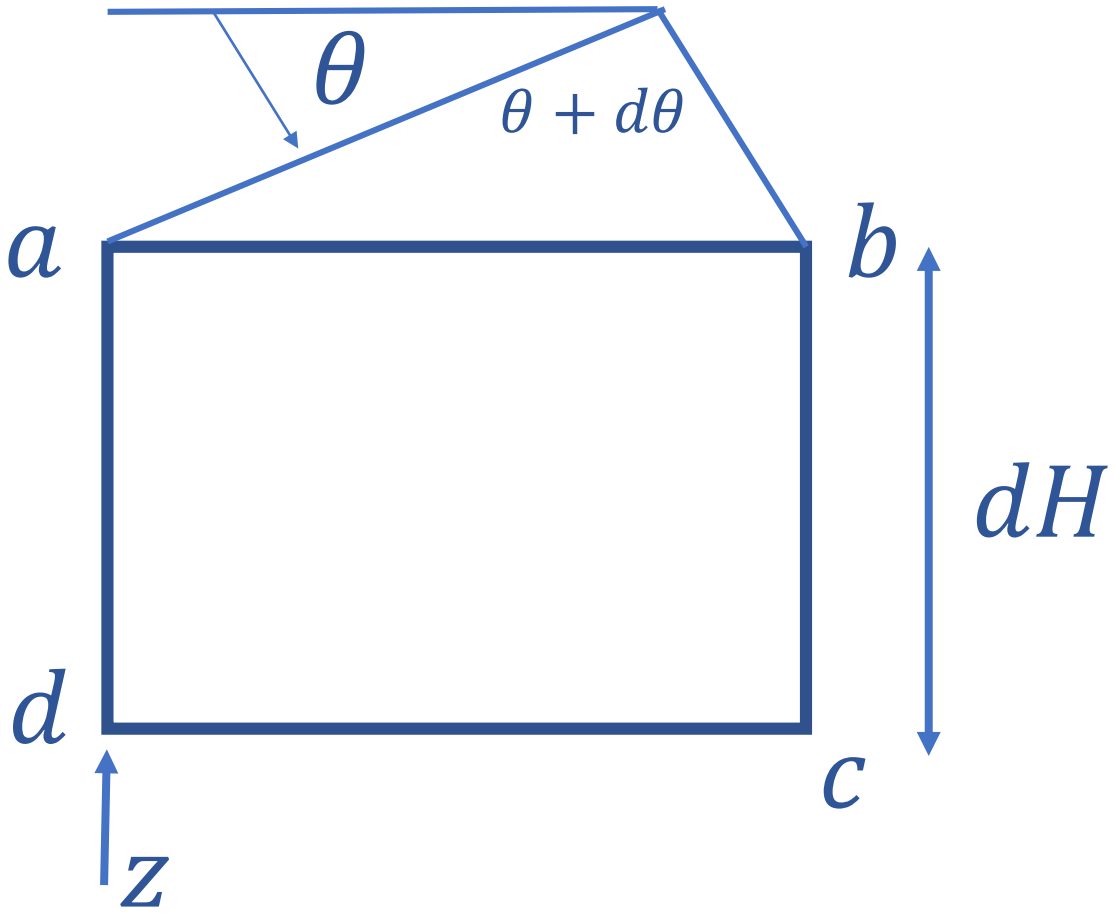
- Crear un cilindro: Primero se parametriza la circunferencia usando coordenadas polares (Radio, Theta), el objetivo luego es conectar cuatro vértices formando una cara.



¿Cómo calculamos a,b,c,d?

Superficies con sólidos de revolución

- Crear un cilindro:



a= $[r * \cos(\theta), r * \sin(\theta), dH + z]$

b= $[r * \cos(\theta + d\theta), r * \sin(\theta + \theta), dH + z]$

c= $[r * \cos(\theta + d\theta), r * \sin(\theta + \theta), z]$

d= $[r * \cos(\theta), r * \sin(\theta), z]$

Si conectamos todos estos puntos con OpenGL se forma la cara. Notar el orden en el sentido horario: a,b,c,d!!

Superficies con sólidos de revolución

- Crear un cilindro

```
# Create cilynder, the objective is create many cuads from the bottom, top and
# mantle. The cilynder is parametrized using an angle theta, a radius r and
# the height
h = 1
r = 0.25

# Latitude and longitude of the cylinder, latitude subdivides theta, longitude
# subdivides h
lat = 20
lon = 20

# Angle step
dang = 2 * np.pi / lat

# Color
color = {
    'r': 1,  # Red
    'g': 0,  # Green
    'b': 0,  # Blue
}

cylinder_shape = []  # Store shapes
```

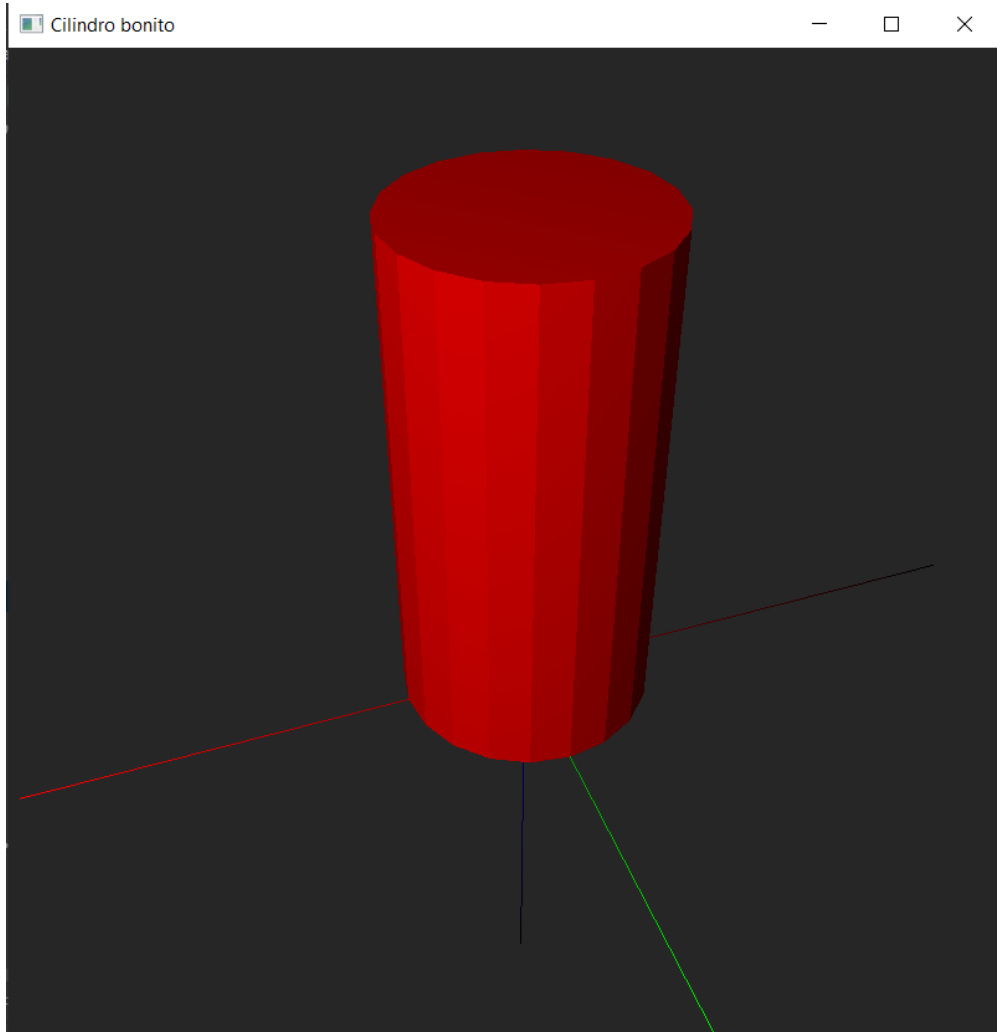
Superficies con sólidos de revolución

- Crear un cilindro

```
# Create mantle
for i in range(lon):  # Vertical component
    for j in range(lat):  # Horizontal component

        # Angle on step j
        ang = dang * j

        # Here we create a quad from 4 vertices
        #
        #    a/---- b/
        #     |      |
        #     d ---- c
        a = [r * np.cos(ang), r * np.sin(ang), h / lon * (i + 1)]
        b = [r * np.cos(ang + dang), r * np.sin(ang + dang), h / lon * (i + 1)]
        c = [r * np.cos(ang + dang), r * np.sin(ang + dang), h / lon * i]
        d = [r * np.cos(ang), r * np.sin(ang), h / lon * i]

        # Create quad
        shape = bs_ext.create4VertexColorNormal(a, b, c, d, color['r'], color['g'
        cylinder_shape.append(es.toGPUShape(shape))
```

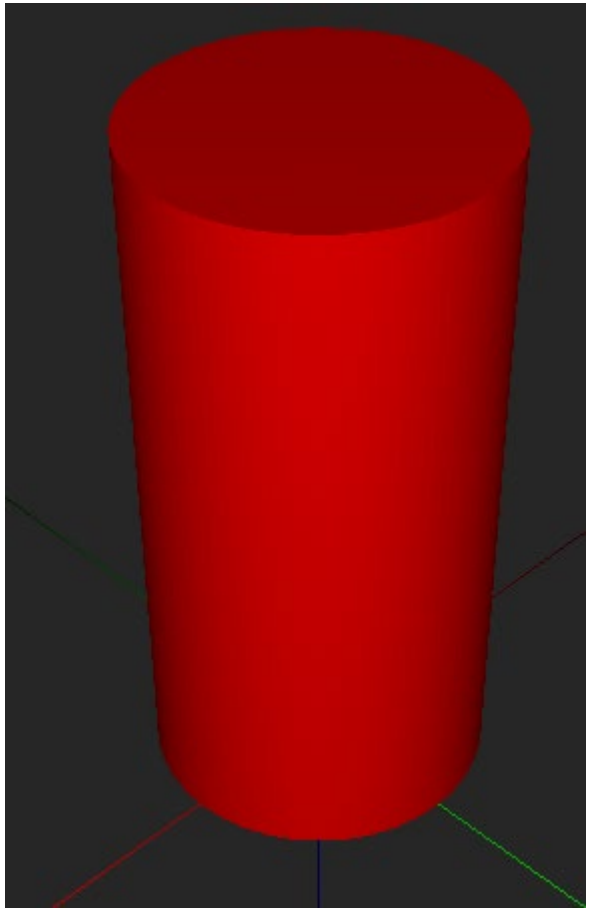
Superficies con sólidos de revolución

- Crear un cilindro



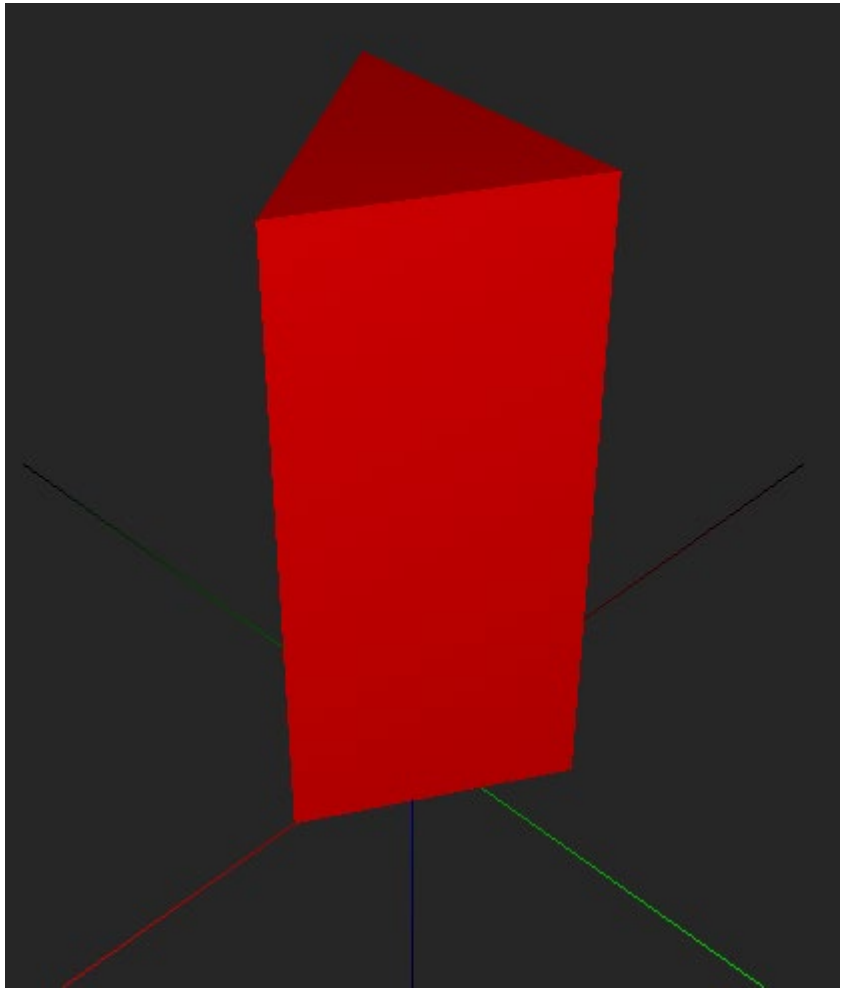
Superficies con sólidos de revolución

- Crear un cilindro
  - ¿Qué pasa si aumentamos las subdivisiones?
  - Se obtiene más detalle pero el modelo se hace más lento



Superficies con sólidos de revolución

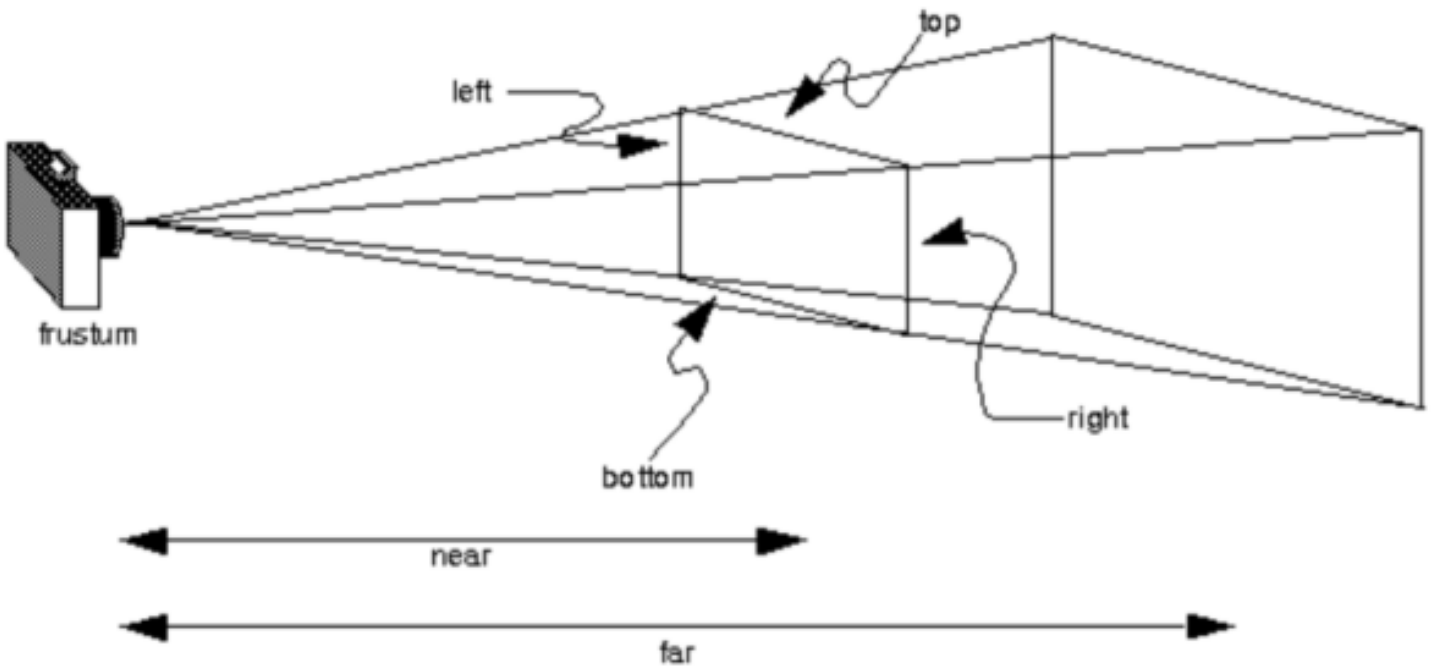
- Crear un cilindro
  - ¿Qué pasa si disminuimos las subdivisiones?
  - Se pierde mucho detalle, el optimo se debe buscar



2. Repaso concepto de cámara

Concepto de la cámara

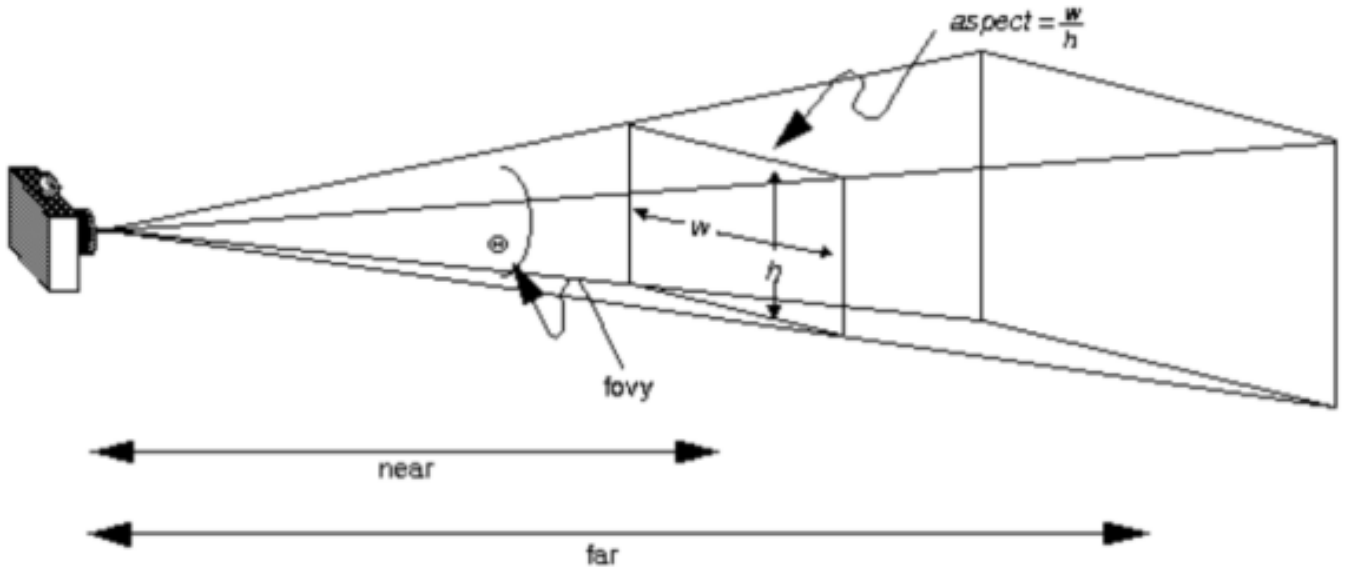
- Perspectivas: glFrustrum, volumen de visualización



glFrustum(izquierda, derecha, abajo, arriba, cerca, lejos)

Concepto de la cámara

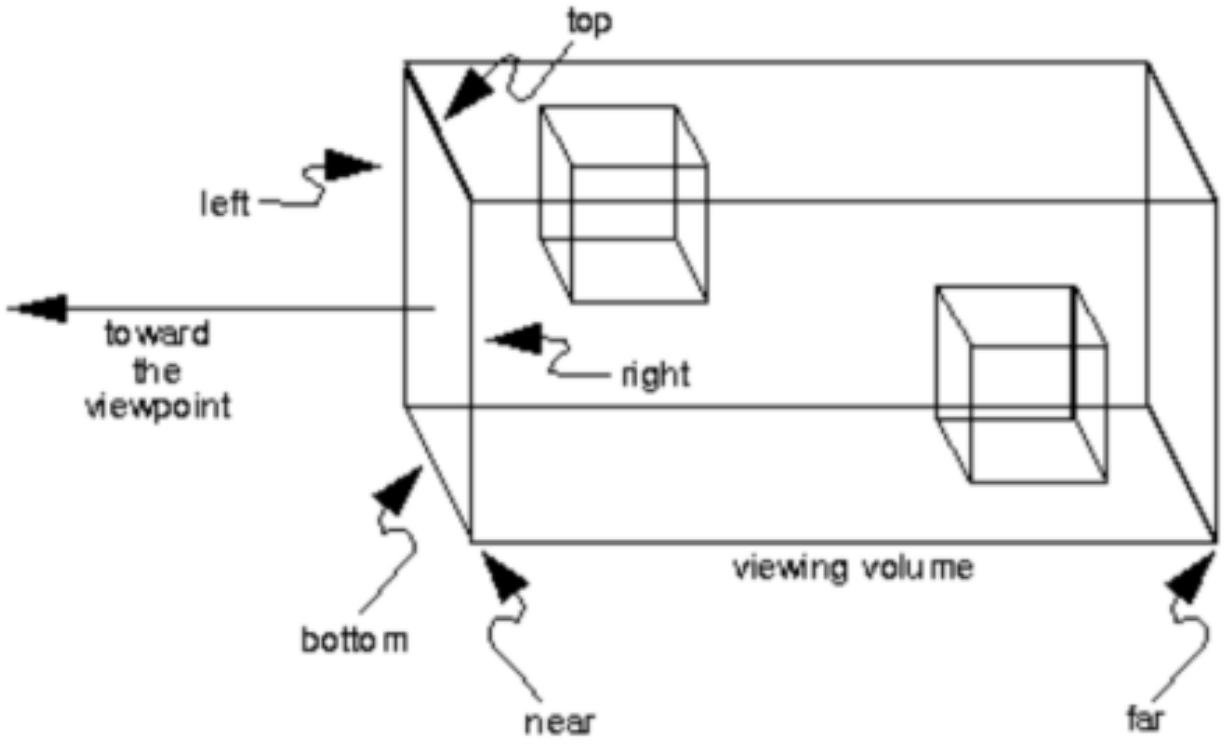
- Perspectivas: gluPerspective – Perspectiva vía ángulo de fobia



gluPerspective(fovy, aspecto, cerca, lejos)

Concepto de la cámara

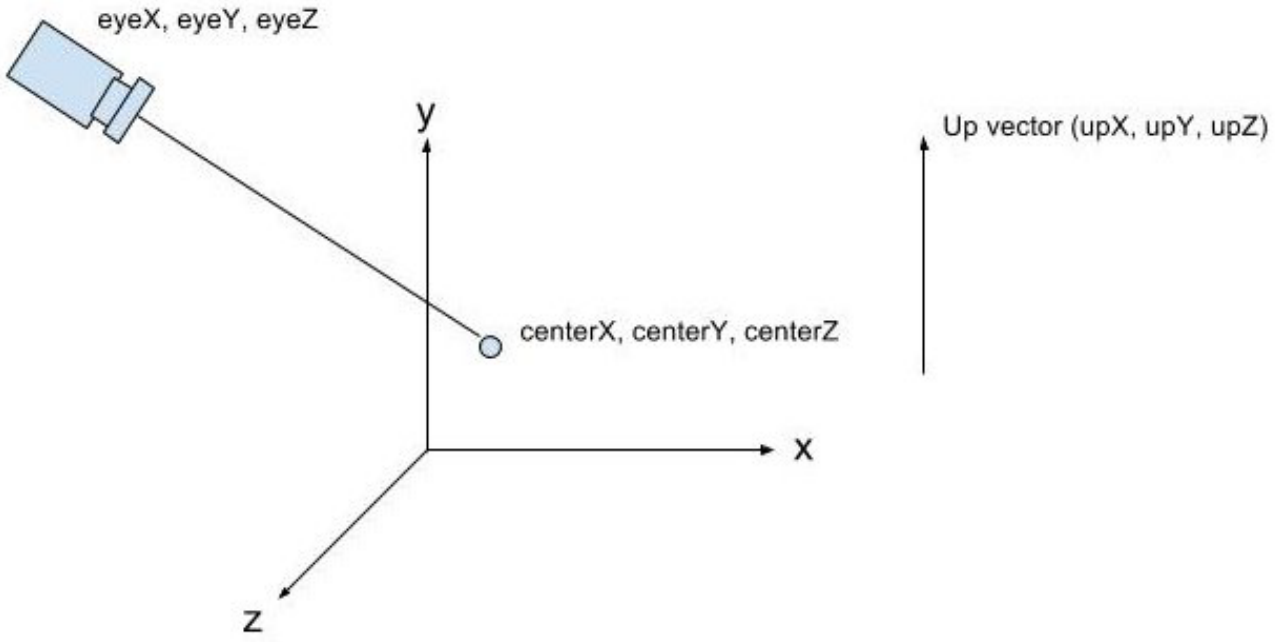
- Perspectivas: glOrtho – Proyección ortográfica



glOrtho(izquierda, derecha, abajo, arriba, cerca, lejos)

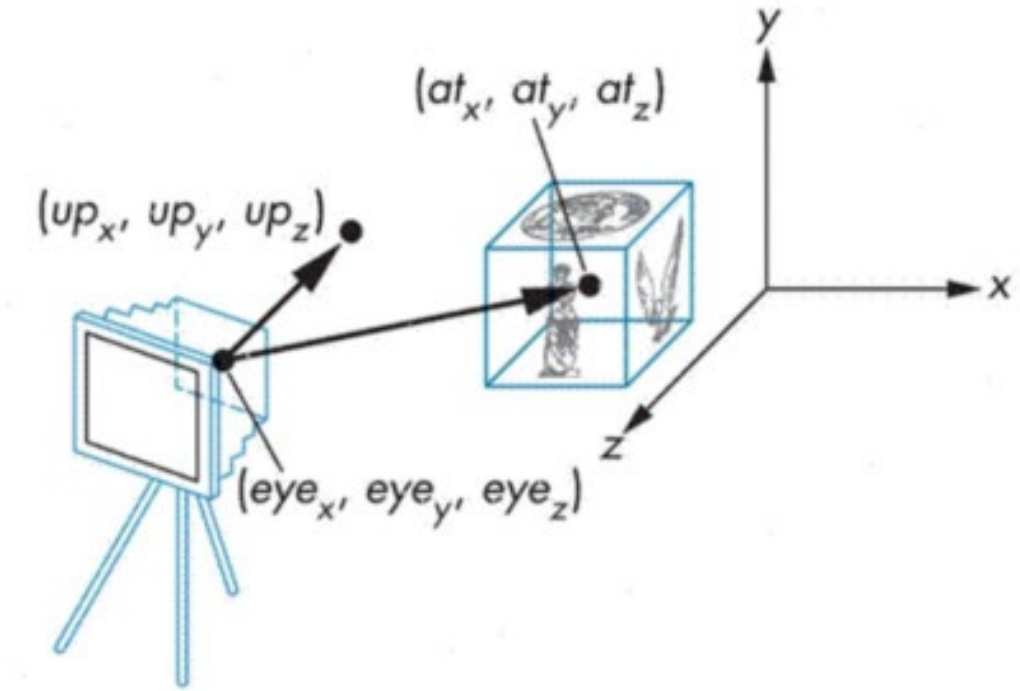
Concepto de la cámara

- Cómo implementar una cámara: Básicamente se busca generar la matriz de visualización, que considera la posición del objetivo de la cámara (Adonde apunta), la posición de la cámara (Donde estoy parado) y el “ángulo” de la cámara (Vector up)



Concepto de la cámara

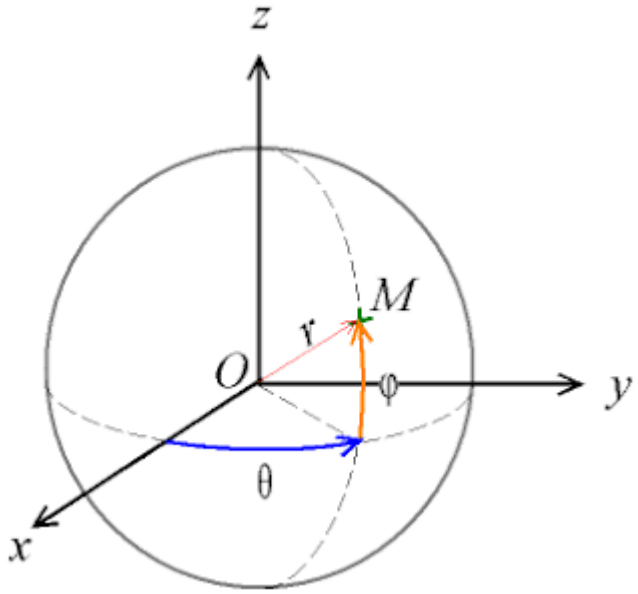
- Cómo implementar una cámara: Matriz de visualización



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

Concepto de la cámara

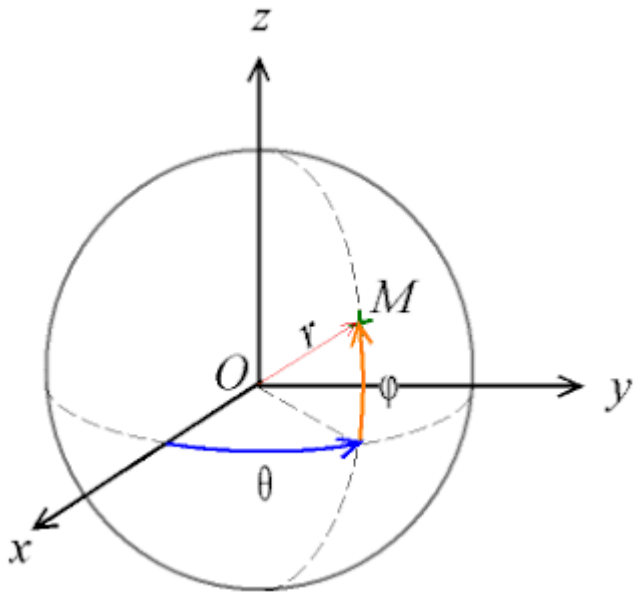
- Cómo implementar una cámara: Varias alternativas:
  - Cámara radial, coordenadas esféricas. La posición de la cámara se conoce a partir del radio, el ángulo cenital y polar.



  - Coordenadas cartesianas, la posición de la cámara es directa

Concepto de la cámara

- Cámara radial: Se implementó la clase **CameraR** que considera un radio **r**, θ y ϕ, la posición del objetivo (centro) y el vector up.
- Múltiples métodos para rotar la cámara en torno a cada eje, acercar o alejar la cámara (Modificando el radio), modificar la posición del centro, entre otros.



Concepto de la cámara

- Cámara radial

```
# Create camera
camera = cam.CameraR(r=3, center=Point3())
camera.set_r_vel(0.1)  # Define velocidad radial
camera.close()
camera.far()
camera.rotate_theta(1)
camera.rotate_phi(1)
camera.move_center_x(1)
camera.move_center_y(1)
camera.move_center_z(1)
view = camera.get_view()  # Get view matrix
```

Concepto de la cámara

- Cámara radial, obtención de la matriz de visualización

```
def get_view(self):
    """
    Get view matrix.

    :return:
    """
    return _tr2.lookAt(
        _np.array([self._r * _sin(self._theta) * _cos(self._phi),
                   self._r * _sin(self._theta) * _sin(self._phi),
                   self._r * _cos(self._theta)]),
        _np.array([self._center.get_x(), self._center.get_y(), self._center.get_z()]),
        _np.array([self._up.get_x(), self._up.get_y(), self._up.get_z()])
    )
```

Concepto de la cámara

- Cámara cartesiana: Se implementó la clase **CameraXYZ** que considera la posición del objetivo en coordenadas cartesianas, la posición del objetivo (centro) en coordenadas cartesianas y el vector up.
- Múltiples métodos para mover la cámara y el objetivo.

Concepto de la cámara

- Cámara radial

```
camera = cam.CameraXYZ(pos=Point3(5, 5, 5), center=Point3(1, 1, 1), up=Point3(0, 0, 1))
camera.move_center_z(1)
camera.move_center_y(1)
camera.move_center_z(1)
camera.set_radial_vel(0.1)
camera.far()  # Uses radial velocity
camera.close()
camera.move_x(1)
camera.move_y(1)
camera.move_z(1)
view = camera.get_view()
```

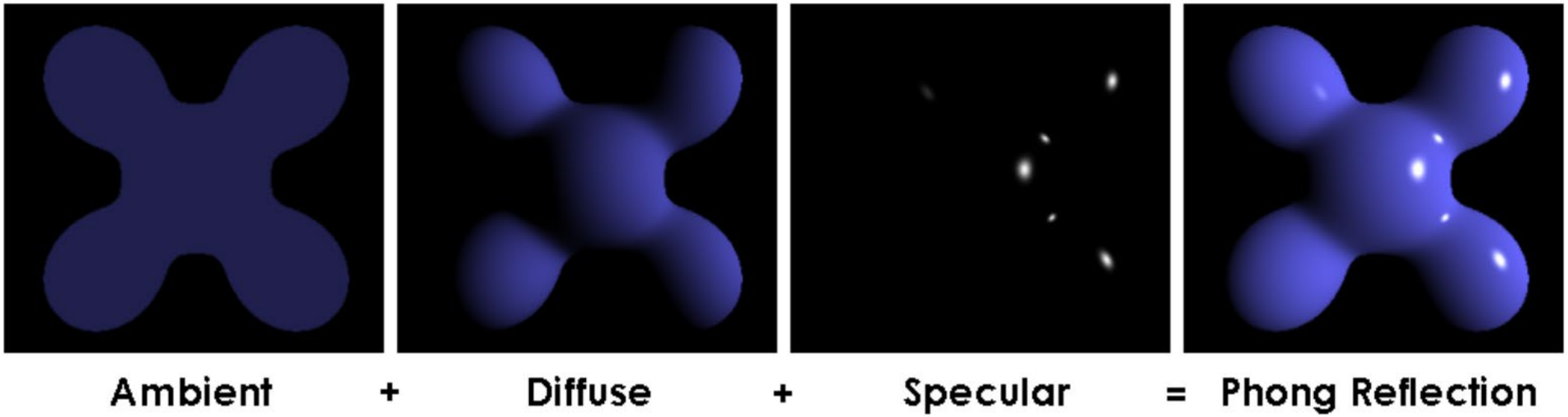
Concepto de la cámara

- Cámara radial: Obtención matriz de vista es trivial

```
def get_view(self):
    """
    Get view matrix.

    :return:
    """
    return _tr2.lookAt(
        _np.array([self._pos.get_x(), self._pos.get_y(), self._pos.get_z()]),
        _np.array([self._center.get_x(), self._center.get_y(), self._center.get_z()]),
        _np.array([self._up.get_x(), self._up.get_y(), self._up.get_z()])
    )
```

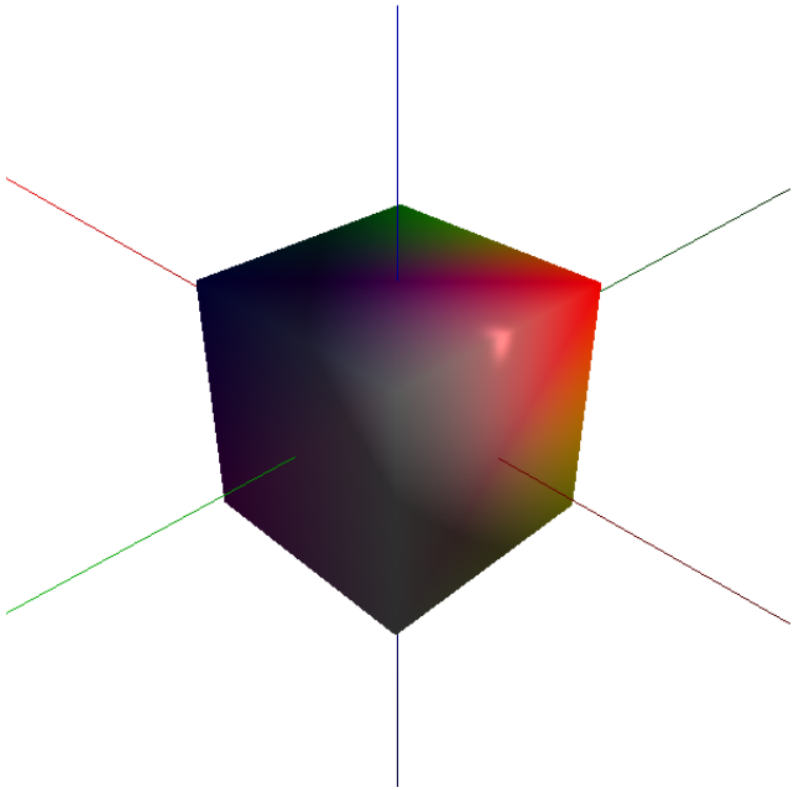
3.Iluminación

Iluminación



Modelo de iluminación de Phong

Iluminación

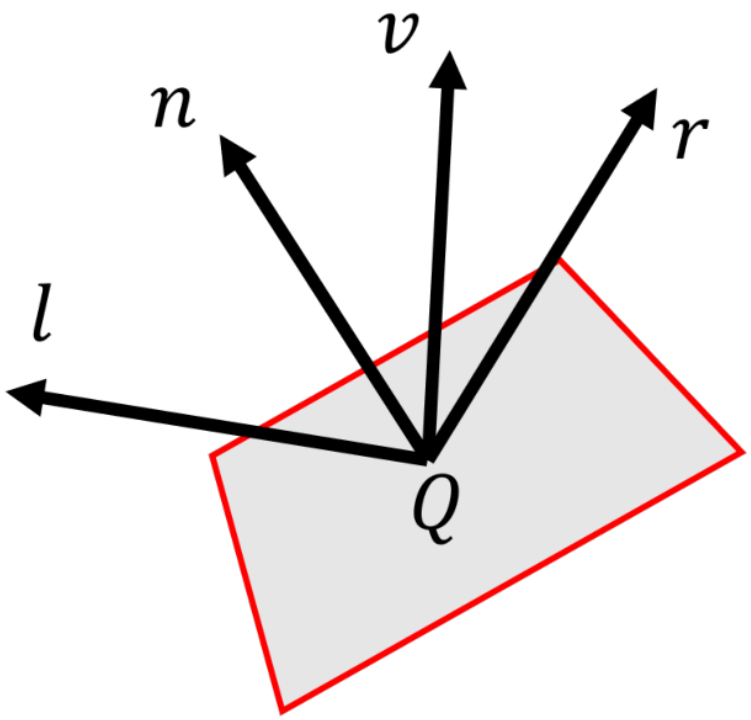


Necesitamos las siguientes especificaciones:

- Fuente de luz
- Material/color/textura
- Geometría de la superficie (i.e. normal del triángulo)

Iluminación

- ¿Qué necesitamos?
  - Material/Color/Textura
  - Geometría de la superficie, vector normal
  - Fuente de luz



Iluminación

- ¿Qué necesitamos?
  - Material/Color/Textura
- Definición de material: Configurar GL_SPECULAR, GL_AMBIENT, GL_DIFFUSE, GL_SHININESS y GL_EMISSION
  - Muy fácil
- Cada material tiene sus propios valores
- Colores y texturas ya lo sabemos

```
def material_silver(emission=None, face=_GL_FRONT_AND_BACK):
    """
    Silver material.

    :param emission: Material emission constant
    :param face: Face mode (OpenGL constant)
    :type emission: list
    :type face: int
    """
    if emission is None:
        emission = _MATERIAL_DEFAULT_EMISSION
    _glMaterialfv(face, _GL_AMBIENT, [0.19225, 0.19225, 0.19225, 1.0])
    _glMaterialfv(face, _GL_DIFFUSE, [0.50754, 0.50754, 0.50754, 1.0])
    _glMaterialfv(face, _GL_SPECULAR, [0.508273, 0.508273, 0.508273, 1.0])
    _glMaterialfv(face, _GL_SHININESS, 0.4 * 128)
    _glMaterialfv(face, _GL_EMISSION, emission)
```

Iluminación

- ¿Qué necesitamos?
  - ~~Material/Color/Textura~~
  - Geometría de la superficie, vector normal
    - Usamos la maravillosa álgebra lineal…
    - O bien podemos usar alguna librería

A(1,2,-4)
B(2,3,7)
C(4,-1,3)

Determinar el plano que los contiene

$V_{AB} = (1, 1, 11)$
$V_{AC} = (3,-3,7)$

$$\begin{vmatrix} i & j & k \\ 1 & 1 & 11 \\ 3 & -3 & 7 \end{vmatrix} = \vec{n}$$

$\vec{n}=(40, 26,-6)$

$40x + 26y -6z +D =0$

$40.1 + 26.2 -6.4+D =0$ $\quad D=-116$

$40x + 26y -6z-116=0$

Iluminación

- ¿Qué necesitamos?
  - ~~Material/Color/Textura~~
  - Geometría de la superficie, vector normal
    - create4VertexColorNormal(p1, p2, p3, p4, r, g, b)
    - create4VertexTextureNormal(image_filename, p1, p2, p3, p4, nx=1, ny=1)
    - createTriangleTextureNormal(image_filename, p1, p2, p3, nx=1, ny=1)
    - createTriangleColorNormal(p1, p2, p3, r, g, b)

    - Con esto sale fácil

Iluminación

- ¿Qué necesitamos?
  - ~~Material/Color/Textura~~
  - ~~Geometría de la superficie, vector normal~~
  - Fuente de luz
    - Configurar parámetros de OpenGL, modificando las variables del shader

```
# Setting all uniform shader variables
glUniform3f(glGetUniformLocation(lightingPipeline.shaderProgram, "lightColor"), 1.0, 1.0, 1.0)
glUniform3f(glGetUniformLocation(lightingPipeline.shaderProgram, "lightPos"), -5, -5, 5)
glUniform3f(glGetUniformLocation(lightingPipeline.shaderProgram, "viewPos"), viewPos[0], viewPos[1], viewPos[2])
glUniform1ui(glGetUniformLocation(lightingPipeline.shaderProgram, "shininess"), 100)
glUniform1f(glGetUniformLocation(lightingPipeline.shaderProgram, "constantAttenuation"), 0.001)
glUniform1f(glGetUniformLocation(lightingPipeline.shaderProgram, "linearAttenuation"), 0.1)
glUniform1f(glGetUniformLocation(lightingPipeline.shaderProgram, "quadraticAttenuation"), 0.01)
```

Iluminación

- ¿Qué necesitamos?
  - ~~Material/Color/Textura~~
  - ~~Geometría de la superficie, vector normal~~
  - Fuente de luz
    - Nuevo objeto Light que necesita parámetros de shader, posición, luz contantes.

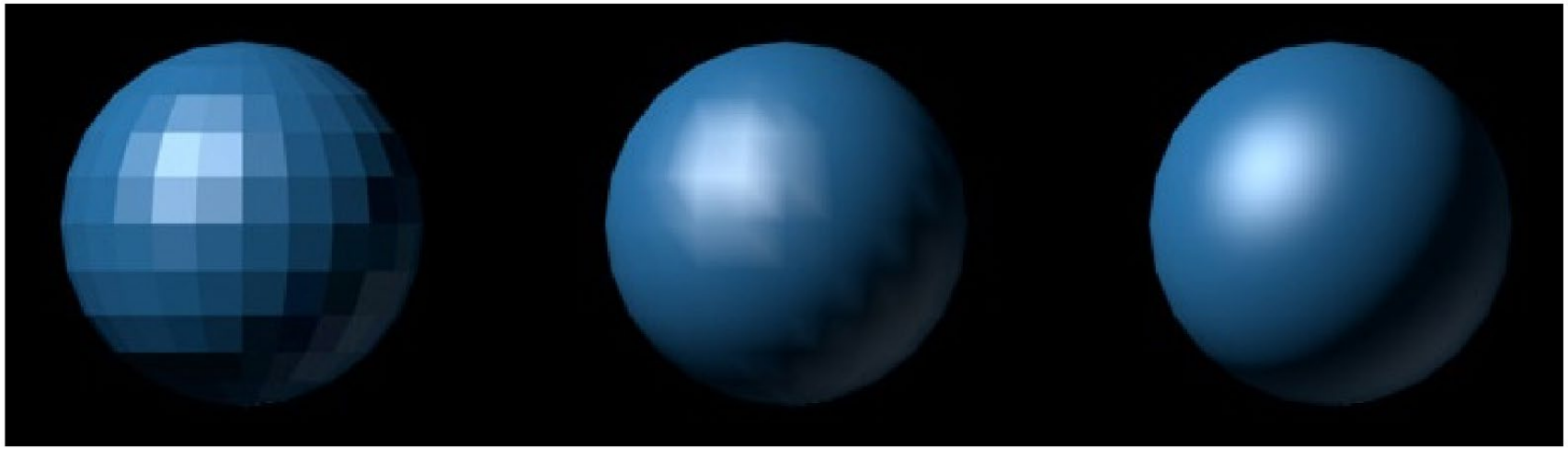
```
# Create light
obj_light = light.Light(shader=phongPipeline, position=[5, 5, 5], color=[1, 1, 1])
```

  - Su uso es muy fácil:

```
# Place light
obj_light.place()

# Draw objects
obj_axis.draw(view, projection, mode=GL_LINES)
obj_cylinder.draw(view, projection)
```

Iluminación

- ¿Qué necesitamos?
  - ~~Material/Color/Textura~~
  - ~~Geometría de la superficie, vector normal~~
  - ~~Fuente de luz~~
- Lo más importante: Modelo de iluminación (shader)



Sobreado Flat, Gouraud y Phong

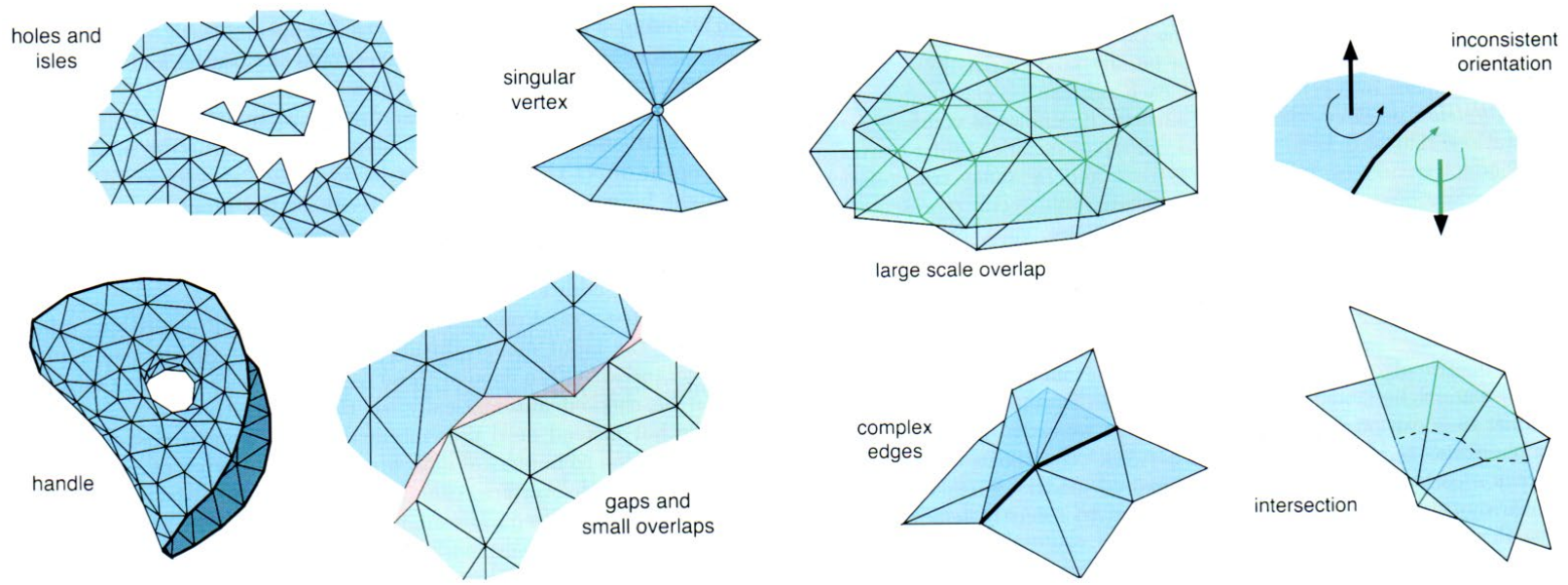
4.Mallas

Mallas

- Conceptos importantes:
  - Estructura de datos
  - Triangulaciones
  - Modelación de terrenos
  - Curvas de nivel

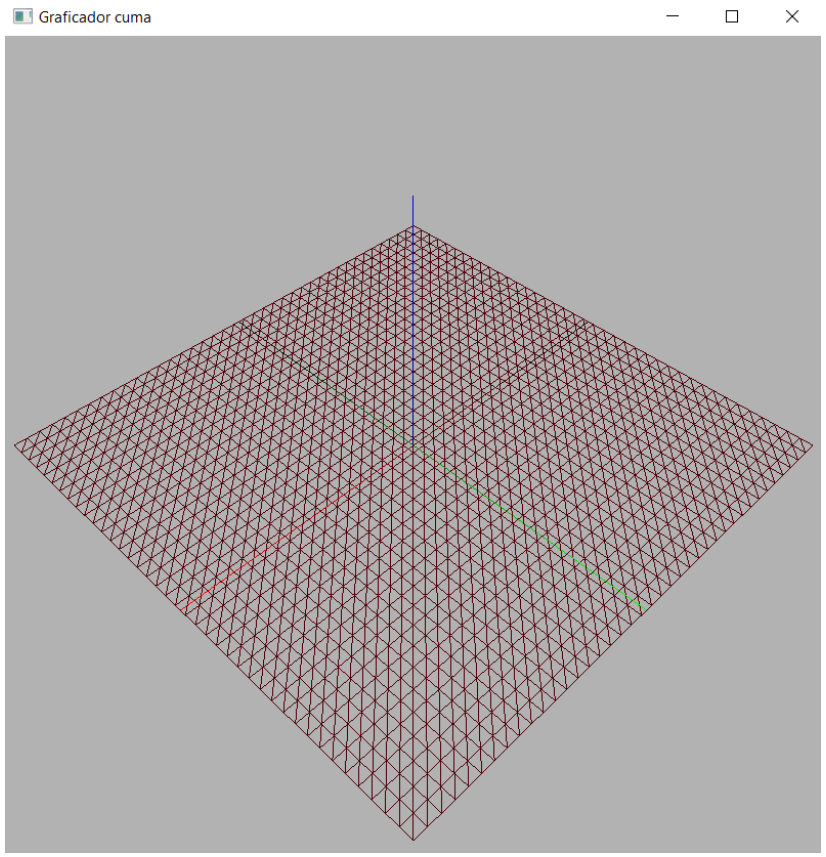
Mallas

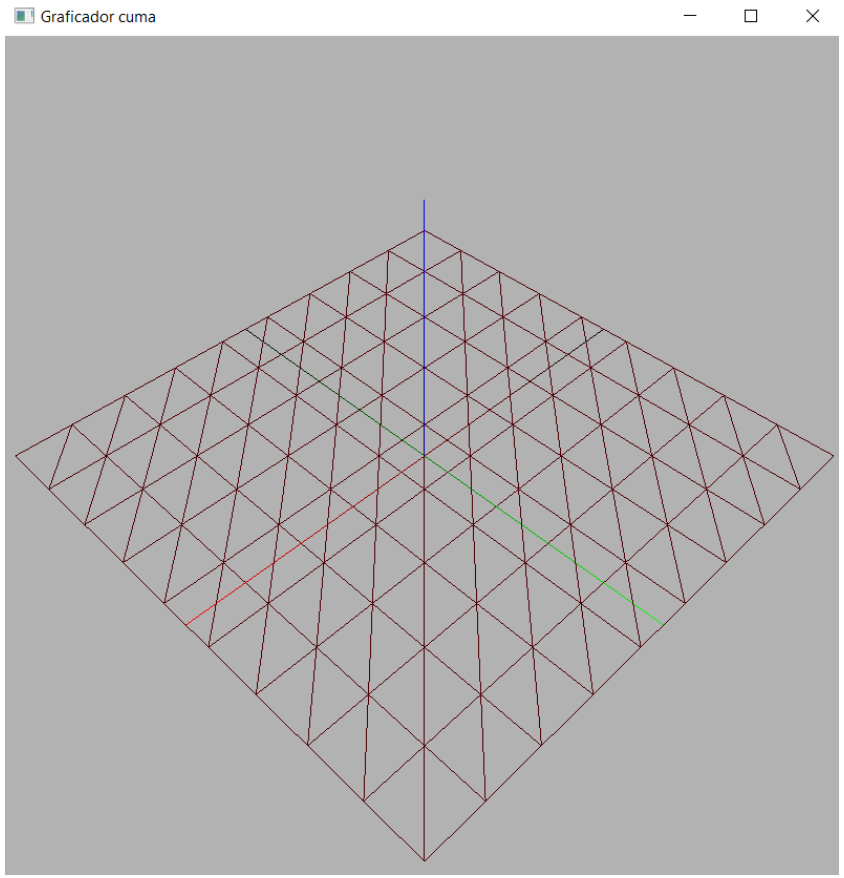
- Todo sigue el mismo concepto, encontrar una forma de generar las caras de los modelos asignando POSICIONES (x,y,z) y TOPOLOGÍA (cómo se conectan).



Mallas

- Ejemplo sencillo: Graficador 3D (Iba a ser una tarea)…
  - Primer paso, definir una grilla de tamaño variable

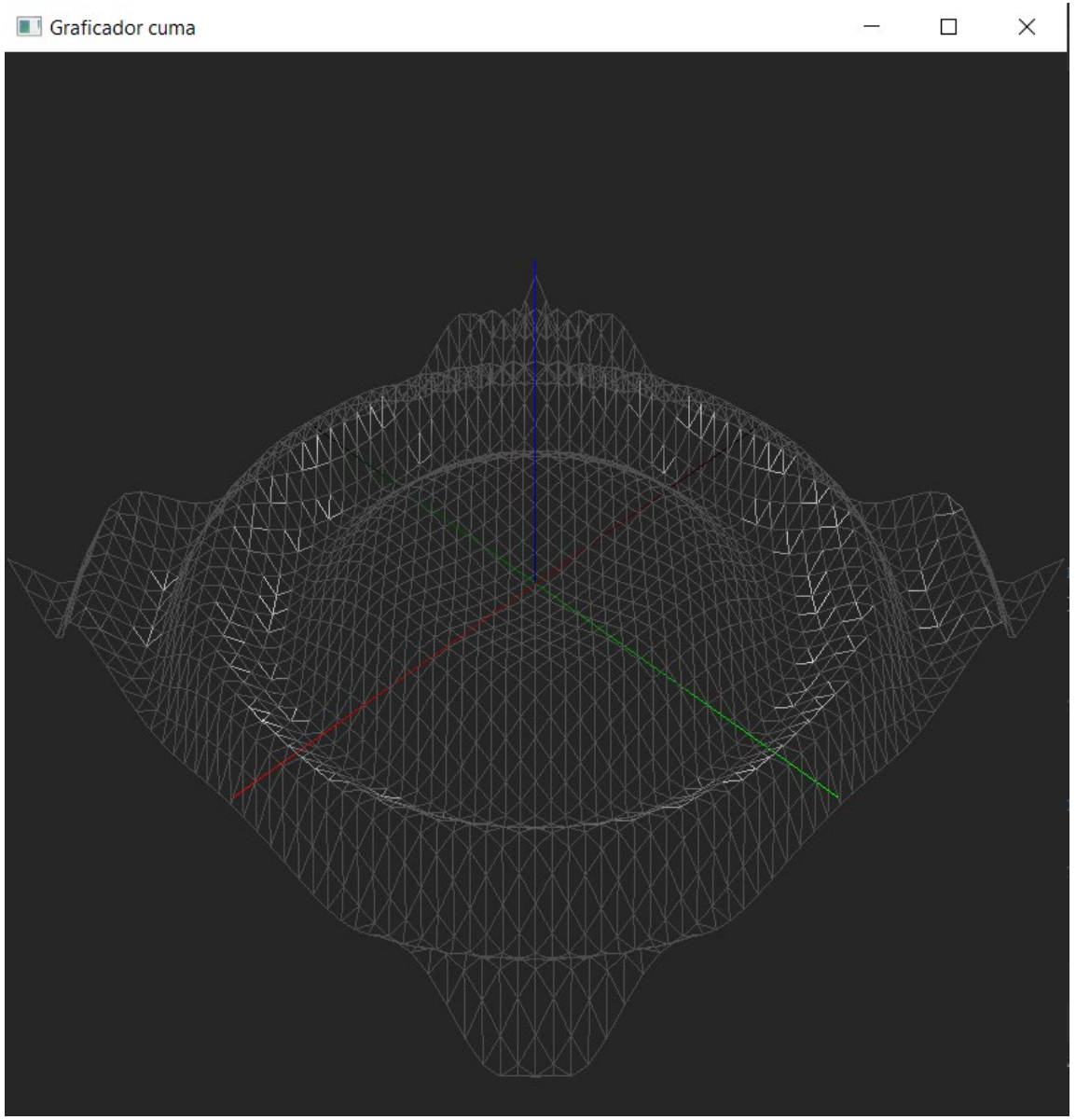




Mallas

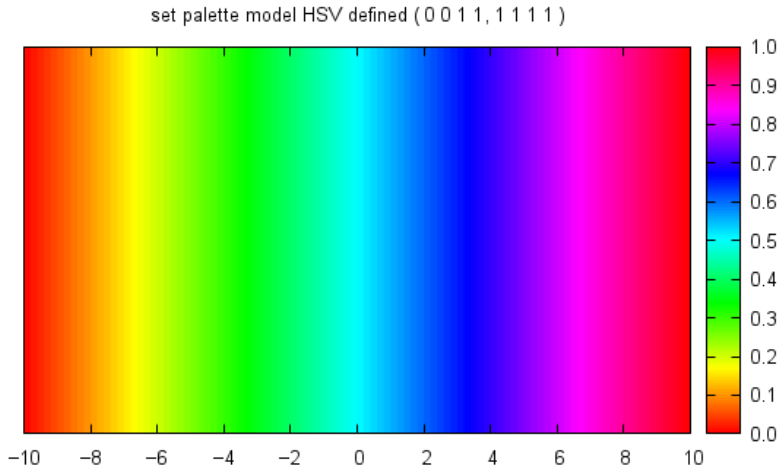
- Ejemplo sencillo: Graficador 3D
  - Segundo paso: Crear una función z=f(x,y) que asigne alturas a los vértices

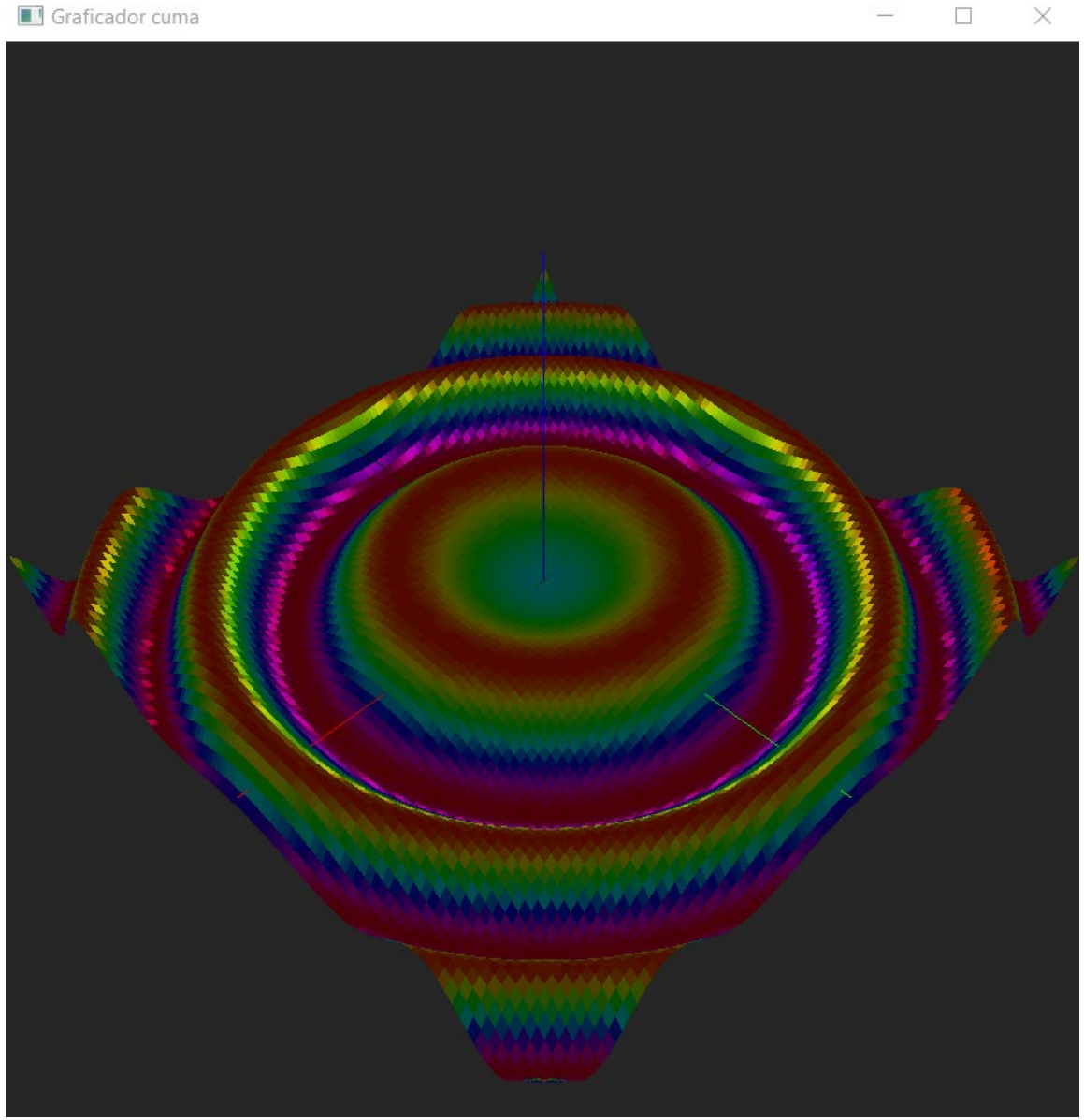
$$z = f(x, y) = \frac{\sin(10 * (x^2 + y^2))}{10}$$



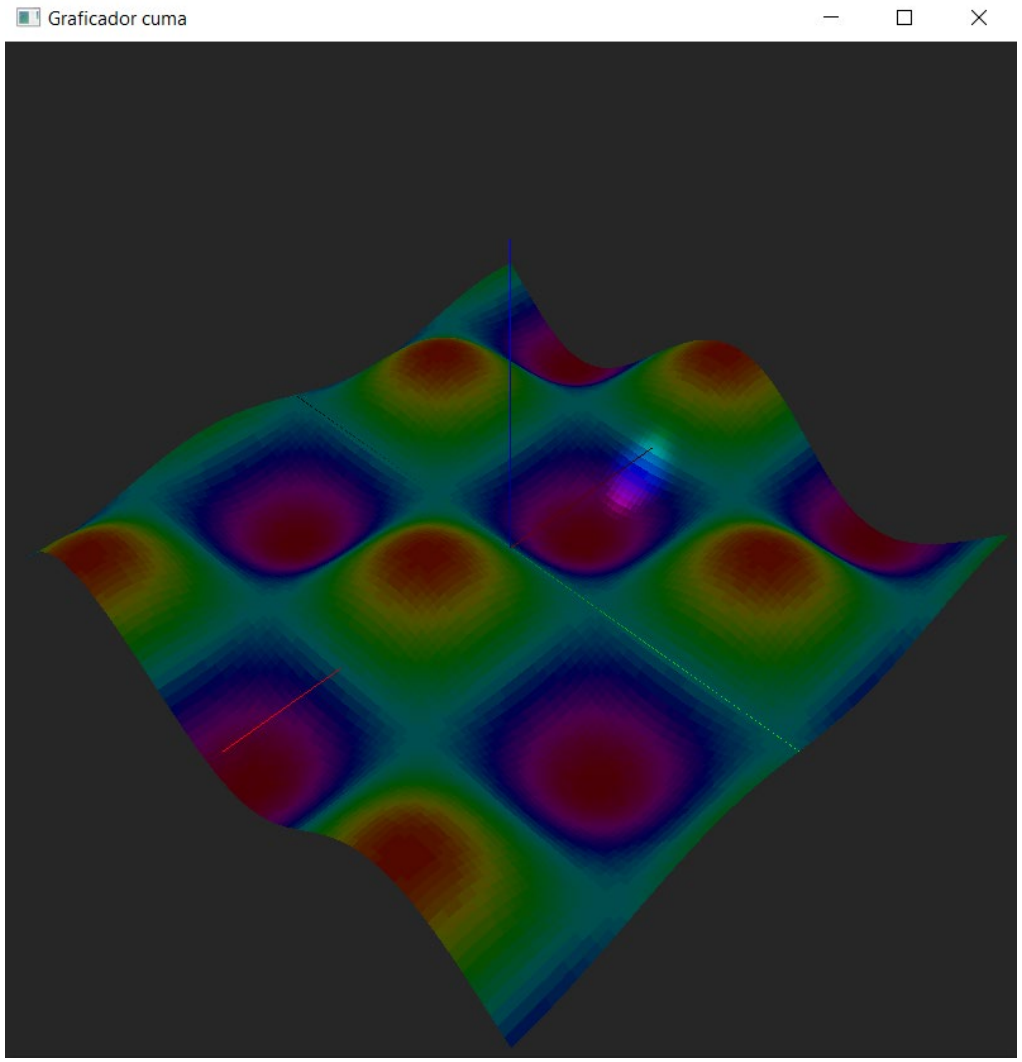
Mallas

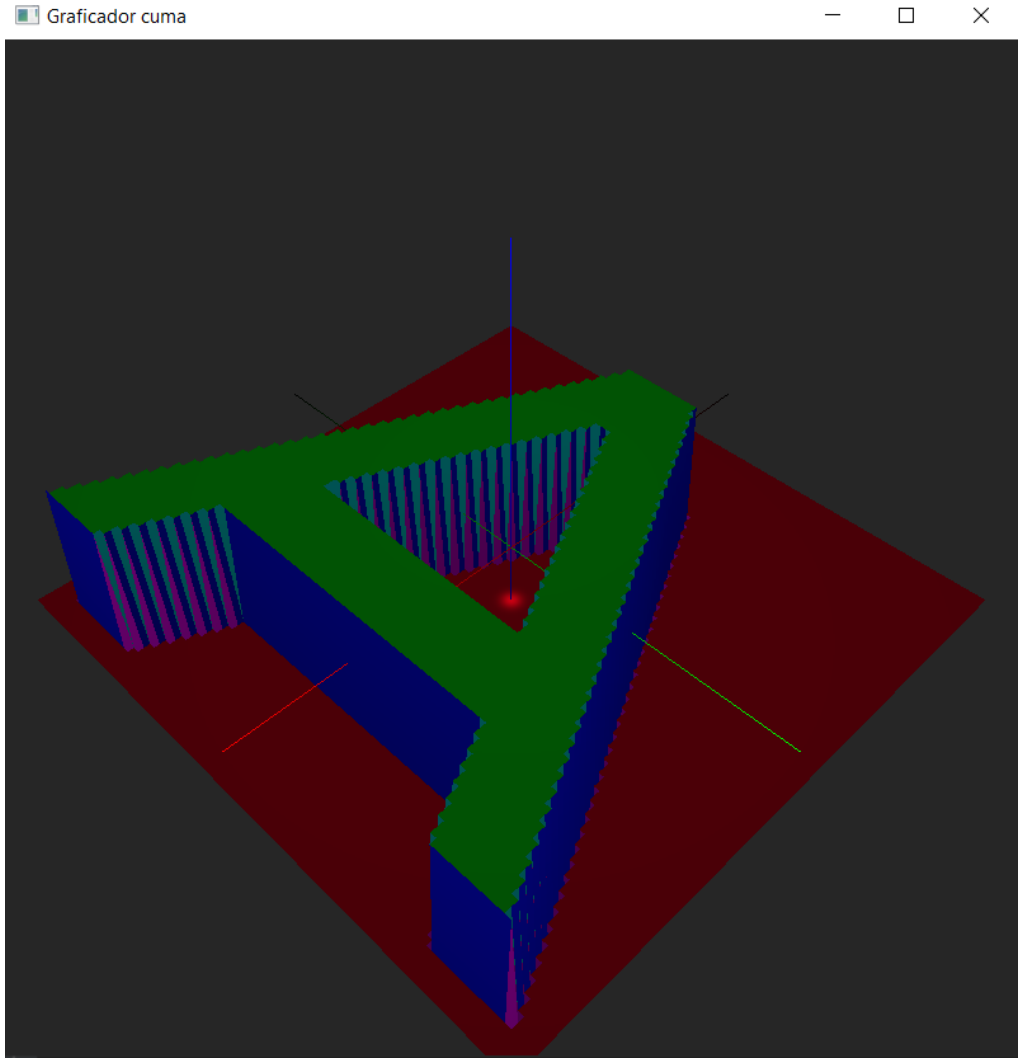
- Ejemplo sencillo: Graficador 3D
  - Tercer paso: Interpolar la altura asignando distintos colores. Se recomienda la paleta HSV





Mallas





Ejercicio propuesto

- Desarrollar la P1 del auxiliar 6 (Subido a material docente)

Muchas gracias por su atención

¿Preguntas?