CC3501 - Aux 5: OpenGL en 3D: Luz, Cámara, Acción!

Profesores: Daniel Calderón y Nancy Hitschfeld

Auxiliares: Diego Donoso y Alonso Utreras

Ayudantes: Ilana Mergudich, Nelson Marambio y Francisco Muñoz

Fecha: 12/09/2019

Contenidos de la sesión:

- Aprender a manipular modelos en OpenGL en 3D
- Aprender a texturizar modelos en 2D y 3D
- Ejercitar distintas configuraciones de cámara
- Configurar un programa de shader para iluminación

Nota: Copiar todos los archivos del auxiliar en una misma carpeta

Repaso!!

Uhmmm, primero, veamos proyecciones!

Wait...qué? Veamos el material que nos armó Alonso :D

Las proyecciones se relacionan directamente con lo que despliega la pantalla. Ya no estamos moviendo las figuras, sino que estamos simulando nuestro propio movimiento y nuestra visión. Sin embargo, el computador no comprende la idea de "movernos nosotros" o "qué es lo que vemos". Tenemos que entregarle un formato matemático que nos haga percibir que estamos cambiando la cámara. Es por eso que se implementa la matriz de vista y la matriz de proyección. Por otra parte, es importante considerar que en una pantalla no se puede ver en 3 dimensiones. La coordenada Z que representa profundidad en el mundo de OpenGL es simulada según el tipo de proyección a utilizar

La matriz de proyección:

Existen distintos tipos de proyecciones, cada una atiende supuestos distintos. Entre las más relevantes, se encuentran la visión en perspectiva y la visión ortográfica.

La proyección ortográfica:

- No es muy realista, da la impresión de que todo está más cerca.

- Las líneas paralelas de las figuras lo seguirán siendo luego de ser proyectadas (en general esto no ocurriría).
- Es más usada para programas de modelamiento de edificios, como en arquitectura. Hablemos de la matemática: requiere definir un left, right, bottom, top, near y far.

$$M_{Ortográfica} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ahora, ¿cómo vemos la posición final de un objeto con esta proyección? Hay que multiplicar por el vector posición.

$$P_{final} = M_{Ortográfica} \cdot Posición(x, y, z, 1)$$

Pero seamos realistas. Nunca ocuparemos una matriz ortográfica únicamente. Normalmente incluiremos transformaciones para los objetos. Luego, una posición final quedará dada por:

$$P_{final} = M_{Ortográfica} \cdot M_{transformaciones} \cdot Posición(x, y, z, 1)$$

Claramente hay más cambios que podemos agregar, como la matriz de vista. Pero esto se verá después y el cambio se verá reflejado en el vertex_shader (ver easy_shaders.py)

La proyección en perspectiva:

- Se asemeja más a la manera en que vemos, pues las cosas más lejanas se ven más pequeñas y viceversa.
- Las líneas originalmente paralelas sí se tocan. Matemáticamente, esto ve reflejado en la matriz:

$$M_{Frustrum} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Al respecto: Notar que la tercera columna tiene términos distintos de 0 para la fila que multiplica a x f_1 y para la fila que multiplica a y f_2. Además, no tiene términos para la coordenada homogénea, sólo en el caso de Z. Recordemos que los puntos con x mayor que 1 e y mayor 1 quedan fuera del plano de vista. Acá tenemos términos positivos que multiplicarán a Z: en el caso de x:

$x' = \frac{2n}{r-l} \cdot x + \frac{r+l}{r-l} \cdot z$

De esta manera podemos pensar que si z es muy grande (el objeto está muy lejos) es más probable que no se vea (x > 1). Lo mismo si está muy cerca, pues z tendrá un valor negativo y x puede resultar siendo menor que 1 (x < 1).

Finalmente, la posición queda dada por:

$$P_{final} = M_{Frustrum} \cdot M_{transformaciones} \cdot Posición(x, y, z, 1)$$

Vale la pena considerar que se puede transformar entre Frustrum y Perpsectiva, pues ambas siguen la misma idea, sólo que reciben parámetros distintos.

La matriz de vista:

La cámara busca crear un espacio ortogonal a partir de tres vectores:
* eye = Es la posición original de la cámara.
* at = Es hacia donde mira la cámara.
* up = Apunta hacia arriba de la cámara. Indica la orientación de esta.

Estos vectores deben ser linealmente independientes para lograr hacer la matriz de Mundo a Vista.

Esta matriz perfectamente se puede componer con la matriz de proyecciones e incluso con la de transformaciones (generalmente llamada model matrix en la comunidad OpenGL).
La matriz de vista es la siguiente:

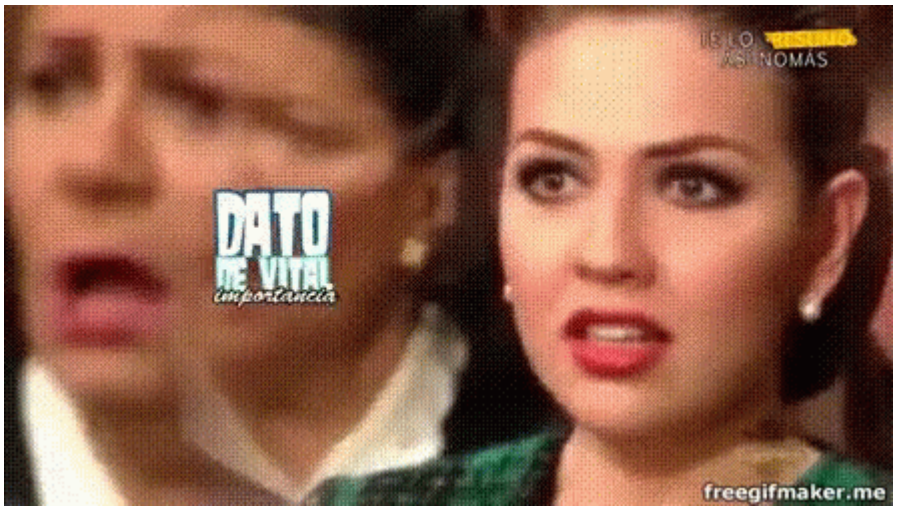
$$M_{view} = \begin{bmatrix} s_x & s_y & s_z & -e \cdot s \\ u'_x & u'_y & u'_z & -e \cdot u' \\ -f_x & -f_y & -f_z & e \cdot f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde:
f: forward vector = a - e (at - eye). Normalizado es $\frac{a-e}{||a-e||}$
s: sideways vector (que surge de f x u). Donde u = up.
u': vector up redefinido a partir de s x f. Esto se realiza para asegurar ortonormalidad.

Revisemos Código de vital importancia para el desarrollo de la trama

In [14]:
```
from IPython.display import HTML
HTML('<center><img src="https://media1.tenor.com/images/c507b9da5828f5854e0751b7b
```

Out[14]:



Recuerdan el módulo transformations.py? Acá veremos el resto de las transformaciones interesantes que no se relacionan con el modelo.

Un error típico que suele surgir en la función lookAt surge al no entregar vectores linealmente independientes.

In [15]:

```
"""
transformations.py
"""

def frustum(left, right, bottom, top, near, far):
    r_l = right - left
    t_b = top - bottom
    f_n = far - near
    return np.array([
        [ 2 * near / r_l,
        0,
        (right + left) / r_l,
        0],
        [ 0,
        2 * near / t_b,
        (top + bottom) / t_b,
        0],
        [ 0,
        0,
        -(far + near) / f_n,
        -2 * near * far / f_n],
        [ 0,
        0,
        -1,
        0]], dtype = np.float32)


def perspective(fovy, aspect, near, far):
    halfHeight = np.tan(np.pi * fovy / 360) * near
    halfWidth = halfHeight * aspect
    return frustum(-halfWidth, halfWidth, -halfHeight, halfHeight, near, far)


def ortho(left, right, bottom, top, near, far):
    r_l = right - left
    t_b = top - bottom
    f_n = far - near
    return np.array([
        [ 2 / r_l,
        0,
        0,
        -(right + left) / r_l],
        [ 0,
        2 / t_b,
        0,
        -(top + bottom) / t_b],
        [ 0,
        0,
        -2 / f_n,
        -(far + near) / f_n],
        [ 0,
        0,
        0,
        1]], dtype = np.float32)
```

```
def lookAt(eye, at, up):

    forward = (at - eye)
    forward = forward / np.linalg.norm(forward)

    side = np.cross(forward, up)
    side = side / np.linalg.norm(side)

    newUp = np.cross(side, forward)
    newUp = newUp / np.linalg.norm(newUp)

    return np.array([
            [side[0],       side[1],    side[2], -np.dot(side, eye)],
            [newUp[0],     newUp[1],   newUp[2], -np.dot(newUp, eye)],
            [-forward[0], -forward[1], -forward[2], np.dot(forward, eye)],
            [0,0,0,1]
        ], dtype = np.float32)
```

Observe que las funciones frustrum(), perspective() y ortho() entregan exactamente las matrices de las que estábamos hablando. Estas matrices se entregan explícitamente al shader (si trabajamos en OpenGL Core Profile).

Revisemos sobre los shaders

Aquí es de especial interés el pipeline Model View Projection que contiene las matrices de modelo, vista y proyección.
gl_Position = projection * view * model * vec4(position, 1.0f);
Cuático.
El vertex shader tomará las posiciones de cada vértice y les aplicará las transformaciones, aplicará la matriz de vista y finalmente la de proyección. Es importante notar que estas matrices deben aplicarse a cada vértice, de lo contrario se formaría una escena sin lógica.

Hay que tener especial cuidado con seleccionar el shader correcto, e indicar que lo utilizaremos con glUseProgram(...) ¿Y cómo le entregamos estos parámetros al shader?

projection = una matriz de projection
view = una matriz de view
model = una matriz de model

glUniformMatrix4fv(glGetUniformLocation(pipeline.shaderProgram, "view"), 1, GL_TRUE, view)
glUniformMatrix4fv(glGetUniformLocation(pipeline.shaderProgram, "projection"), 1, GL_TRUE, projection)
glUniformMatrix4fv(glGetUniformLocation(pipeline.shaderProgram, "model"), 1, GL_TRUE, model)

Supongamos que queremos poder movernos con la cámara. Bastaría manipular el controller y declarar en el modelo algo que se conecte con esto. Lo cual se agregó en el loop principal:

if glfw.get_key(window, glfw.KEY_LEFT) == glfw.PRESS:
camera_theta -= 2 * dt
if glfw.get_key(window, glfw.KEY_RIGHT) == glfw.PRESS:
camera_theta += 2 * dt

Finalmente, vemos que esto modifica directamente la matriz de vista.

In [ ]:
```
# Setting up the view transform

        camX = 10 * np.sin(camera_theta)
        camY = 10 * np.cos(camera_theta)

        viewPos = np.array([camX, camY, 10])

        view = tr.lookAt(
            viewPos,
            np.array([0, 0, 0]),
            np.array([0, 0, 1])
        )
```

UFF, y ahora qué?

Pos, Iluminación

Dentro de la computación gráfica se desea tener un efecto realista en la escena, para ello se construyen distintos modelos que nos sirven para acercar un poco la realidad a la pantalla, por ejemplo se tienen los modelos de iluminación de Phong.
Para ello tenemos que definir los siguientes conceptos:

- Luz ambiental: Fuente o dirección no identificable
- Fuente puntual: Generada por un punto
- Luz distante: Solo apreciamos una dirección
- Spot Light: Desde una fuente hacia una dirección específica (Ángulo de corte define un cono de luz)

Notas sobre el modelo de Phong:

- Es un modelo de iluminación local, no considera proyección de sombras ni reflexiones
- Usado por su simpleza y dado como una buena aproximación
  Dato rosa:
  También se definen modelos de iluminación globales que usan algoritmos de ray tracing, estos son más cercanos a la realidad dado que existe una interacción del modelo con el ambiente

Rtx es la tecnología que sacó Nvidia que "implementa" ray tracing en tiempo real

Reflexión ambiental
Modela los objetos con colores simples

$$\mathcal{I}_a = \mathcal{K}_a \mathcal{L}_a$$

- en donde $\mathcal{K}_a$ es el coeficiente de reflexión ambiental (cuánta luz refleja el objeto) y es mayor o igual a 0
- $\mathcal{L}_a$ corresponde a la luz ambiental (no es signitifactivo)

Reflexión especular
La dirección de la luz está dada por un vector

$$\mathcal{I}_d = \frac{\mathcal{K}_d \mathcal{L}_d}{k_c + k_l d + k_q d^2}(l \cdot n)$$

- n es el vector normal a la superficie
- l es el vector unitario hacia la fuente de luz
- $\mathcal{L}_d$ es la componente difusa de la luz

Reflexión especular Indica que "tan brillante" es la superficie (distinguiendo la superficie de una roca de un metal)

$$\mathcal{I}_s = \frac{\mathcal{K}_s \mathcal{L}_s}{k_c + k_l d + k_q d^2}(v \cdot r)^\alpha$$

- n es la normal a la superficie
- α es el coeficiente de brillo
- l es el vector unitario hacia la luz
- r es el vector unitario reflejado
- $\mathcal{L}_s$ es la componente especular de la luz

Y entonces la magia!

El modelo de Phong junta estas tres componentes y entonces se tiene la siguiente fórmula:

$$\mathcal{I} = \mathcal{K}_a\mathcal{L}_a + \frac{1}{k_c + k_l d + k_q d^2}(\mathcal{K}_d\mathcal{L}_d\mathcal{K}_s(l \cdot n) + \mathcal{L}_s(v \cdot r)^\alpha)$$

Nota: En todas operaciones, es importante notar que los $\mathcal{K}$ son vectores

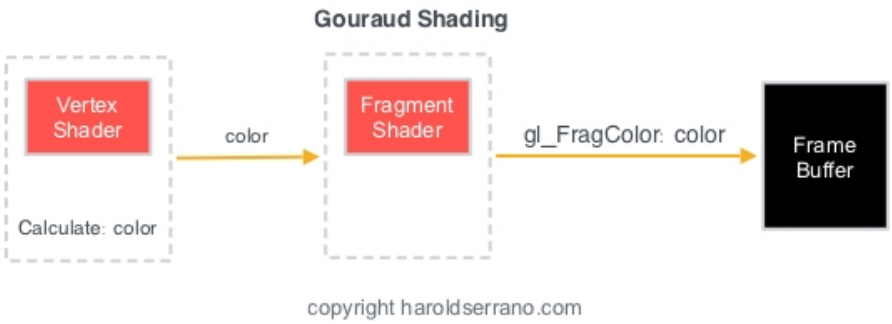
Solo nos falta un detalle!
Tenemos que aplicar el modelo de iluminación sobre nuestro espacio, para ello tenemos 3 estrategias de sombreado:

Sombreado Flat

- La forma más simple de asignación de colores, se elige el color de los triángulos en función del color de alguno de los vértices
- Es el método más económico

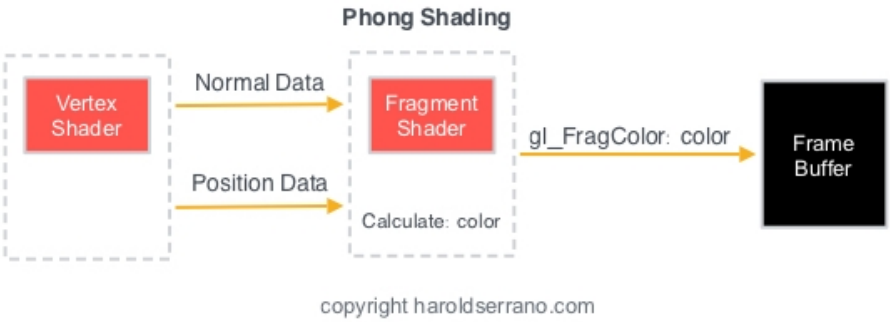
Sombreado Gouraud

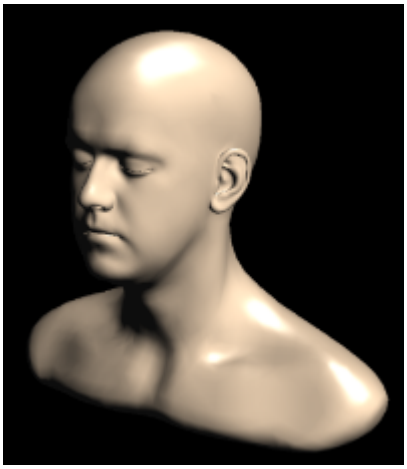
- Realiza interpolación de colores en el interior del triángulo en función de los vértices
- Es más costoso que Flat



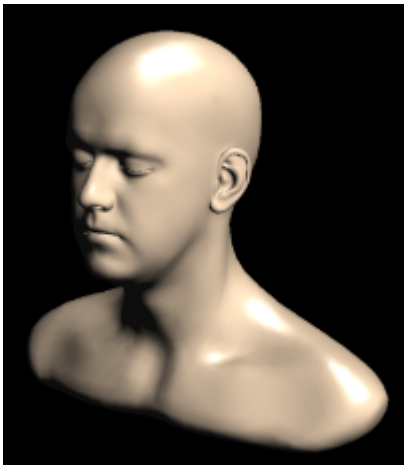
Sombreado Phong

- Realiza interpolación de colores en el interior del triángulo en función de las normales de los vértices a nivel de píxel
- Es el más costoso de los 3





Sombreado Gouraud



Sombreado Phong

Veamos como quedan los shaders utilizando el modelo de Phong

Notar cómo es necesario dar como parámetros los valores de la fórmula antes mencionada

In [ ]:

```
class SimpleFlatShaderProgram():

    def __init__(self):

        vertex_shader = """
            #version 130

            in vec3 position;
            in vec3 color;
            in vec3 normal;

            flat out vec4 vertexColor;

            uniform mat4 model;
            uniform mat4 view;
            uniform mat4 projection;

            uniform vec3 lightPosition;
            uniform vec3 viewPosition;
            uniform vec3 La;
            uniform vec3 Ld;
            uniform vec3 Ls;
            uniform vec3 Ka;
            uniform vec3 Kd;
            uniform vec3 Ks;
            uniform uint shininess;
            uniform float constantAttenuation;
            uniform float linearAttenuation;
            uniform float quadraticAttenuation;

            void main()
            {
                vec3 vertexPos = vec3(model * vec4(position, 1.0));
                gl_Position = projection * view * vec4(vertexPos, 1.0);

                // ambient
                vec3 ambient = Ka * La;

                // diffuse
                vec3 norm = normalize(normal);
                vec3 toLight = lightPosition - vertexPos;
                vec3 lightDir = normalize(toLight);
                float diff = max(dot(norm, lightDir), 0.0);
                vec3 diffuse = Kd * Ld * diff;

                // specular
                vec3 viewDir = normalize(viewPosition - vertexPos);
                vec3 reflectDir = reflect(-lightDir, norm);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
                vec3 specular = Ks * Ls * spec;

                // attenuation
                float distToLight = length(toLight);
                float attenuation = constantAttenuation
                    + linearAttenuation * distToLight
                    + quadraticAttenuation * distToLight * distToLight;
```

```
                vec3 result = (ambient + ((diffuse + specular) / attenuation)) *
                vertexColor = vec4(result, 1.0);
            }
            """

        fragment_shader = """
            #version 130

            flat in vec4 vertexColor;
            out vec4 fragColor;

            void main()
            {
                fragColor = vertexColor;
            }
            """

        self.shaderProgram = OpenGL.GL.shaders.compileProgram(
            OpenGL.GL.shaders.compileShader(vertex_shader, OpenGL.GL.GL_VERTEX_SH
            OpenGL.GL.shaders.compileShader(fragment_shader, OpenGL.GL.GL_FRAGMEN


    def drawShape(self, shape, mode=GL_TRIANGLES):
        assert isinstance(shape, GPUShape)

        # Binding the proper buffers
        glBindVertexArray(shape.vao)
        glBindBuffer(GL_ARRAY_BUFFER, shape.vbo)
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, shape.ebo)

        # 3d vertices + rgb color + 3d normals => 3*4 + 3*4 + 3*4 = 36 bytes
        position = glGetAttribLocation(self.shaderProgram, "position")
        glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 36, ctypes.c_void_
        glEnableVertexAttribArray(position)

        color = glGetAttribLocation(self.shaderProgram, "color")
        glVertexAttribPointer(color, 3, GL_FLOAT, GL_FALSE, 36, ctypes.c_void_p(1
        glEnableVertexAttribArray(color)

        normal = glGetAttribLocation(self.shaderProgram, "normal")
        glVertexAttribPointer(normal, 3, GL_FLOAT, GL_FALSE, 36, ctypes.c_void_p
        glEnableVertexAttribArray(normal)

        # Render the active element buffer with the active shader program
        glDrawElements(mode, shape.size, GL_UNSIGNED_INT, None)
```

Para el sombreado de gouraud ahora se realizan interpolaciones para obtener los parámetros necesitados. Observa que el proceso se realiza en el vertex shader

In [ ]:
```
class SimpleGouraudShaderProgram():

    def __init__(self):
        """
        ...
        void main()
            {
                vec3 vertexPos = vec3(model * vec4(position, 1.0));
                gl_Position = projection * view * vec4(vertexPos, 1.0);

                // ambient
                vec3 ambient = Ka * La;

                // diffuse
                vec3 norm = normalize(normal);
                vec3 toLight = lightPosition - vertexPos;
                vec3 lightDir = normalize(toLight);
                float diff = max(dot(norm, lightDir), 0.0);
                vec3 diffuse = Kd * Ld * diff;

                // specular
                vec3 viewDir = normalize(viewPosition - vertexPos);
                vec3 reflectDir = reflect(-lightDir, norm);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
                vec3 specular = Ks * Ls * spec;

                // attenuation
                float distToLight = length(toLight);
                float attenuation = constantAttenuation
                    + linearAttenuation * distToLight
                    + quadraticAttenuation * distToLight * distToLight;

                vec3 result = (ambient + ((diffuse + specular) / attenuation)) *
                vertexColor = vec4(result, 1.0);
            }
            """
```

...En cambio con Phong, el proceso se cambia al fragment shader!

In [ ]:

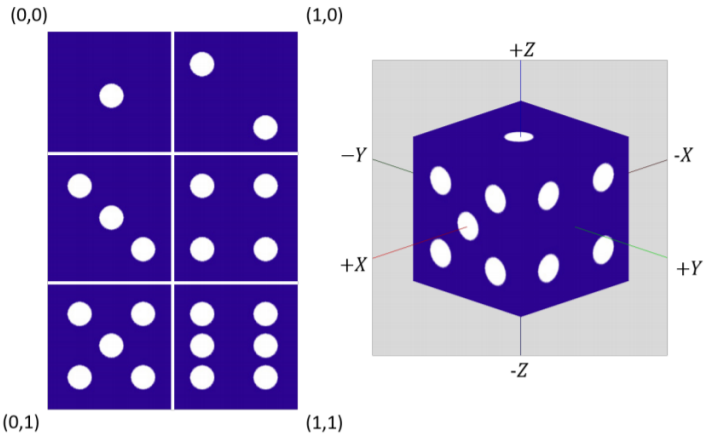
```
class SimplePhongShaderProgram:

    def __init__(self):
        vertex_shader = """
            #version 330 core

            layout (location = 0) in vec3 position;
            layout (location = 1) in vec3 color;
            layout (location = 2) in vec3 normal;

            out vec3 fragPosition;
            out vec3 fragOriginalColor;
            out vec3 fragNormal;

            uniform mat4 model;
            uniform mat4 view;
            uniform mat4 projection;

            void main()
            {
                fragPosition = vec3(model * vec4(position, 1.0));
                fragOriginalColor = color;
                fragNormal = mat3(transpose(inverse(model))) * normal;

                gl_Position = projection * view * vec4(fragPosition, 1.0);
            }
            """

        fragment_shader = """
            #version 330 core

            out vec4 fragColor;

            in vec3 fragNormal;
            in vec3 fragPosition;
            in vec3 fragOriginalColor;

            uniform vec3 lightPosition;
            uniform vec3 viewPosition;
            uniform vec3 La;
            uniform vec3 Ld;
            uniform vec3 Ls;
            uniform vec3 Ka;
            uniform vec3 Kd;
            uniform vec3 Ks;
            uniform uint shininess;
            uniform float constantAttenuation;
            uniform float linearAttenuation;
            uniform float quadraticAttenuation;

            void main()
            {
                // ambient
                vec3 ambient = Ka * La;

                // diffuse
```

```
                // fragment normal has been interpolated, so it does not necessar
                vec3 normalizedNormal = normalize(fragNormal);
                vec3 toLight = lightPosition - fragPosition;
                vec3 lightDir = normalize(toLight);
                float diff = max(dot(normalizedNormal, lightDir), 0.0);
                vec3 diffuse = Kd * Ld * diff;

                // specular
                vec3 viewDir = normalize(viewPosition - fragPosition);
                vec3 reflectDir = reflect(-lightDir, normalizedNormal);
                float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
                vec3 specular = Ks * Ls * spec;

                // attenuation
                float distToLight = length(toLight);
                float attenuation = constantAttenuation
                    + linearAttenuation * distToLight
                    + quadraticAttenuation * distToLight * distToLight;

                vec3 result = (ambient + ((diffuse + specular) / attenuation)) *
                fragColor = vec4(result, 1.0);
            }
            """

        self.shaderProgram = OpenGL.GL.shaders.compileProgram(
            OpenGL.GL.shaders.compileShader(vertex_shader, OpenGL.GL.GL_VERTEX_SH
            OpenGL.GL.shaders.compileShader(fragment_shader, OpenGL.GL.GL_FRAGMEN
```

Y por último...Texturas!

Forma de simplificar el problema de modelar objetos complejos!
Consiste en añadir imágenes a cada triángulo, sin bajar el rendimiento del procesamiento, pues están "pre-cargados"

Pillow (nuestro amigo de las imágenes) toma a las imágenes en coordenadas 0 a 1 y para pasarlas al modelo se debe hacer una "conversión", como la que se muestra a continuación:

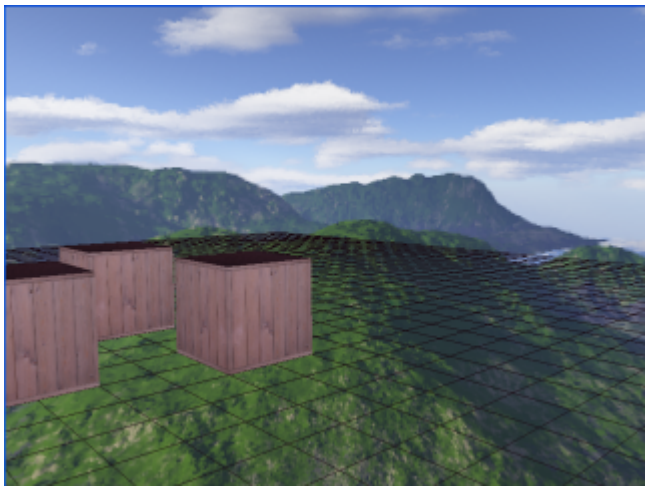


Número 4 queda definido por:

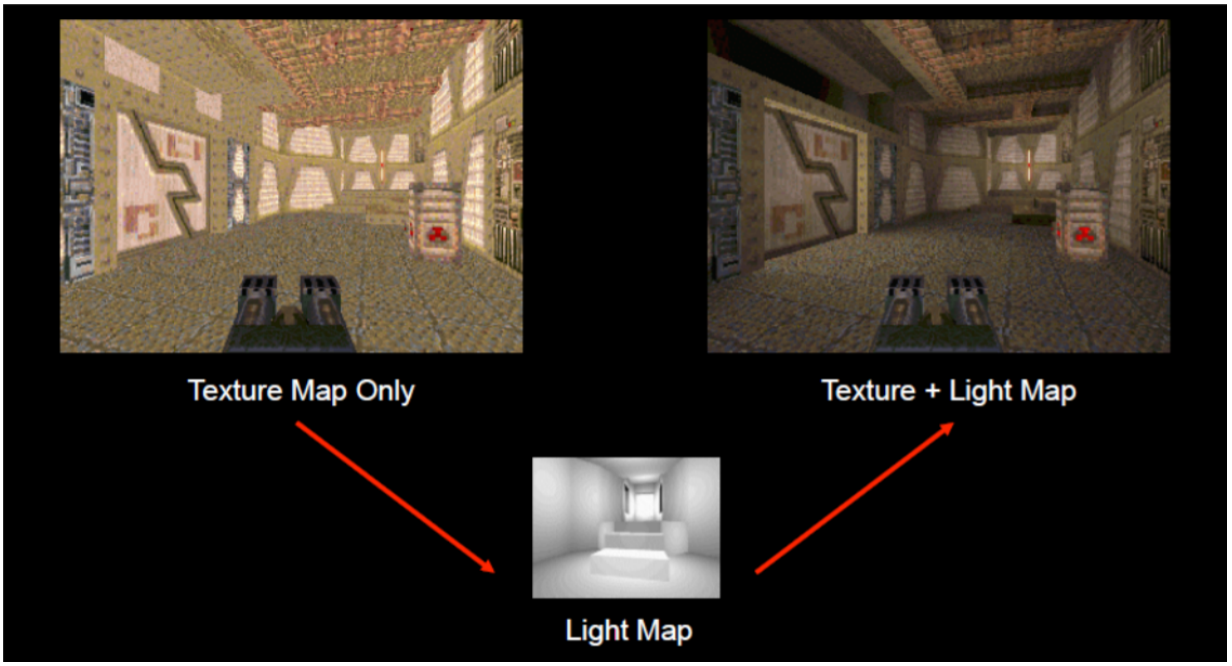
```
vertices = [
...
# positions        # tex coords
-0.5,  0.5, -0.5, 1/2, 2/3,
 0.5,  0.5, -0.5,   1, 2/3,
 0.5,  0.5,  0.5,   1, 1/3,
-0.5,  0.5,  0.5, 1/2, 1/3,
...
]
```

Recordar que siempre se trabajan con triángulos y el orden en que se usen importa!

Algunas cosas que se pueden hacer con Texturas:
Environment Mapping:
Muy útil a la hora de construir fondos, porque permiten dar la noción de que hay una escena de fondo cuando no la hay, el más clásico es el skybox



También se tiene el light mapping, que es muy utilizado para los videojuegos, dado que ayuda a dar iluminación global a la escena de forma pre-computada



Y aún quedan más!

Si quedaste con más ganas de saber sobre iluminación y texturas y como han progresado los algoritmos al respecto, te recomiendo ver el siguiente video

In [9]:
```
from IPython.lib.display import YouTubeVideo
YouTubeVideo('tbsudki8Sro')
```

Out[9]:

Comencemos el auxiliar!

Ahora que hemos vistos los conceptos necesarios para esta sesión, llegó la hora de trabajar lo aprendido :3
De manera que el problema de hoy consta de lo siguiente:

Controller

Importante, basic_shapes es distinto al de cátedra porque se agregan dos modelos nuevos

In [1]:
```
#Los módulos anteriores forman parte del modelo.
# The figures are in the leaves.

import scene_graph as sg
import basic_shapes as bs
import easy_shaders as es
import transformations as tr
import lighting_shaders as ls
import glfw
from OpenGL.GL import *
import numpy as np
import sys

INT_BYTES = 4


# A class to store the application control
class Controller:
    fillPolygon = True
    useShader2 = False

# We will use the global controller as communication with the callback function
controller = Controller() # Here we declare this as a global variable.


def on_key(window, key, scancode, action, mods):

    if action != glfw.PRESS:
        return

    global controller # Declares that we are going to use the global object contr

    if key == glfw.KEY_SPACE:
        controller.fillPolygon = not controller.fillPolygon
        print("Toggle GL_FILL/GL_LINE")

    elif key == glfw.KEY_ESCAPE:
        sys.exit()

    else:
        print('Unknown key')
```

Models

Lo importante es que solo es necesario añadir una componente normal como parámetro para implementar el modelo de iluminación (1 si será con iluminación, 0 si no)

In [2]:

```
def createCar(r1,g1,b1, r2, g2, b2, isNormal):
    if isNormal:
        gpuBlackQuad = es.toGPUShape(bs.createColorNormalsCube(0, 0, 0))
        gpuChasisQuad_color1 = es.toGPUShape(bs.createColorNormalsCube(r1, g1, b
        gpuChasisQuad_color2 = es.toGPUShape(bs.createColorNormalsCube(r2, g2, b
        gpuChasisPrism = es.toGPUShape(bs.createColorNormalTriangularPrism(153 /
    else:
        gpuBlackQuad = es.toGPUShape(bs.createColorCube(0,0,0))
        gpuChasisQuad_color1 = es.toGPUShape(bs.createColorCube(r1,g1,b1))
        gpuChasisQuad_color2 = es.toGPUShape(bs.createColorCube(r2,g2,b2))
        gpuChasisPrism = es.toGPUShape(bs.createColorTriangularPrism(153/255, 204

    # Cheating a single wheel
    wheel = sg.SceneGraphNode("wheel")
    wheel.transform = tr.scale(0.2, 0.8, 0.2)
    wheel.childs += [gpuBlackQuad]

    wheelRotation = sg.SceneGraphNode("wheelRotation")
    wheelRotation.childs += [wheel]

    # Instanciating 2 wheels, for the front and back parts
    frontWheel = sg.SceneGraphNode("frontWheel")
    frontWheel.transform = tr.translate(0.3,0,-0.3)
    frontWheel.childs += [wheelRotation]

    backWheel = sg.SceneGraphNode("backWheel")
    backWheel.transform = tr.translate(-0.3,0,-0.3)
    backWheel.childs += [wheelRotation]

    # Creating the bottom chasis of the car
    bot_chasis = sg.SceneGraphNode("bot_chasis")
    bot_chasis.transform = tr.scale(1.1,0.7,0.1)
    bot_chasis.childs += [gpuChasisQuad_color1]

    # Moving bottom chasis
    moved_b_chasis = sg.SceneGraphNode("moved_b_chasis")
    moved_b_chasis.transform = tr.translate(0, 0, -0.2)
    moved_b_chasis.childs += [bot_chasis]

    # Creating light support
    light_s = sg.SceneGraphNode("light_s")
    light_s.transform = tr.scale(1, 0.1, 0.1)
    light_s.childs += [gpuChasisQuad_color2]

    # Creating right light
    right_light = sg.SceneGraphNode("right_light")
    right_light.transform = tr.translate(0, 0.25, 0)
    right_light.childs += [light_s]

    # Moving right light
    left_light = sg.SceneGraphNode("left_light")
    left_light.transform = tr.translate(0, -0.25, 0)
    left_light.childs += [light_s]

    # Creating center chasis
    center_chasis = sg.SceneGraphNode("center_chasis")
```

```
center_chasis.transform = tr.scale(1, 0.4, 0.15)
center_chasis.childs += [gpuChasisQuad_color1]

# Moving center chasis
m_center_chasis = sg.SceneGraphNode("m_center_chasis")
m_center_chasis.transform = tr.translate(0.05, 0, 0)
m_center_chasis.childs += [center_chasis]

# Creating center quad
center_quad = sg.SceneGraphNode("center_quad")
center_quad.transform = tr.scale(0.26, 0.5, 0.2)
center_quad.childs += [gpuChasisQuad_color2]

# Moving center quad
m_center_quad = sg.SceneGraphNode("m_center_quad")
m_center_quad.transform = tr.translate(-0.07, 0, 0.1)
m_center_quad.childs += [center_quad]

# Creating front wind shield
f_wind_shield = sg.SceneGraphNode("f_wind_shield")
f_wind_shield.transform = tr.scale(0.25, 0.5, 0.2)
f_wind_shield.childs += [gpuChasisPrism]

# Moving front wind shield
m_f_wind_shield = sg.SceneGraphNode("m_f_wind_shield")
m_f_wind_shield.transform = tr.translate(0.2, 0, 0.1)
m_f_wind_shield.childs += [f_wind_shield]

# Creating back wind shield
b_wind_shield = sg.SceneGraphNode("b_wind_shield")
b_wind_shield.transform = tr.scale(0.25, 0.5, 0.2)
b_wind_shield.childs += [gpuChasisPrism]

# Rotate back wind shield
r_b_wind_shield = sg.SceneGraphNode("r_b_wind_shield")
r_b_wind_shield.transform = tr.rotationZ(np.pi)
r_b_wind_shield.childs += [b_wind_shield]

# Moving back wind shield
m_b_wind_shield = sg.SceneGraphNode("m_b_wind_shield")
m_b_wind_shield.transform = tr.translate(-0.3, 0, 0.1)
m_b_wind_shield.childs += [r_b_wind_shield]

# Joining chasis parts
complete_chasis = sg.SceneGraphNode("complete_chasis")
complete_chasis.childs += [moved_b_chasis]
complete_chasis.childs += [right_light]
complete_chasis.childs += [left_light]
complete_chasis.childs += [m_center_chasis]
complete_chasis.childs += [m_center_quad]
complete_chasis.childs += [m_b_wind_shield]
complete_chasis.childs += [m_f_wind_shield]


# All pieces together
car = sg.SceneGraphNode("car")
car.childs += [complete_chasis]
```

```
    car.childs += [frontWheel]
    car.childs += [backWheel]

    return car
```

View

In [3]:

```
if __name__ == "__main__":

    # Initialize glfw
    if not glfw.init():
        sys.exit()

    width = 800
    height = 600

    window = glfw.create_window(width, height, "Window name", None, None)

    if not window:
        glfw.terminate()
        sys.exit()

    glfw.make_context_current(window)

    # Connecting the callback function 'on_key' to handle keyboard events
    glfw.set_key_callback(window, on_key)

    # Assembling the shader program (pipeline) with both shaders
    phongPipeline = ls.SimplePhongShaderProgram()

    # Telling OpenGL to use our shader program
    glUseProgram(phongPipeline.shaderProgram)

    # Setting up the clear screen color
    glClearColor(0.85, 0.85, 0.85, 1.0)

    # Creating shapes on GPU memory
    car = createCar(252 / 255, 246 / 255, 246 / 255, 255 / 255, 153 / 255, 153 /

    glEnable(GL_DEPTH_TEST)

    # Our shapes here are always fully painted
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)

    normal_view = tr.lookAt(
        np.array([5, 5, 6]),
        np.array([0, 0, 0]),
        np.array([0, 0, 1])
    )

    projection = tr.ortho(-1, 1, -1, 1, 0.1, 100)
    projection = tr.perspective(45, float(width) / float(height), 0.1, 100)

    model = tr.identity()

    while not glfw.window_should_close(window):
        # Using GLFW to check for input events
        glfw.poll_events()

        # Filling or not the shapes depending on the controller state
        if controller.fillPolygon:
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
        else:
```

```
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)

        # Clearing the screen in both, color and depth
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

        glUseProgram(phongPipeline.shaderProgram)

        # White light in all components: ambient, diffuse and specular.
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "La"), 1.0
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "Ld"), 1.0
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "Ls"), 1.0

        # Object is barely visible at only ambient. Diffuse behavior is slightly
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "Ka"), 0.2
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "Kd"), 0.9
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "Ks"), 1.0

        # TO DO: Explore different parameter combinations to understand their ef

        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "lightPosi
        glUniform3f(glGetUniformLocation(phongPipeline.shaderProgram, "viewPosit
        glUniform1ui(glGetUniformLocation(phongPipeline.shaderProgram, "shinines

        glUniform1f(glGetUniformLocation(phongPipeline.shaderProgram, "constantA
        glUniform1f(glGetUniformLocation(phongPipeline.shaderProgram, "linearAtt
        glUniform1f(glGetUniformLocation(phongPipeline.shaderProgram, "quadratic

        glUniformMatrix4fv(glGetUniformLocation(phongPipeline.shaderProgram, "pr
        glUniformMatrix4fv(glGetUniformLocation(phongPipeline.shaderProgram, "vi
        glUniformMatrix4fv(glGetUniformLocation(phongPipeline.shaderProgram, "mo

        sg.drawSceneGraphNode(car, phongPipeline, "model")

        # Once the render is done, buffers are swapped, showing only the complete
        glfw.swap_buffers(window)

    glfw.terminate()
```

Queremos construir una escena de autitos en 3D que se muevan de alguna forma, al menos con un suelo lindo para mostrar los conceptos de la sesión. Por lo que el trabajo se dividirá en las siguientes secciones:
P1 - Crear suelo a partir de una textura (puede ser la que se encuentra dentro de los archivos, "ground.jpg")

P2 - Mostrar un auto que se mueva con una trayectoria circunferencial (el modelo de un auto ya está implementado), la mejor forma es utilizar coordenadas esféricas (con las funciones seno, coseno y un ángulo definido por ti)

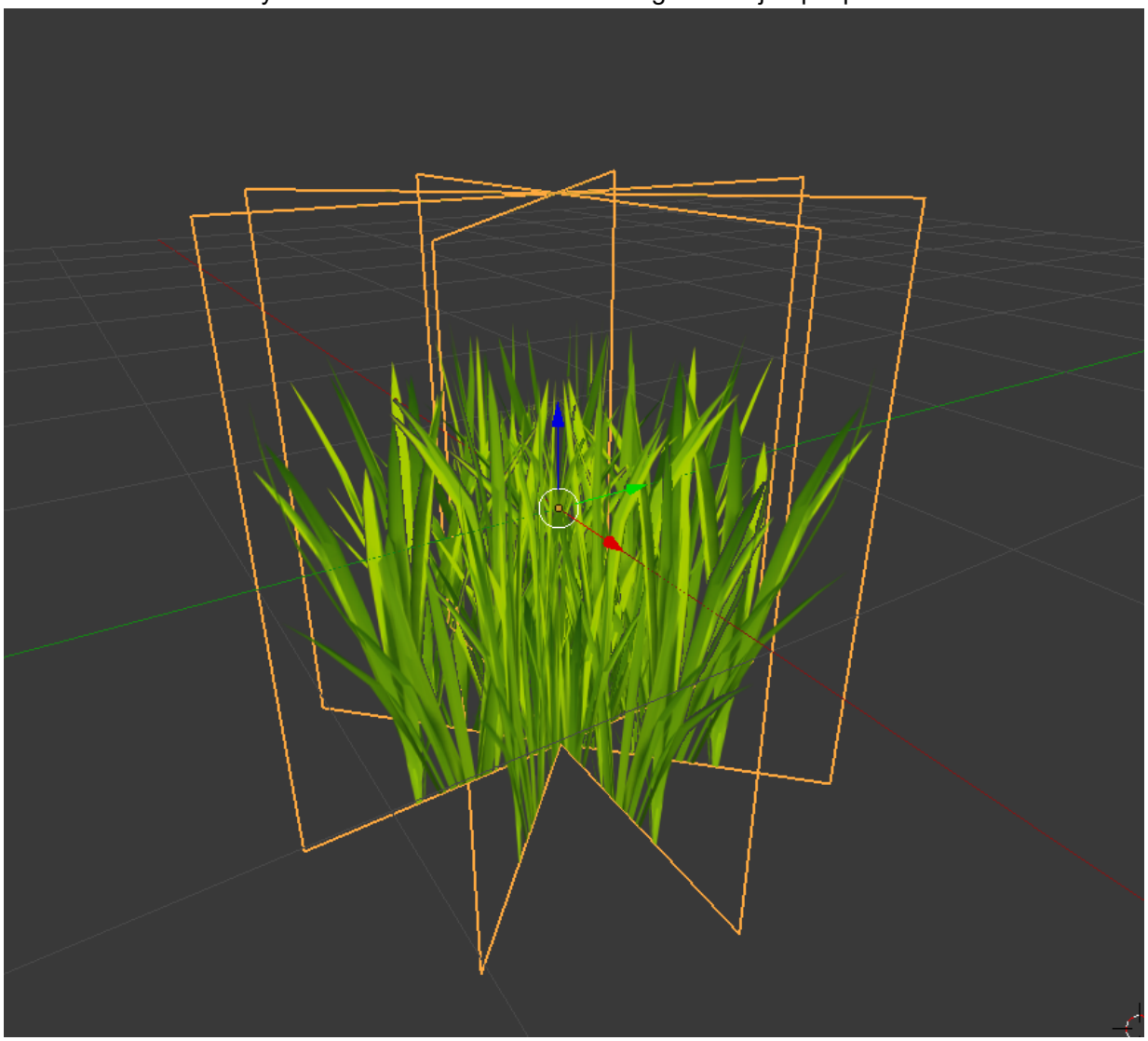
P3 - Agregar una textura en el ambiente, de preferencia a ricardo cerca del extremo de la pantalla, o un skybox, tu decides :0 (imagina a ricardo como skybox xD)

P4 - Implementar una cámara nueva que visualice el auto en primera persona (o tercera), para

ello es necesario ubicarla cámara en la posición relativa del auto, con dirección tangencial a la trayectoria (o una escogida convenientemente), debes dar la posibilidad de intercambiar las cámaras.

P5 - Modifica la posición de la fuente de luz para que se mueva según una función periódica (seno, coseno, blah)

P6 (Décima) Billboard!: Utilizando una imagen de árbol y un fondo transparente, modela un árbol utilizando dos cuadrados intersectados. Así, podrás ver el árbol en cualquier ángulo. Crea varias instancias del modelo y ubícalas en la escena. Mira el siguiente ejemplo para ilustrar:



In [ ]: