



Introducción a OpenGL 4

Computación Gráfica

Marc-Antoine Le Guen



OpenGL

- **OpenGL es :**
 - **Open Graphics Library**
 - **API (Application Programming Interface)**
 - **Multi Platform**
 - **Rendering 2D/3D**
 - **Interactúa con el GPU**
- **OpenGL NO es :**
 - **sistema de creación de ventana**
 - **sistema de interfaz de usuario**
 - **no contiene el framebuffer**
 - **gestionado por el sistema de ventana (GLUT/Qt/GLFW :))**



Historia

- 1992 Silicon graphics
- 2006 Grupo Khronos - OpenGL 2.1
 - Competencia directa es DirectX
- 2008 OpenGL 3.0 - Declaró muchas funcionalidades obsoletas
- 2014 OpenGL 4.5
- 2016 Vulkan



Versiones

- OpenGL 2.1
 - Más accesible que las últimas versiones
- OpenGL > 3.3 (4.5)
 - Imperativo usar una tarjeta gráfica (nvidia/ AMD)
 - GLM (OpenGL Mathematics)
 - operaciones sobre matrices optimizadas
 - shader programming language (glsl)



Versiones

- OpenGL ES (Embedded Systems)
 - Versión depurada de OpenGL 2.0
 - 2.0 Mayoría de los smartphones actuales
 - 3.0 Últimos smartphones de gama alta
 - 3.1 Procesadores gráficos específicos
 - Basado sobre los shaders
 - Recomendando el uso de LibGDX
- WebGL
 - Basado sobre OpenGL ES
 - Usar Three.js
 - Compatible Mozilla Firefox Chrome



Librerías

GLU Some additional functions for OpenGL programs

GLUT The OpenGL utility toolkit.

freeglut Open source alternative to GLUT

GLUI a GUI toolkit made with GLUT

SDL The Simple DirectMedia Layer

Glee The OpenGL Easy Extension library.

GLEW The OpenGL Extension Wrangler Library.

GLM C++ mathematics toolkit for OpenGL based on the GLSL specification.

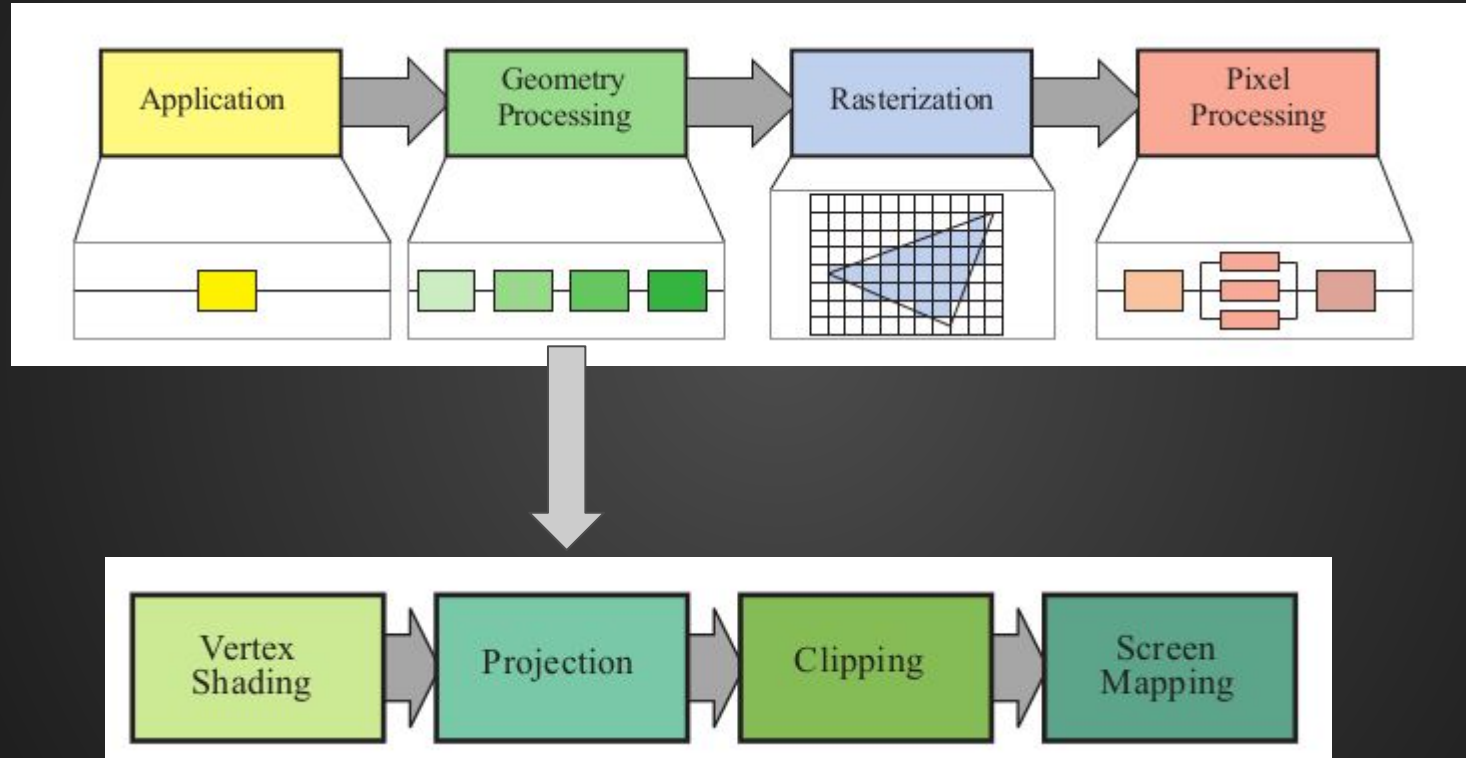
SFML Simple and Fast Multimedia Library.

JOGL Java bindings for OpenGL API.

GLFW Open Source, multi-platform library for creating windows with OpenGL



Pipeline



Pipeline

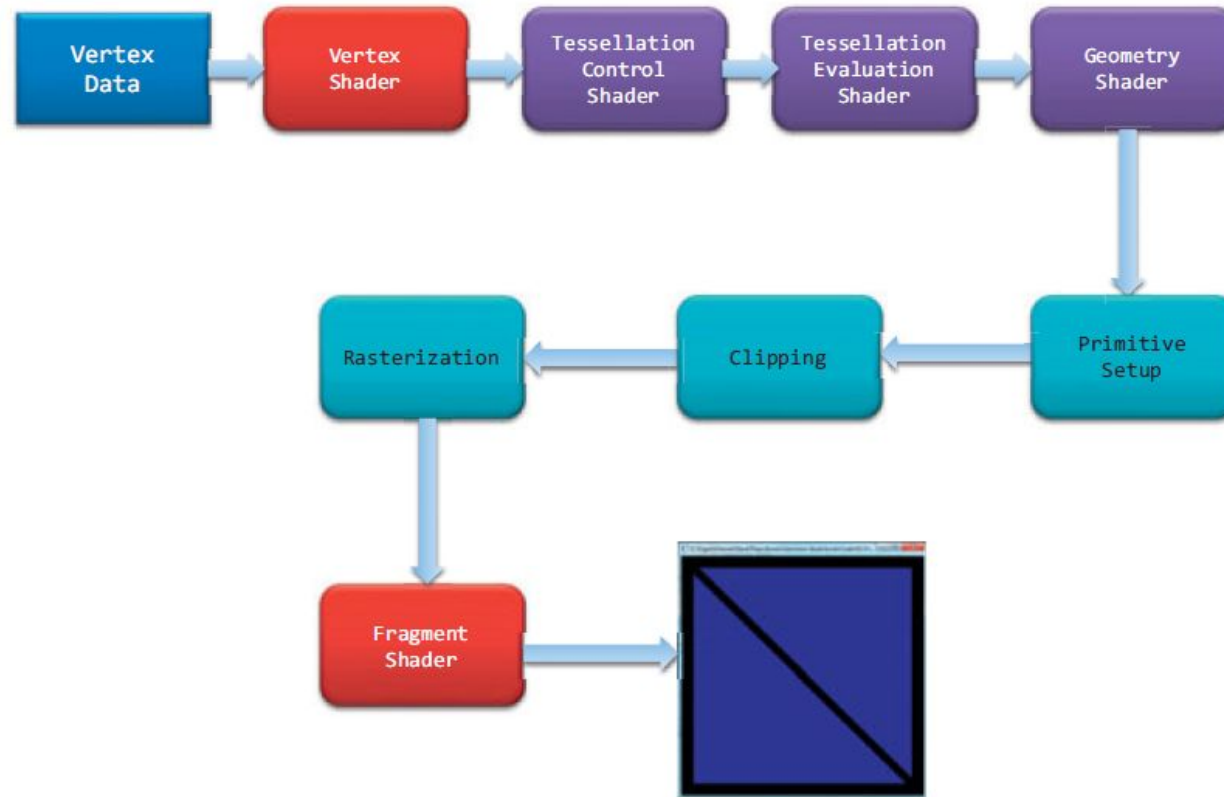


Figure 1.2 The OpenGL pipeline



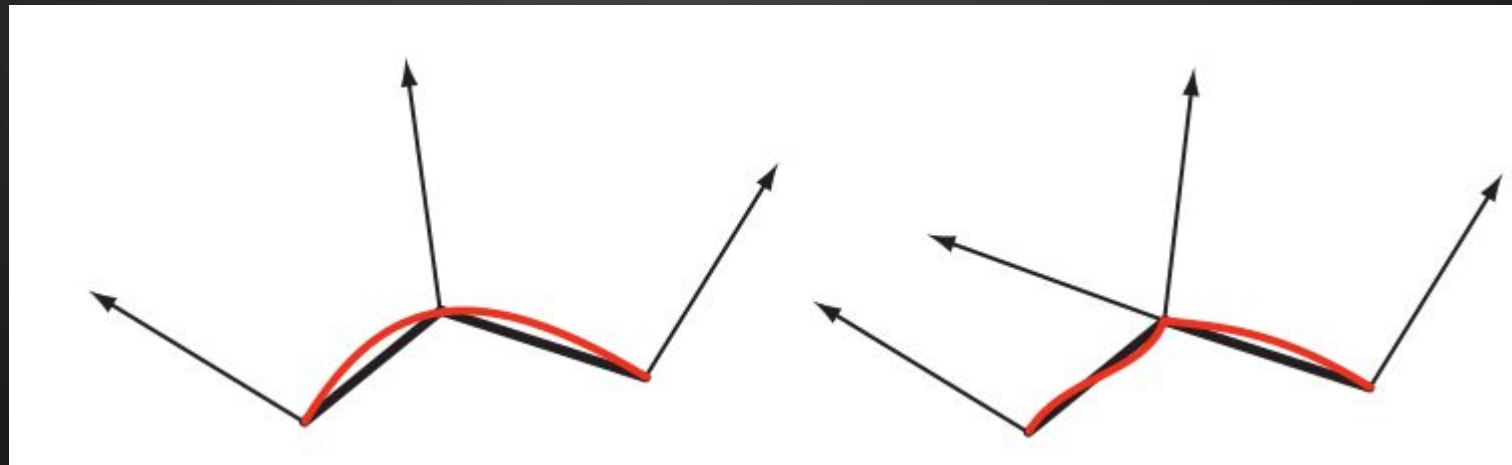
Pipeline

- Especificar los datos para la construcción de formas a partir de primitivas geométricas de OpenGL.
- Ejecutar varios shaders para realizar cálculos sobre las primitivas de entrada para determinar sus posiciones, colores o otros atributos de rendering.
- Convertir la descripción matemática de las primitivas de entrada en píxeles para visualizar en la pantalla. Este proceso se llama la rasterización.
- Finalmente ejecutar el fragment shader para cada pixel generado lo que determinará el color final del pixel.
- Adicionalmente es posible realizar per-fragment operaciones para determinar si el objeto asociado al píxel es visible o mezclar el pixel con el color actual de pantalla a esta posición.



Pipeline

- Vertex shaders
 - Programa que se ejecutará para cada vértice
 - manipular las propiedades de un vértice
 - color, posición, coordenadas de texturas
 - movimiento, color, iluminación



Pipeline

- Tessellation shaders
 - Incrementar la cantidad de primitivas geométricas
 - Mejorar el rendering de objetos 3D
 - potencialmente dos shaders



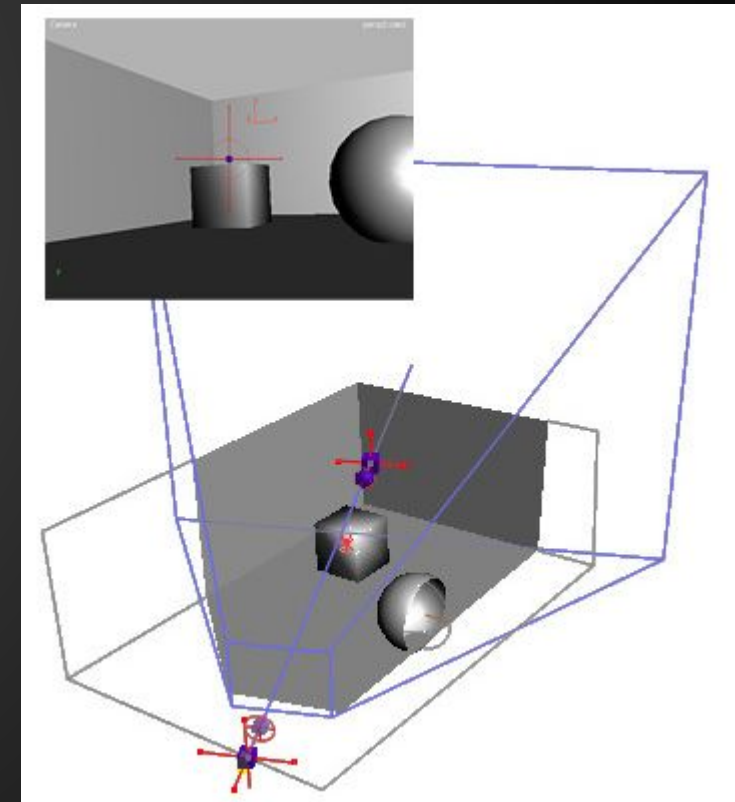
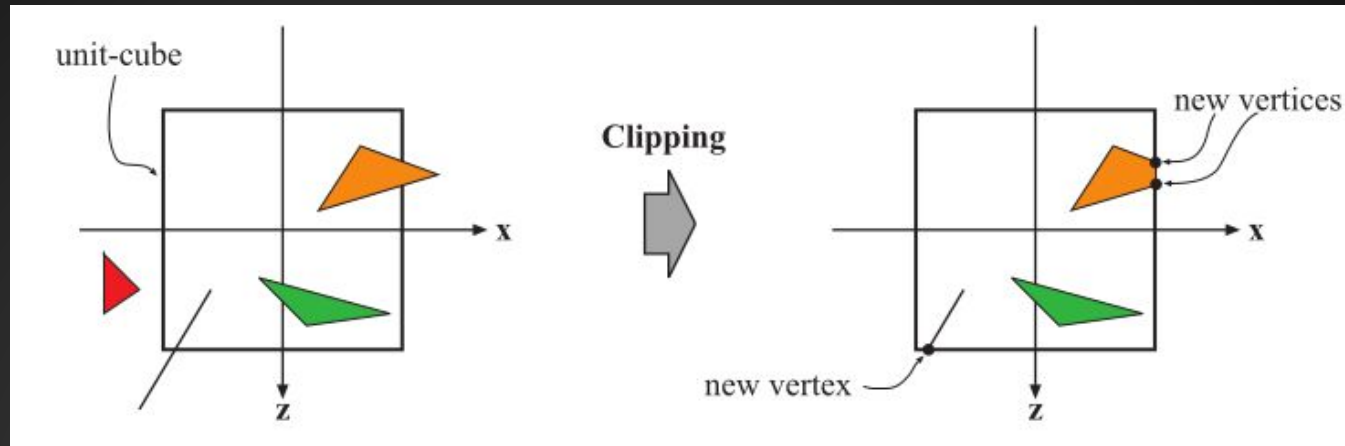
Pipeline

- Primitive Assembly
 - Después de estos previos tratamientos de los vértices por los shaders, cada vértice es organizado en su primitiva geométrica asociada para la fase de clipping y rasterización.



Pipeline

- Clipping
 - A veces los vértices se encuentran fuera de viewport (la ventana de visualización) y causa la modificación de la primitiva geométrica asociada a dichos vértices, para que ninguno de sus píxeles queden fuera del viewport. Esta operación de clipping es automáticamente realizada por OpenGL



Pipeline

- Fragment Shaders
 - manipular color por fragment (pixels)
 - Color desde Texturas
 - efectos especiales, iluminación etc.



Pipeline

- Per-Fragment operación
 - Procesamiento final de cada fragment
 - Determinación de la visibilidad de cada fragmento: depth test (z-buffering)
 - pixeles translúcidos (color será mezclado con el color del framebuffer)
 - efectos especiales, iluminación etc.



Shaders

```
//vertex

uniform float time;

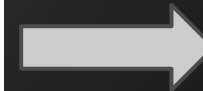
void main(void)
{
    vec4 v = vec4(gl_Vertex);
    v.z = sin(5.0*v.x + time*0.01)*0.25;
    gl_Position =
gl_ModelViewProjectionMatrix * v;

}
```



```
//fragment

varying float intensity;
void main()
{
    vec4 color;
    if (intensity > 0.95)
        color = vec4(1.0,0.5,0.5,1.0);
    else if (intensity > 0.5)
        color = vec4(0.6,0.3,0.3,1.0);
    else if (intensity > 0.25)
        color = vec4(0.4,0.2,0.2,1.0);
    else
        color = vec4(0.2,0.1,0.1,1.0);
    gl_FragColor = color;
}
```



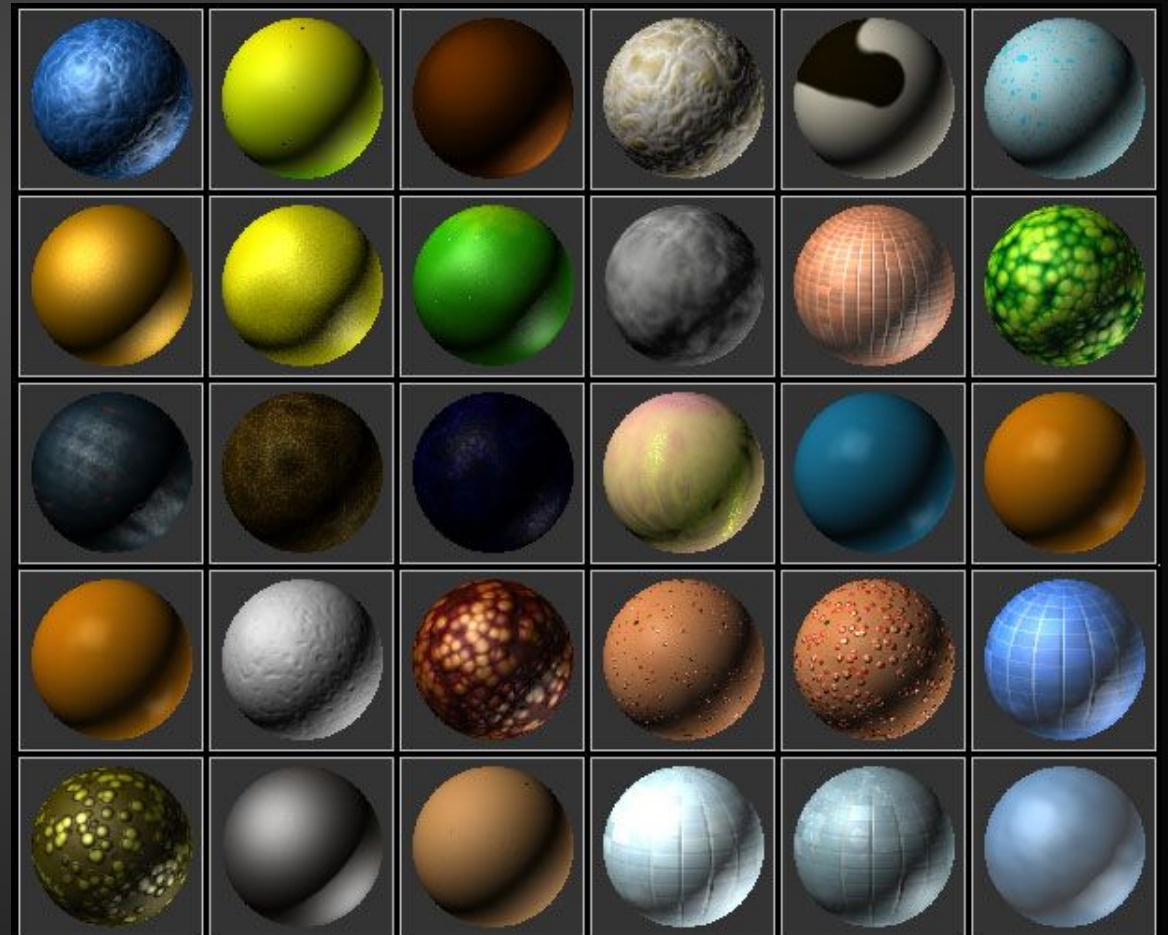
Shaders

OpenGL 4 Shading Language Cookbook - Second Edition

David Wolff

Herramienta : Shader maker

<http://shadermaker.codeplex.com/>



OpenGL - máquina de estado finito

- OpenGL provee variables almacenadas como “globales”
 - Rendering Mode (wireframe / solid)
 - glEnable() glDisable()
 - ...



Funciones de OpenGL

nombre de la
funcion

Dimension

`glVertex3f(x,y,z)`

3 floats

Un puntero hacia un array de
3 floats

Pertenece
a OpenGL

Typo de
parametro



OpenGL tipos de datos

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield



GLFW

- Crear un contexto OpenGL
- Crear ventanas (múltiples si es necesario)
- Gestionar eventos
 - teclado
 - mouse
 - joystick
- basic GUI



GLFW

- glfwInit() inicializa la librería GLFW
- glfwTerminate() al momento de terminar el programa permite liberar la memoria alocada para la libreria

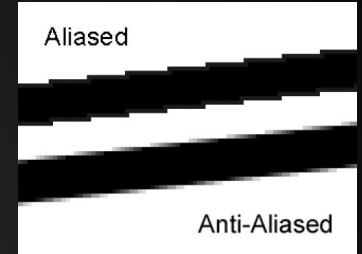
```
if (!glfwInit())
{
    fprintf(stderr, "Failed to initialize GLFW\n");
    return -1;
}

glfwTerminate();
```



GLFW

Antialiasing 4x



Redimensionar la ventana - False

```
glfwWindowHint(GLFW_SAMPLES, 4);  
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);  
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);  
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 4);  
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_P  
ROFILE);
```

Version OpenGL 4.4

Versión OpenGL **Core** -
No retrocompatibilidad
+ ultimas
funcionalidades



GLFW

Tamaño ventana
(width, height)

Nombre de la
ventana

Monitor (en caso de
tener varios- possible
usar `glfwGetPrimaryMonitor()`.)

Otra ventana para
compartir recursos

```
window = glfwCreateWindow(1024, 768, " Basics ", NULL, NULL);  
if (window == NULL){  
    fprintf(stderr, "Failed to open GLFW window. Not 4.4 compatible ");  
    glfwTerminate();  
    return -1;  
}
```

Verificar si se pudo
crear la ventana y el
contexto OpenGL en
versión 4.4



GLFW

Escoger la ventana en la cual trabajaremos

```
glfwMakeContextCurrent(window);  
glfwSwapInterval(1);  
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

Vsync (vertical synchronization) -
Cantidad de update de la pantalla despues de swap el buffer

Asegurarnos que recuperamos los inputs del teclado



GLFW

Bucle de rendering

Limpiar el buffer

```
do  
{  
    // Clear the screen  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
  
    // Draw objects  
  
    glfwSwapBuffers(window);  
    glfwPollEvents();  
  
} // Check if the ESC key was pressed or the window was closed  
while (glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&  
glfwWindowShouldClose(window) == 0);
```

swap el buffer

Procesar los
eventos recibidos

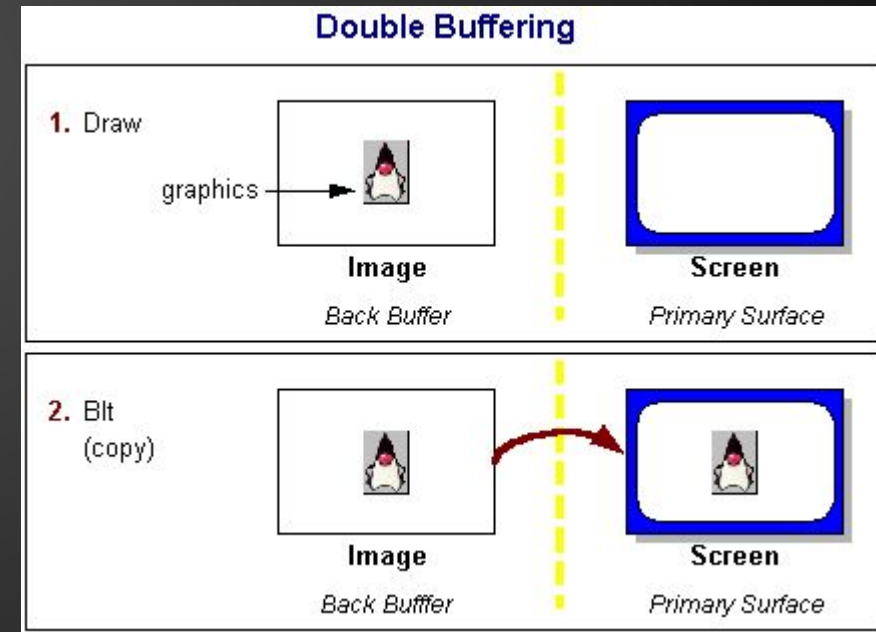
Verificar si hemos
apretado ESC o cerrado
la ventana, sino
seguimos dibujando



Opengl - Double Buffering


- El doble Buffer permite crear una imagen mientras se esta visualizando otra
- Evita efectos de tearing

`glfwSwapBuffers(window);`



Primitivas geométricas

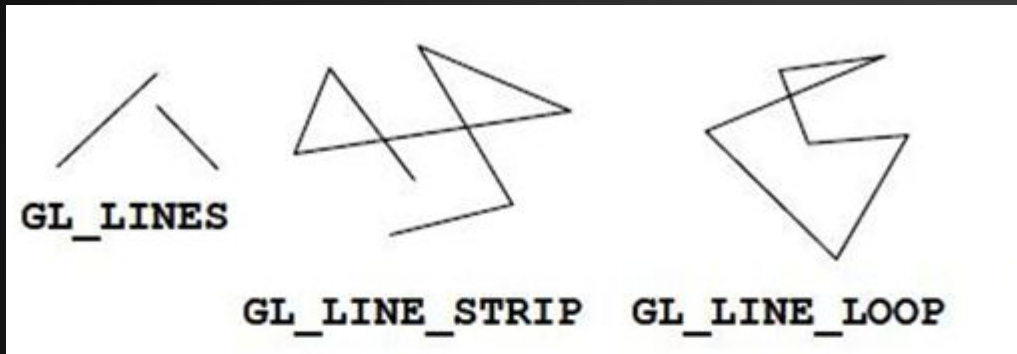
```
void glPointSize(GLfloat size);  
GL_PROGRAM_POINT_SIZE is  
not enabled.
```



GL_POINTS



Primitivas geométricas



```
void glLineWidth(GLfloat width);
```

Primitivas geométricas

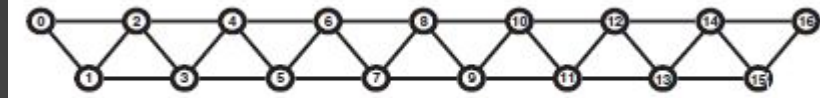
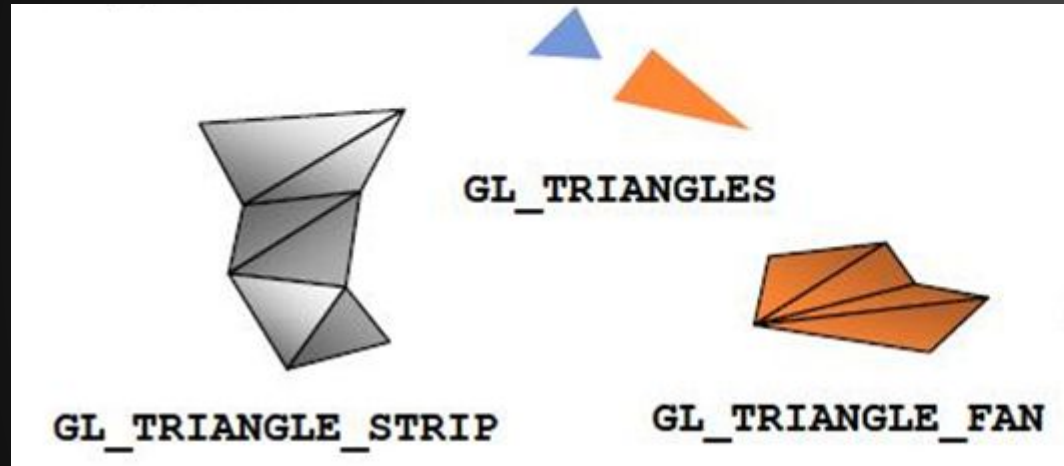


Figure 3.1 Vertex layout for a triangle strip

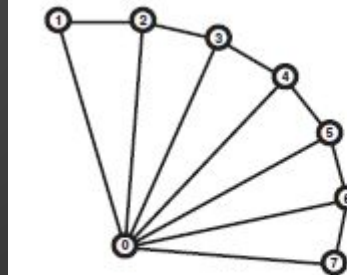
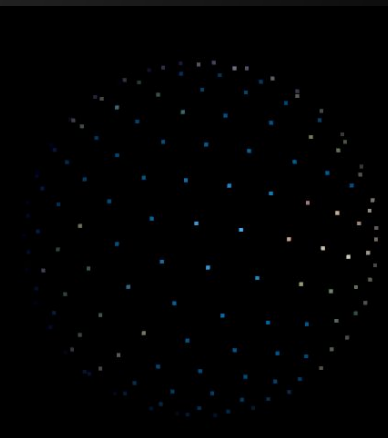
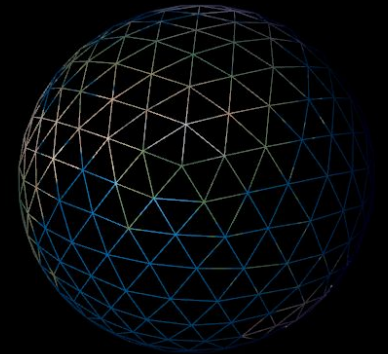
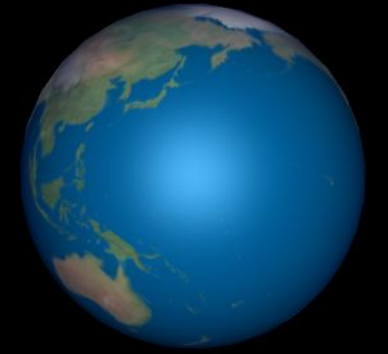


Figure 3.2 Vertex layout for a triangle fan

Rendering de polígonos

- Front and Back side
- `void glPolygonMode(GLenum face, GLenum mode);`
 - face
 - `GL_FRONT / GL_FRONT_AND_BACK`
 - mode
 - `GL_FILL`
 - `GL_LINE`
 - `GL_POINT`



Rendering de polígonos

- Front and Back side
 - Un polígono que aparece en la pantalla en el sentido contra reloj es considerado como FRONT sino como BACK (winding)
 - Este es método por defecto de OpenGL
 - Se puede **invertir** utilizando
 - `void glFrontFace(GLenum mode);`
 - `GL_CCW`
 - `GL_CW`

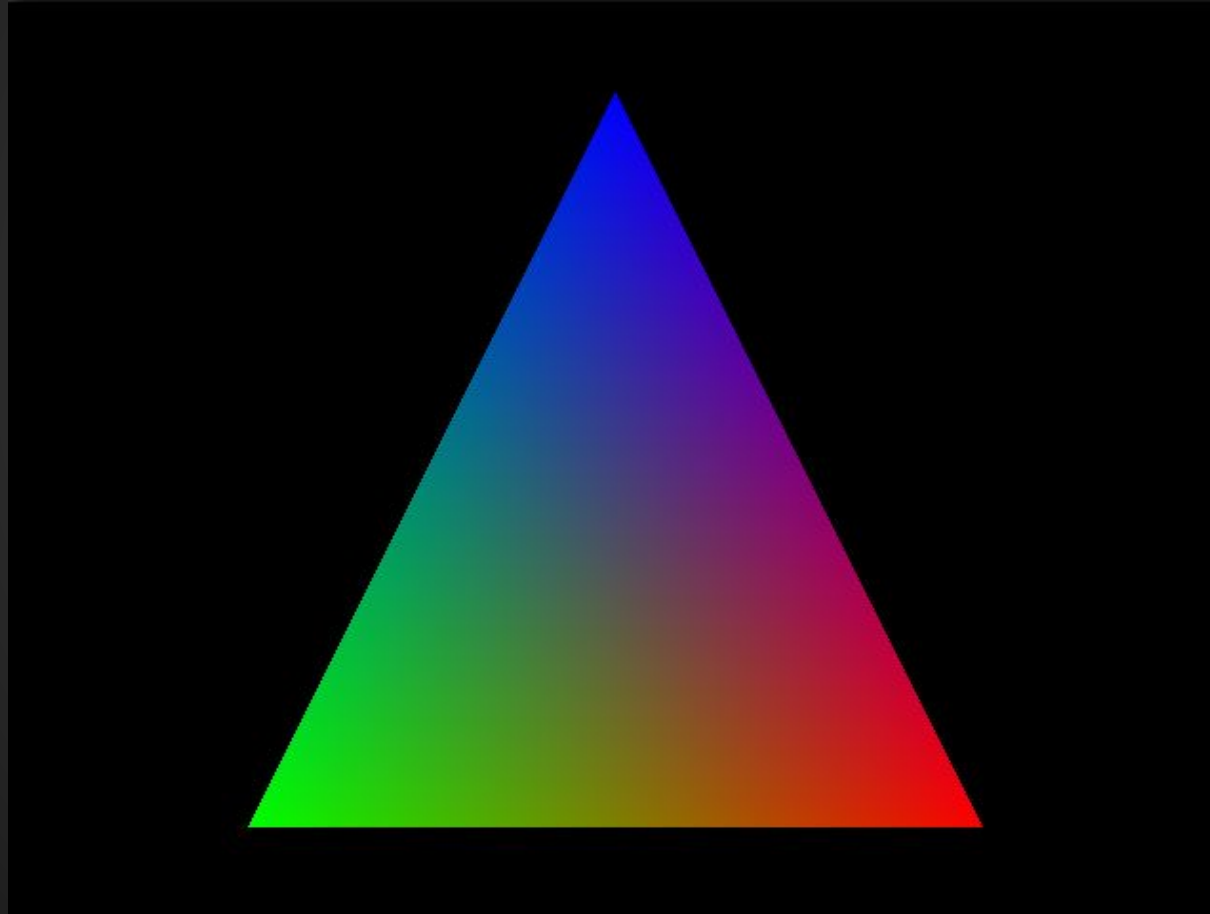


Rendering de polígonos

- Front and Back side
 - Culling : ocultar los polígonos Front or/and Back
 - glEnable(GL_CULL_FACE)
 - void glCullFace(GLenum mode);
 - GL_FRONT
 - GL_BACK



Gestión de los colores



Visualizar 2 triángulos

Después de inicializar GLFW

- InitGL()

```
void initGL()
{
    // Dark background
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // Enable depth test
    glEnable(GL_DEPTH_TEST);
    // Accept fragment if it closer to the camera than the former one
    glDepthFunc(GL_LESS);
    // Cull triangles which normal is not towards the camera
    glEnable(GL_CULL_FACE);
}
```

Color de fondo

Depth test - test de profundidad

Solo se acepta el fragment más cerca de la cámara

Ocultar los polígonos "Back"



Visualizar 2 triángulos

Creación de la geometría

Alocación de un Vertex
Array Object VAO

```
void geometryCreate()
{
    glGenVertexArrays(1, &vertexArrayID);
    glBindVertexArray(vertexArrayID);
```

activamos el VAO

```
    GLfloat vertices[NumVertices][2] = {
        { -0.90, -0.90 }, // Triangle 1
        { 0.85, -0.90 },
        { -0.90, 0.85 },
        { 0.90, -0.85 }, // Triangle 2
        { 0.90, 0.90 },
        { -0.85, 0.90 }
    };
    . . . . .
}
```

Creamos una serie
de vértices 2D (6 = 2
triángulos)



Visualizar 2 triángulos

Creación de la geometría

Creación de un vertex buffer object VBO que contendrá los datos del VAO

```
void geometryCreate()
{
    . . . . .
    glGenBuffers(1, &vertexbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);

    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices, GL_STATIC_DRAW);

    . . . . .
}
```

Volvemos activo el VBO

Inicialización del VBO con los vértices.
GL_STATIC_DRAW, significa que no cambiaras los datos contenido en *vertices*.



Visualizar 2 triángulos

Creación de la geometría

Creación y activación
de shaders

```
void geometryCreate()
{
    . . . . .

    ShaderInfo shaders[] = {
        { GL_VERTEX_SHADER, "triangles.vert" },
        { GL_FRAGMENT_SHADER, "triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders(shaders);
    glUseProgram(program);

    glVertexAttribPointer(0, 2, GL_FLOAT,
        GL_FALSE, 0, BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);

}
```

Pasar la información de
los vértices al vertex
shader



Visualizar 2 triângulos

```
//vertex shader
#version 430 core
layout(location = 0) in vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
```

```
//fragment shader
#version 430 core
out vec4 Color;
void main()
{
    Color = vec4(0.0, 0.0, 1.0, 1.0);
}
```



OpenGL - Render

La rutina `display()` realiza el rendering. Casi todas las funciones de `display` tienen estas 3 etapas presentes en nuestro ejemplo.

1. Limpiar el buffer `glClear()`.
2. Las llamadas de OpenGL para hacer el rendering del objeto.
3. Pedir que la imagen sea visualizada en la pantalla.



Visualizar 2 triángulos

Visualizar la geometría

Limpiar el buffer

Activar el vertex array object

Dibujar la geometría

Todas las llamadas previas a OpenGL son enviadas a OpenGL y procesadas

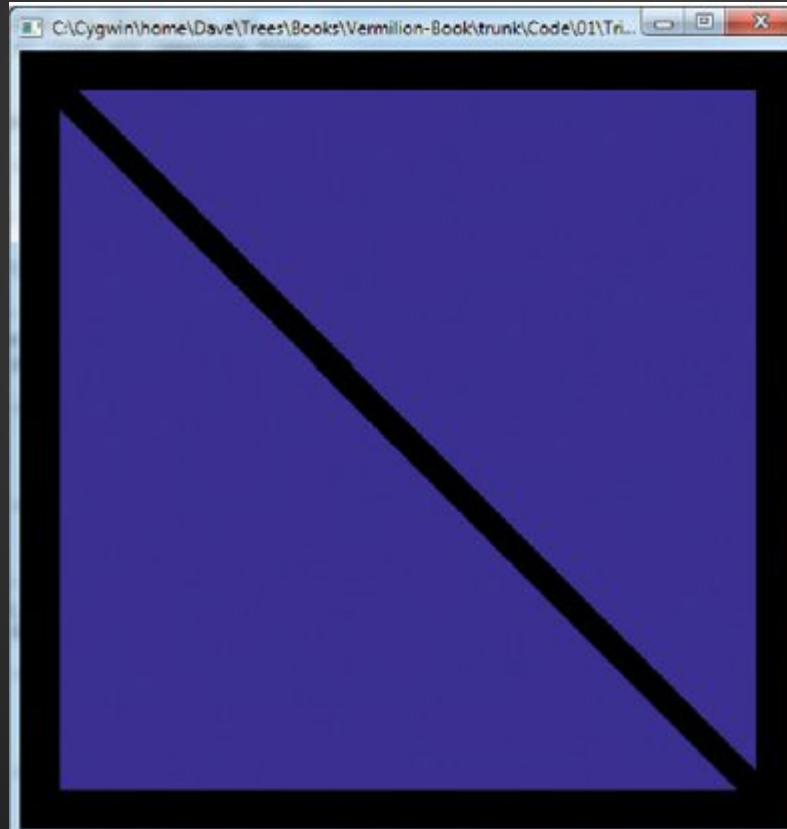
```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBindVertexArray(vertexArrayID);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glFlush();

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```



Visualizar 2 triángulos

Visualizar la geometría



Activar/Desactivar capacidad

- `glEnable(GLenum capability);`
 - Activa la capacidad
- `glDisable(GLenum capability);`
 - Desactiva la capacidad
- `glIsEnabled(GLenum capability);`
 - Retorna si la capacidad es activada
- Ejemplo

`glEnable(GL_BLEND); //Activa el uso de la transparencia`



OpenGL - 3D engine

- **Blender**
- **Ogre 3D**
- Id Tech 2 (Quake)
- Irrlicht
- OpenSceneGraph



Ogre 3D



Irrlicht



Tutoriales

- Neon Helium <http://nehe.gamedev.net/>
- <http://www.opengl-tutorial.org/>
- <http://www.lighthouse3d.com/>



Bibliografía

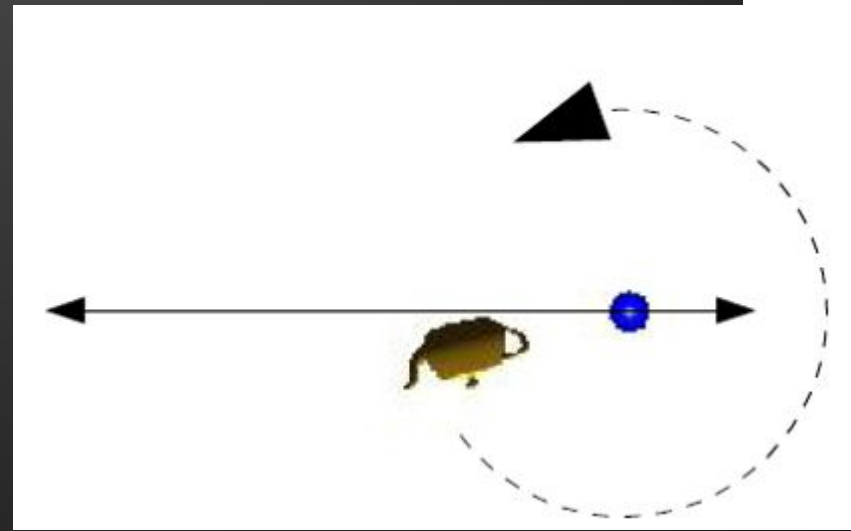
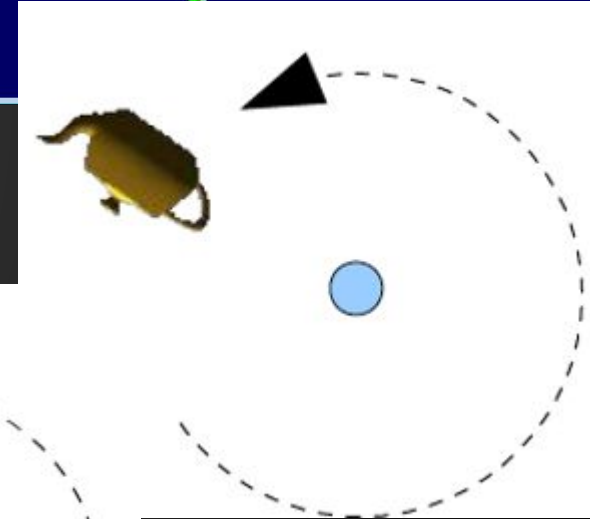
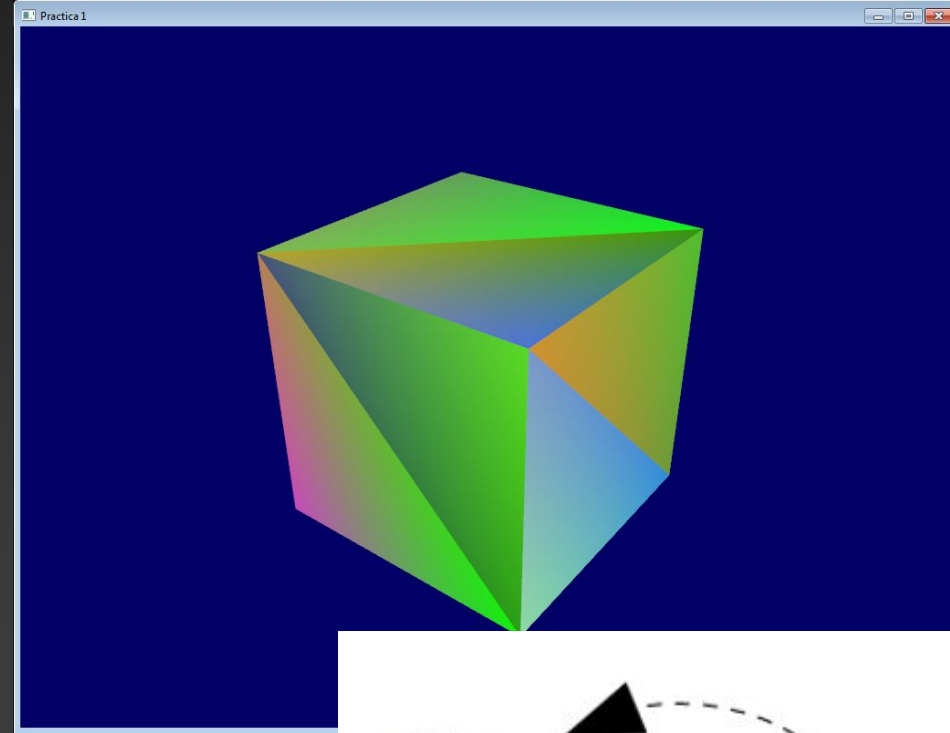
Graham Sellers, Richard S. Wright, Nicholas Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference (6th Edition)*

Dave Shreiner, Graham Sellers, John M. Kessenich, Bill M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3 (8th Edition) Paperback*



Práctica

- Entender la estructura de un programa OpenGL
 - Entender la utilidad de la librería glm
 - Entender el funcionamiento de GLEW
 - Manipular la cámara
 - Realizar una animación de rotación del cubo según el eje z
 - Además de esta animación agregar un va y viene del centro de rotación
-
- [Descarga](#)



Libros

