

# **Introducción a la programación en OpenGL**

# ¿Qué es OpenGL?

- Es una interfaz para la generación de gráficos (Graphics rendering API)
  - Imágenes de alta calidad generadas a partir de primitivas geométricas.
  - Independiente del sistema de ventanas
  - Independiente del sistema operativo.

# ¿Qué es OpenGL?

- Inicialmente desarrollado por Silicon Graphics Inc.
- OpenGL es controlado por Khronos group, algunos de sus miembros son:
  - AMD/ATI, Apple Inc., Ericsson, Google, Intel Corporation, Motorola, Nokia, Nvidia, Samsung Electronics, Sony Computer Entertainment, Oracle/Sun Microsystems, Epic Games (Unreal Engine/Gears of War)

# ¿Qué NO es OpenGL?

- Un sistema de ventanas.
- Un manejador de eventos.
- Un sistema de animación de objetos.
- Un sistema que controla la interacción entre objetos.
- Un sistema de base de datos de objetos tridimensionales.
- Etc.

# ¿Que provee OpenGL?

- Un conjunto de funciones que controlan la configuración del sistema de dibujado.
- Un sistema de proyección que permite especificar objetos en 3 dimensiones y llevarlos a coordenadas de pantalla.
- Un conjunto de funciones para realizar transformaciones geométricas que permiten posicionar los objetos en el espacio.
- ... y pocas cosas más!

# Otras librerías

- **GLU (OpenGL Utility Library)** es un conjunto de funciones que simplifican el uso de OpenGL para especificar la visual, construcciones simplificadas de superficies cuadráticas (entre otras cosas).
- **SDL: (Simple DirectMedia Library):** es una librería multimedia multiplataforma que ofrece funcionalidad para manejo de ventanas, lectura de teclado, reproducción de sonido, etc.

# Algo de Historia

- 1980s
  - Los fabricantes de hardware gráfico no seguían ningún estándar. Era el trabajo de los programadores dar soporte a cada pieza de hardware
- Fines de los 80s y principio de los 90s
  - Silicon Graphics (SGI) es el líder en gráficos 3D. Su librería IrisGL es considerada “state-of-the-art”
  - Ingresan al mercados nuevos proveedores de hardware 3D y se debilita la posición de SGI
  - Se decide convertir IrisGL en un estándar abierto

# Algo de Historia

- 1990
  - Comienza desarrollo de *OpenGL*
  - Comienza la colaboración *SGI – Microsoft*
- 1992
  - Se completa *OpenGL 1.0* (30 de junio)
  - Curso de *OpenGL* en *SIGGRAPH '92*
  - SGI lidera la creación del *OpenGL Architecture Review Board (OpenGL ARB)*, grupo de empresas que mantendrían y extenderían la especificación de *OpenGL*

# Algo de Historia

- 1995
  - Se completa *OpenGL* 1.1
    - Soporte de texturas en GPU
  - Microsoft lanza *Direct3D*, que se convertirá en el principal competidor de *OpenGL*
- 1996
  - Se hace pública la especificación de *OpenGL*
- 1997
  - ***Fahrenheit***: Acuerdo entre *SGI* y *Microsoft* para unir *OpenGL* a *Direct3D*. El proyecto aborta poco tiempo después

# Algo de Historia

- 1998
  - Se completa *OpenGL* 1.2
    - Texturas volumétricas (entre otras funcionalidades)
- 2000
  - *OpenGL* se hace accesible como código abierto
- 2001
  - Se completa *OpenGL* 1.3
    - Multi-texturas (entre otras funcionalidades)

# Algo de Historia

- 2002
  - Se completa *OpenGL 1.4*
    - Soporte para sombreado por hardware, generación automática de MipMap (entre otras funcionalidades)
- 2003
  - Se completa *OpenGL 1.5*
    - Vertex Buffer Objects: VBO (entre otras funcionalidades)
  - Microsoft abandona *OpenGL ARB*

# Algo de Historia

- 2004
  - Se completa *OpenGL 2.0*
    - Se introducen Pixel y Vertex Shaders
    - Definición del *OpenGL Shading Language (GLSL)*
- 2006
  - Se completa *OpenGL 2.1*
    - Soporte para Pixel Buffer Objects: PBO (entre otras funcionalidades)
  - *ARB* transfiere el control de *OpenGL* a *Khronos Group*.

# Algo de Historia

- 2007-2009
  - Se completa *OpenGL 3.0*
    - Mejora del lenguaje de shaders
    - *CUDA*: procesamiento masivo con procesadores de propósito general en la GPU (120+)
  - Surge *OpenGL ES*
    - Definición de un API común para la versión de dispositivos móviles
  - Surge *OpenCL*
    - Framework para escribir programas que se ejecutan en plataformas heterogéneas de CPUs, GPUs y otros procesadores

# Algo de Historia

- 2010
  - Se completa *OpenGL* 4.0
    - GLSL 4.0 (entre otras funcionalidades)
- 2011
  - Surge *WebGL* 1.0
    - API de gráficos 3D de bajo nivel para la web, accesible a través del elemento *Canvas* en *HTML5*
    - Basado en *OpenGL ES* 2.0
    - *Apple (Safari)*, *Google (Chrome)*, *Mozilla (Firefox)* y *Opera (Opera)* son miembros del *WebGL Working Group*
  - Se completa *OpenGL* 4.2

# Algo de Historia

- 2012
  - Se completa *OpenGL* 4.3
- 2014
  - Se completa OpenGL 4.4 (versión actual)
  - Varias especificaciones:
    - OpenGL ES 3.1, WebGL 1.0, OpenCL 2.0 (entre otras)

# Notas Generales

- La forma en la que se trabaja es dibujando una secuencia de imágenes estáticas. Cada una de dichos fotogramas las llamaremos “*frame*”, está compuesto por la información de color de cada pixel que conforma la imagen.
- La porción de memoria en donde se almacena el *frame* se llama “*colorBuffer*” (o “*frameBuffer*”).
- El contenido del *colorBuffer* es el que se muestra en pantalla.

# Notas Generales

- El funcionamiento general de los dispositivos gráficos de salida (monitor CRT/LCD, cañón, etc.) está basado en que la información gráfica se muestra como un barrido de arriba hacia abajo y de izquierda a derecha.
- El intervalo de tiempo entre que se dibuja el último pixel de la última línea y se pasa a dibujar el primer pixel de la primer línea se llama “*Vertical Blanking Interval*”.
  - La acción se suele llamar “*Vertical Retrace*”.

# Notas Generales

- En dispositivos antiguos con cantidad de memoria y poder de cómputo limitados, se tendía a calcular el siguiente *frame* durante el *Vertical Blanking Interval*.
- De esta manera se evitaba modificar el *colorBuffer* al mismo tiempo que se lo estaba desplegando en pantalla.

# Notas Generales

- Actualmente es común que las aplicaciones gráficas utilicen más de un *colorBuffer*. El más conocido es el “*DoubleBuffer*”.
- Mientras que un buffer se muestra en pantalla (*front buffer*) se dibuja sobre otro (*back buffer*).
- Cuando el *back buffer* está listo para ser mostrado, se hace un “*swap*” entre las referencias a donde comienza el *front* y el *back buffer* (sin hacer un intercambio físico de pixels).

# Notas Generales

- Cuando el *swap* de buffers no se hace sincronizado con el *vertical retrace*, lo que ocurre es que mientras se está mostrando un cierto *frame*, se pasa a mostrar el siguiente *frame*.
- El artefacto gráfico resultante se lo llama “*Tearing*”

# Notas Generales

- Ejemplo de “*Tearing*”



# Notas Generales

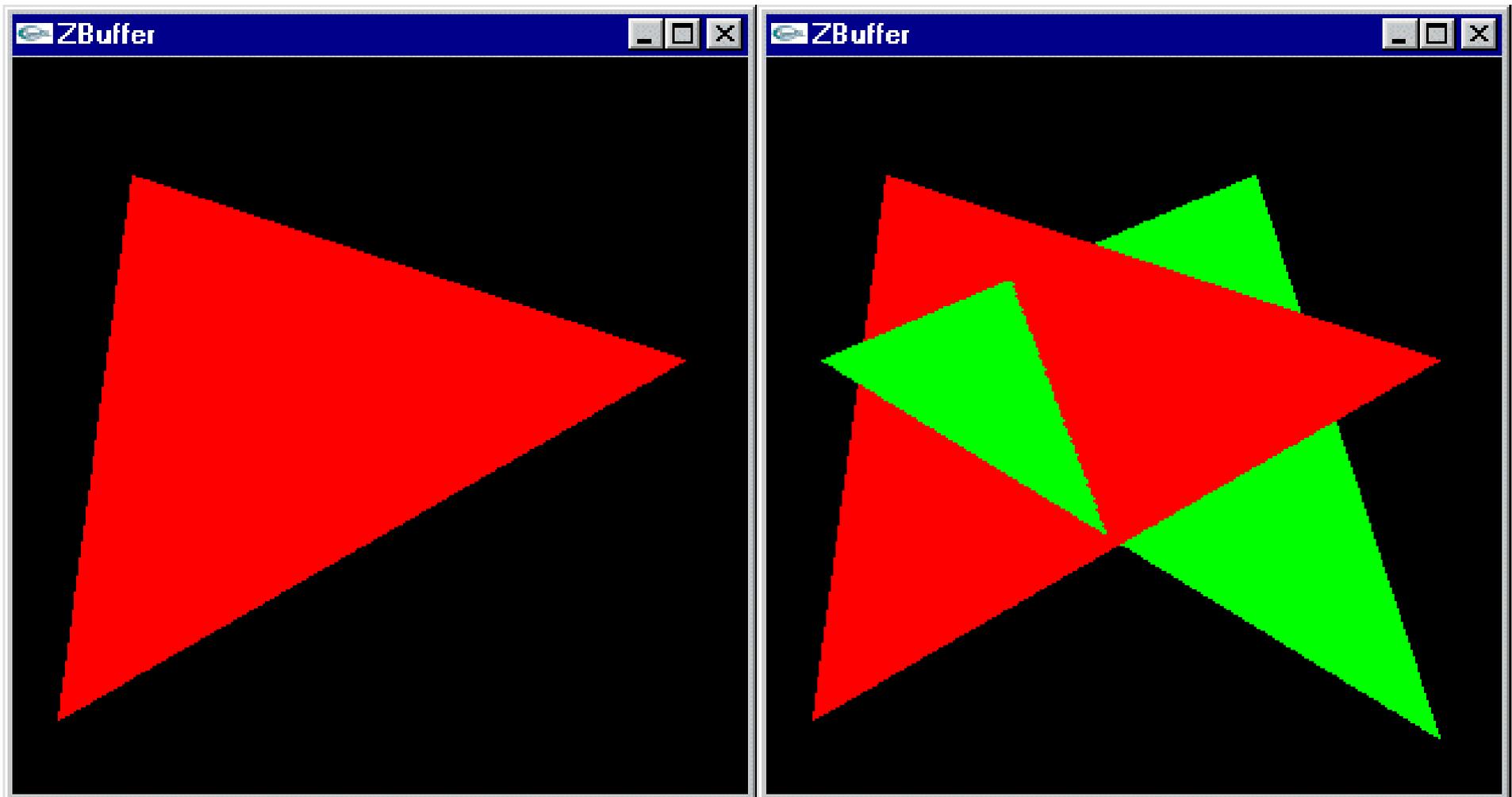
- La forma de trabajar con una librería gráfica es indicarle que dibuje una serie de primitivas gráficas. Cada primitiva (de forma individual) al dibujarse hace que se actualice el *color buffer*.
- No se puede garantizar un orden total entre primitivas con respecto a la profundidad, por lo que potencialmente no existe una secuencia en la que todas las primitivas se dibujen con su profundidad correcta.

# Notas Generales

- La alternativa es hacer un chequeo de profundidad a nivel de pixel. A ésto se lo conoce como *Z-buffer*
- Al intentar dibujar un nuevo pixel en el buffer de color, se chequea que la profundidad sea menor a la que se encuentra en el *Z-buffer*:
  - Si es menor se dibuja el pixel y se actualiza el z-buffer.
  - En caso contrario no se pinta el pixel.

# Notas Generales

- Ejemplo de uso de Z-Buffer



# SDL

- Presenta varios sub-sistemas que pueden ser inicializados de forma independiente.
  - En el ejemplo se presenta la inicialización del sub-sistema de video.

```
If( SDL_Init(SDL_INIT_VIDEO) == -1 )
{
    fprintf( stderr, "[Video Error]: %s\n", SDL_GetError() );
}
```

# SDL 1.2

- Para configurar el modo de video y resolución se debe utilizar la función ***SDL\_SetVideoMode***.
- Se puede configurar a SDL para dibujar con OpenGL en vez de usar las primitivas de SDL.

```
If ( SDL_SetVideoMode (640, 480, 32, SDL_OPENGL) == NULL )
{
    fprintf( stderr, "[Video Error]: %s\n", SDL_GetError() );
    SDL_Quit();
    Exit(1);
}
```

# SDL 2.0

- Si se está utilizando SDL 2.0 se utiliza la función ***SDL\_CreateWindow()*** seguido de ***SDL\_GL\_CreateContext()***.

```
SDL_Window* window = SDL_CreateWindow("Title",
                                       SDL_WINDOWPOS_CENTERED,
                                       SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL)
if(window == NULL){
    fprintf( stderr, "[Video Error]: %s\n", SDL_GetError() );
    SDL_Quit();
    exit(1);
}

SDL_GLContext glContext = SDL_GL_CreateContext(window);
if(window == NULL){
    fprintf(stderr, "[GL Context Error]: %s\n", SDL_GetError() );
    SDL_Quit();
    exit(1);
}
```

# SDL 2.0

- Luego de haber terminado la ejecución del programa, para liberar la memoria asociada al contexto de OpenGL se debe invocar `SDL_GL_DeleteContext()`.

```
SDL_GL_DeleteContext(glcontext);
```

- SDL 2.0 puede intercambiar entre modo ventana y fullscreen sin perder el contexto de OpenGL.

# SDL

- **SDL\_Quit()**: Libera todos los recursos usados por SDL.

**SDL\_PollEvent(&evento)**: se fija si ha sucedido algún evento y devuelve inmediatamente. En ***evento.type*** está el tipo de evento sucedido.

- **Uint8\* SDL\_GetKeyState(NULL)**: devuelve el estado completo del teclado en un array. Se pueden utilizar constantes de SDL para especificar las posiciones en dicho array.
  - Ej: ***SDLK\_a*** se corresponde con la tecla “a”

# SDL Swap Buffers

- La forma de indicar el intercambio entre el *front buffer* y el *back buffer* se implementa diferente según la versión del SDL
- **SDL 1.2:**
  - `SDL_GL_SwapBuffers()` ;
- **SDL 2.0:**
  - `SDL_GL_SwapWindow(window)` ;

# OpenGL: Notas generales

- El prefijo de las funciones indican la librería a la que pertenece.
  - Ej: `glColor3f`, `gluPerspective`, etc.
- El postfijo de las funciones indican el tipo de datos con los que trabaja
  - Ej: `glVertex3iv` recibe vértices definido con tres coordenadas de tipo *int* en formato vector.
  - Ej: `glTexCoord2f` recibe una coordenada de textura compuesta por dos valores de tipo *float*.

# OpenGL: Notas generales

- Las constantes se escriben en mayúsculas.
  - Ej: `SDL_VIDEO_INIT`, `GL_DEPH_TEST`, etc.
- Encabezados

```
#include "SDL.h"
#include "SDL_opengl.h"
```
- Al incluir el segundo encabezado se resuelven conflictos de nombres dependientes de la plataforma.

# OpenGL: Notas generales

- OpenGL es una máquina de estados!
  - El efecto de cada comando queda definido por el estado actual de dibujado.
  - Los estados son banderas que especifican que funcionalidades están habilitadas/deshabilitadas y cómo se deben aplicar.
  - Existen datos internos que son utilizados para determinar cómo un vértice debe ser transformado, iluminado, texturado, etc.

# Estructura del programa

- **Main:**

  - Abrir la ventana y configurar las librerías.

  - Inicializar estado de OpenGL

  - Loop principal

- **Loop principal:**

  - Chequear eventos y tomar decisiones

  - Actualizar el sistema según el tiempo que pasó

  - Redibujar

- **Redibujar:**

  - Limpiar los buffers (color, z-buffer, etc.)

  - Cambiar estado de dibujado y dibujar

# Loop principal

- Para dar la ilusión de movimiento (al igual que en el cine) se genera una secuencia de imágenes estáticas levemente diferente.
- Frame rate: cantidad de fotogramas (frames) que se presentan por segundo.
- Las distancias que los objetos se mueven entre un frame y el siguiente puede depender tiempo que ha transcurrido (o no).

# Loop principal

- Si el movimiento de los objetos es **dependiente** del frame rate, al cambiar de hardware los objetos van a demorar más (o menos) para ir desde un punto a otro en la escena.
- Si el movimiento es **independiente** del frame rate, al cambiar de hardware los objetos van a demorar lo mismo para ir de un punto a otro.
  - Lo que cambia es la cantidad de cuadros intermedios.

# Limpiar los buffers

- La acción de limpiar los buffers se refiere a inicializarlos con un valor definido.
- El valor utilizado para limpiar el buffer de color es el color de fondo.
- El valor utilizado para limpiar el z-buffer es la profundidad correspondiente a la distancia máxima visible. Cualquier primitiva que se encuentre más lejos no se va a dibujar.

# Dibujando primitivas

- Una forma en la que se dibujan primitivas utilizando OpenGL es indicando los datos que definen a cada vértice
  - Posición, color, coordenadas de textura, etc.
- Dado que OpenGL es una máquina de estados, tenemos que cambiar al estado correcto.

```
glBegin( primitiveType );  
//Definición de vértices  
glEnd();
```

# Tipos de primitivas

- GL\_POINTS
- GL\_LINES
- GL\_LINE\_STRIP
- GL\_LINE\_LOOP
- GL\_POLYGON
- **GL\_TRIANGLES\***
- GL\_TRIANGLE\_STRIP
- GL\_TRIANGLE\_FAN
- **GL\_QUADS\***
- GL\_QUAD\_STRIP

\* Se usan en los prácticos

# Transformaciones

- La transformación que se le aplica a un vértice antes de dibujarlo en pantalla queda definida por el estado de dos matrices:

```
glMatrixMode ( GL_PROJECTION );  
glMatrixMode ( GL_MODELVIEW );
```

- `GL_PROJECTION`: definición de las características de la cámara.
- `GL_MODELVIEW`: transformaciones sobre los objetos 3D de la escena.

# Transformaciones

- Es recomendable que la información de cada una de las matrices sea el adecuado al rol que cumple.
- Si la matriz `GL_PROJECTION` no almacena la información de la cámara, cada vez que se “limpia” la matriz `GL_MODELVIEW` se tiene que calcular de nuevo la matriz de transformación que describe la cámara.

# Transformaciones

- **glLoadIdentity ()** : carga la transformación identidad.
- **glTranslatef (x , y , z)** : traslación según el vector definido por (x,y,z)
- **glRotatef (angle , x , y , z)** : realiza una rotación de *angle* grados según el eje (x,y,z)
- **glScale (x , y , z)** : escala cada eje dependiendo del valor de (x,y,z)