Notas sobre OpenGL

Para profundizar el abordaje de los contenidos aquí expuestos, sugerimos consultar el Red Book (o Libro Rojo, llamado oficialmente "OpenGL Programming Guide: The Official Guide to Learning OpenGL") y el manual de referencia The Blue Book (o Libro Azul, llamado oficialmente "The OpenGL Reference Manual"), que se detallan al final del documento.

CONTENIDOS

1. Introducción a OpenGL

En este punto presentaremos las ideas básicas para realizar dibujos simples utilizando OpenGL. También veremos técnicas de programación que permitan dibujar diferentes formas geométricas y la utilización de librerías gráficas.

2. Abriendo una ventana gráfica para dibujar

El primer paso para realizar un dibujo es abrir una ventana en la pantalla. A continuación mostramos un primer ejemplo con el código para abrir una ventana. Las primeras cinco funciones del programa se encargan de abrir una ventana en la pantalla:

```
int main(int argc,char** argv) {
  // Inicializa la librería GLUT
  glutInit(&argc,argv);
  // Selecciona modo de display: RGB y simple buffer
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  // Fija tamaño y posición inicial de las ventanas
  glutInitWindowSize(480,360);
  glutInitWindowPosition(50,50);
  // Crea la ventana
  glutCreateWindow ("Ejemplo 1");
  glutMainLoop(); // entra en loop de reconocimiento
de eventos
  return 0;
}
```

Luego, se realiza una breve descripción de las funciones utilizadas para la inicialización de GLUT.

- *glutInit(&argc,argv):* esta función inicializa la librería GLUT, pudiéndosele pasar algún parámetro por línea de comandos a través de *argc* y *argv*.

- *glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE):* esta función especifica cómo se inicializa la ventana. Está indicando que se va a utilizar un buffer simple y que los colores se van a especificar mediante la mezcla del rojo(Red), verde(Green) y azul(Blue). El doble buffer se utiliza para animaciones.

- *glutInitWindowSize(480,360):* especifica que la ventana se debe iniciar con un tamaño de 480x360 píxeles de ancho por alto. Una vez que el programa está corriendo, el usuario puede modificar el tamaño si lo desea.

- *glutInitWindowPosition (50,50):* con esta función se fija la posición inicial de las ventanas que se creen.

- *glutCreateWindow (Mi primer dibujo):* esta función crea una ventana principal colocando el título indicado en la barra superior de la ventana.

La inicialización de GLUT provee memoria y recursos. Por razones de eficiencia, se solicita solamente lo que se va a utilizar. Por eso, en el ejemplo anterior no se requiere el canal Alpha para transparencias, ya que se usarán figuras totalmente opacas.

El mismo criterio se aplica al solicitar simple buffer, debido a que no habrá animaciones en el ejemplo. Por el contrario, siempre que se pretenda trabajar en tiempo real, se exige doble buffer para evitar el parpadeo. En los próximos ejemplos explicaremos más en detalle estas cuestiones.

Otro aspecto importante de este tema es el origen de coordenadas. En una ventana, para GLUT y para el sistema operativo (cursor, posición de la ventana), el origen de coordenadas (0,0) se encuentra en la esquina superior izquierda. Esto es inconsistente con OpenGL, cuyo origen está ubicado en la esquina inferior izquierda. Esto, como

veremos más adelante, no supone ningún problema. La ventana generada es de 480x360 píxeles. La coordenada *x* varía entre 0 y 479, y la coordenada *y*, entre 0 y 359.

Para una mayor interiorización en las funciones de GLUT, sugerimos consultar el apéndice "Notas sobre GLUT" (Notas Unidad 1 - Manipulación de objetos en 2D.pdf), que se explicó anteriormente en el presente curso, o la especificación oficial de GLUT que se detalla al final del documento ("The OpenGL Utiliy Toolkit (GLUT) Programming Interface").

3. Dibujo de primitivas gráficas

OpenGL posee herramientas que permiten dibujar primitivas, tales como líneas, polilíneas y polígonos, que están definidos por vértices. Para dibujar los objetos hay que ingresar una lista de vértices entre dos funciones llamadas *glBegin()* y *glEnd()*. El argumento de *glBegin()* determina qué objeto se dibuja. Por ejemplo, en la siguiente imagen se renderizan dos puntos en una ventana previamente creada:

```
glBegin(GL_POINTS);
  glVertex2i(100, 50);
  glVertex2i(100, 130);
glEnd();
```

Muchas funciones en OpenGL, como *glVertex2i()*, tienen distintas alternativas, que permiten distinguir el número y el tipo de argumento pasado a la función. En la siguiente figura se muestra el formato de las funciones de OpenGL:

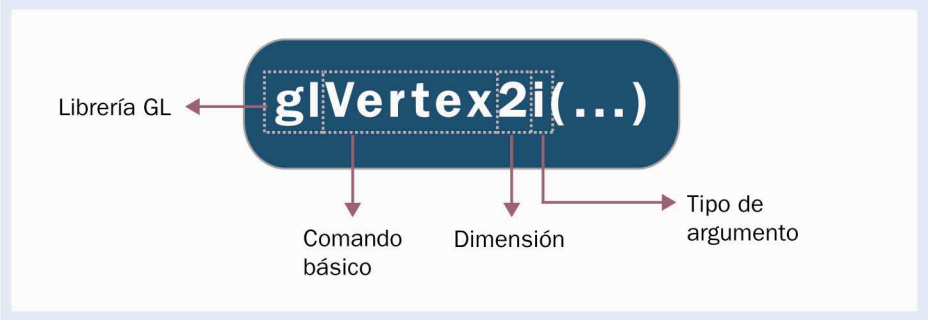

Figura 1: Formato de las funciones de OpenGL.

El prefijo gl indica que se trata de una función de la librería OpenGL. Luego figura el comando básico, seguido por el número de argumentos que se envían a la función, y finalmente, aparece el tipo de argumentos (*i* para enteros, *f* para float, etc.).

A continuación, se muestra el ejemplo anterior pasando valores reales:

```
glBegin(GL_POINTS);
  glVertex2f(100.0f, 50.0f);
  glVertex2f(100.0f, 130.0f);
glEnd();
```

4. Tipo de datos de OpenGL

OpenGL trabaja internamente con tipos de datos especificados. Por ejemplo, en funciones como *glVertex2i(),* se espera un entero de un tamaño de 32 bits. En la siguiente tabla, se muestran los sufijos de los comandos y el tipo de argumento de los datos.

Tipo de datos OpenGL	Representación interna	Definición como tipo C	Sufijo literal en C
GLbyte	Entero de 8 bits	Char con signo	b
GLshort	Entero de 16 bits	Short	s
GLint, GLsizei	Entero de 32 bits	Long	i
GLfloat, GLclampf	Coma flotante 32 bits	Float	f
GLdouble, GLclampd	Coma flotante 64 bits	Double	d
GLubyte, GLboolean	Entero de 8 bits sin signo	Unsigned char	ub
GLushort	Entero de 16 bits sin signo	Unsigned short	us
GLuint, GLenum,GLbitfield	Entero de 32 bits sin signo	Unsigned int or long	ui

Figura 2: Tipo de variables de OpenGL.

5. Estado de OpenGL

OpenGL almacena muchas variables, como el tamaño del punto, el color del dibujo y fondo de la pantalla, entre otros, y funciona como una máquina de estados. Es decir, si a una propiedad se le asigna un valor determinado, todo lo que se haga a partir de ese momento se verá afectado por ese valor, hasta que el mismo se modifique o desactive de forma explícita.

El tamaño del punto se especifica mediante *glPointSize(),* que recibe como argumento un tipo de datos *float.* Si el argumento es 3.0, el punto normalmente se representa como un cuadrado de tres píxeles de lado.

En OpenGL, un color sencillo se representa como una mezcla de componentes roja, verde y azul. El color de las entidades a dibujar se puede especificar mediante *glColor3f(R,G,B).* Dependiendo de la configuración, el rango de cada componente puede variar de 0.0 a 1.0, y los valores para cada una pueden ser cualquier valor de coma flotante válido entre 0 y 1.

En la siguiente figura se muestra alguno de los colores normales con sus valores por componente:

Color compuesto	*Componente roja*	*Componente verde*	*Componente azul*
Negro	0.0	0.0	0.0
Rojo	1.0	0.0	0.0
Verde	0.0	1.0	0.0
Amarillo	1.0	1.0	0.0
Azul	0.0	0.0	1.0
Magenta	1.0	0.0	1.0
Cian	0.0	1.0	1.0
Gris oscuro	0.25	0.25	0.25
Gris claro	0.75	0.75	0.75
Blanco	1.0	1.0	1.0

Figura 3: Algunos colores comunes por componentes.

El color de fondo se determina con *glClearColor (R, G, B, A),* donde A (Alpha) especifica el grado de opacidad de un objeto (por defecto se utiliza valor 1.0). Para limpiar la ventana y usar el color de fondo se debe utilizar *glClear(GL_COLOR_BUFFER_BIT).* El argumento *(GL_COLOR_BUFFER_BIT)* es una constante propia de OpenGL.

6. Programa completo de OpenGL

A continuación, veremos el segundo ejemplo con el código completo de un programa simple, que representa dos puntos.

En la iniciación, *myInit(),* se selecciona el color de fondo y de dibujo, el tamaño de punto y el sistema de coordenadas. En la siguiente unidad se explicarán en detalle los conceptos vinculados al sistema de coordenadas (funciones *glMatrixMode, glLoadIdentity, glOrtho y viewport*).

El dibujo se encapsula en el *callback de display* o *dibujo Display_cb()* (consistente en suministrar la dirección de la función que deberá llamar cuando se produzca determinado evento) y allí se puede ver cómo se arma la pantalla que se visualizará. Dado que todo lo dibujamos con OpenGL, la mayoría de los comandos de dibujo de OpenGL estará allí o en rutinas que se llaman desde allí. El callback de display debe estar definido obligatoriamente.

Luego, por medio de *glFlush,* se obliga a OpenGL a pintar lo que está en la lista de primitivas a dibujar. Este programa de ejemplo no es interactivo, por lo que se utiliza el modo simple buffer *(GLUT_SINGLE)* en combinación con *glFlush.*

Más adelante, en programas interactivos, veremos el uso de doble buffer de la mano de *glutSwapBuffers.*

```
#include <GL/glut.h>
using namespace std;
void myInit(void){
  // Selecciona el color de fondo
  glClearColor(0.85f,0.9f,0.95f,1.f);
  // El color a dibujar
  glColor3f(1.0f, 0.0f, 0.0f);
  // Tamaño de los puntos
  glPointSize(5.0);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, 480.0, 0.0, 360.0,-1,1);
}

void Display_cb() {
  // Limpia la pantalla
  glClear(GL_COLOR_BUFFER_BIT);
  // Dibuja los dos puntos
  glBegin(GL_POINTS);
    glVertex2i(100, 50);
    glVertex2i(100, 130);
  glEnd();
  // Envia toda la salida al monitor
  glFlush();
}

int main(int argc,char** argv) {
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  glutInitWindowSize(480,360);
  glutInitWindowPosition(50,50);
  glutCreateWindow ("Ejemplo 2");
  // Declara el callback de display
  glutDisplayFunc(Display_cb);
  myInit();
  glutMainLoop();
  return 0;
}
```

7. Dibujo de líneas con OpenGL

Con OpenGL es muy sencillo dibujar líneas empleando *GL_LINES* como argumento en la función *glBegin()* y pasando los dos puntos correspondientes a las vértices del segmento a representar. Por ejemplo, para dibujar una línea entre los puntos de coordenadas (50, 100) y (70, 155), el código sería el siguiente:

```
glBegin(GL_LINES);  //  Se  pasa  como  argumento  la
constante GL_LINES
  glVertex2i(50, 100);
  glVertex2i(70, 155);
glEnd();
```

Si se especifican más de dos vértices entre *glBegin(GL_LINES)* y *glEnd(),* éstos se toman de dos en dos y se dibujan líneas separadas cada dos puntos.
El siguiente ejemplo representa una línea horizontal y otra vertical:

```
glBegin(GL_LINES);
  glVertex2i(30, 40);
  glVertex2i(100, 40);
  glVertex2i(20, 10);
  glVertex2i(20, 150);
  //aqui se pueden añadir más puntos si se desea
glEnd();
```

El color de la línea se designa de la misma manera que los puntos, usando *glColor3f().*

El ancho se configura utilizando *glLineWidth(GLfloat ancho).* Esta función toma un solo parámetro que especifica el ancho aproximado, en píxeles, de la línea dibujada. Por defecto, el valor de ancho es 1.0. Al igual que el grosor de puntos, todos los anchos de línea no están soportados y hay que verificar que el ancho de línea que se desea está disponible.

A continuación, se muestra el código que hay que utilizar para averiguar el rango del ancho de línea y el paso mínimo.

```
// Almacena el rango de ancho de linea soportado
GLfloat size[2];
//  Almacena  los  incrementos  en  el  ancho  de  linea
soportado
GLfloat paso;
// Obtiene el rango y paso de los grosores de líneas
soportados
glGetFloatv(GL_LINE_WIDTH_RANGE, size);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &paso);
```

Aquí, el vector *size* contiene dos elementos con el valor más pequeño y el más grande disponibles entre ancho de línea. La implementación de Microsoft permite un ancho de línea desde 0.5 a 10.0, con el paso mínimo de 0.125.

8. Patrones de línea

Además de cambiar el ancho de línea, se pueden crear líneas con un patrón punteado o rayado.

Para utilizar patrones de línea, lo primero que hay que hacer es habilitar el patrón con la función *glEnable(GL_LINE_STIPPLE).*

Por otro lado, es importante mencionar que cualquier función que se activa con *glEnable()* puede desactivarse con *glDisable().*

Luego, con la función *glLineStipple* se establece el patrón que usarán las líneas para dibujar.

```
void glLineStipple(Glint factor, Glushort pattern);
```

El parámetro *pattern* es un valor de 16 bits que especifica el patrón que se deberá usar cuando se dibuje la línea. Cada bit representa una sección del segmento de línea que puede o no estar activo.

El parámetro *factor* especifica un multiplicador que determina a cuántos píxeles afectará cada bit en el parámetro *pattern.* Por ejemplo, seleccionando el factor 5, se logra que cada bit en el patrón represente cinco píxeles en una fila.

En la siguiente tabla se muestran algunos ejemplos de patrones. El patrón se debe ingresar en formato hexadecimal. Por ejemplo, para el patrón hexadecimal 0x3333, el resultado es 11001100110011 (este número binario se obtiene al ingresar a la calculadora científica el valor 11001100110011 como opción binario y convirtiéndolo a hexadecimal):

Patrón	Factor	resultado
0xFF00	1	
0xFF00	2	
0x5555	1	
0x3333	1	

Figura 4: Ejemplos de patrones simples.

A continuación, veremos el tercer ejemplo con la implementación de patrones de línea.

```
void myInit(void){
  // Selecciona el color de fondo
  glClearColor(0.85f,0.9f,0.95f,1.f);
  // El color a dibujar
  glColor3f(1.0f, 0.0f, 0.0f);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, 480.0, 0.0, 360.0,-1,1);
}

void Display_cb() {
  // Limpia la pantalla
  glClear(GL_COLOR_BUFFER_BIT);
  glEnable(GL_LINE_STIPPLE);
  GLint factor=2;
  GLushort pattern=0x3333;
  glLineStipple(factor,pattern);
  // Dibuja las dos lineas
  glBegin(GL_LINES);
    glVertex2i(30, 40);
    glVertex2i(100, 40);
  glEnd();
  glDisable(GL_LINE_STIPPLE);
```

```
  glBegin(GL_LINES);
    glVertex2i(20, 10);
    glVertex2i(20, 150);
  glEnd();
  // Envia toda la salida al monitor
  glFlush();
}

int main(int argc,char** argv) {
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
  glutInitWindowSize(480,360);
  glutInitWindowPosition(50,50);
  glutCreateWindow ("Ejemplo 3");
  // Declara el callback de display
  glutDisplayFunc(Display_cb);
  myInit();
  glutMainLoop();
  return 0;
}
```

Como se puede observar, para renderizar la segunda línea de manera estándar se deshabilitó la utilización de patrones.

Este programa no es interactivo; por lo tanto, se utiliza simple buffer *(GLUT_SINGLE).*

9. Dibujo de polilíneas y polígonos

Por *polilínea* se entiende una colección de segmentos donde el último de uno es el primero del siguiente. Se suelen escribir como una lista ordenada de puntos, tal como se observa en la siguiente ecuación:

$$p_0 = (x_0, y_o), p_1 = (x_1, y_1), \ldots p_n = (x_n, y_n),$$

En OpenGL, una polilínea se dibuja especificando sus vértices, que se encuentran entre *glBegin(GL_LINE_STRIP)* y *glEnd(),* como se muestra en el código del siguiente ejemplo:

```
glBegin(GL_LINE_STRIP); // Dibuja una polilínea abierta
  glVertex2i(20,10);
  glVertex2i(50,10);
  glVertex2i(20,80);
  glVertex2i(50,80);
glEnd();
```

El resultado aparece claramente en la figura. Atributos como color, espesor y patrón se utilizan de igual modo que en el caso de las líneas simples, como vimos anteriormente.

Si se desea conectar el último punto con el primero para obtener una polilínea cerrada, sólo se remplaza *(GL_LINE_STRIP)* por *(GL_LINE_LOOP).* El resultado también se muestra en la siguiente figura:

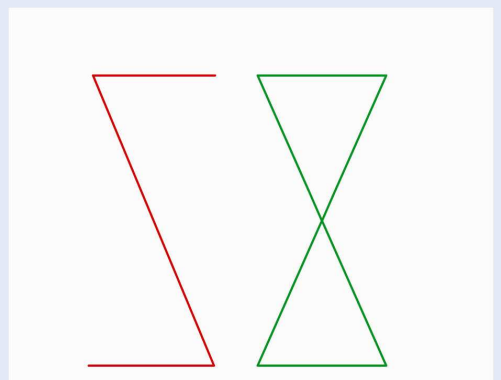

Figura 5: Polilínea abierta y cerrada.

Los polígonos dibujados usando *(GL_LINE_LOOP)* no se pueden rellenar con color. Para hacerlo, se debe emplear *glBegin(GL_ POLYGON),* pero con la precaución de que el polígono sea convexo.

10. Otras primitivas gráficas

Como se observa en la figura, es posible dibujar otras primitivas gráficas.

Para ello hay que alternar el argumento en *gBegin()* de la siguiente manera:

- *GL_TRIANGLES:* toma la lista de vértices de tres en tres, dibujando triángulos independientes.
- *GL_QUADS: toma cuatro vértices y dibuja cuadriláteros independientes.*
- *GL_TRIANGLE_STRIP:* dibuja una serie de triángulos unidos. Cada nuevo vértice forma un triángulo con el par anterior.
- *GL_TRIANGLE_FAN:* dibuja una serie de triángulos conectados, con un único vértice común a todos.
- *GL_QUAD_STRIP:* dibuja una serie de cuadriláteros unidos. Cada par de vértices sucesivos forman un cuadrilátero con el par anterior.

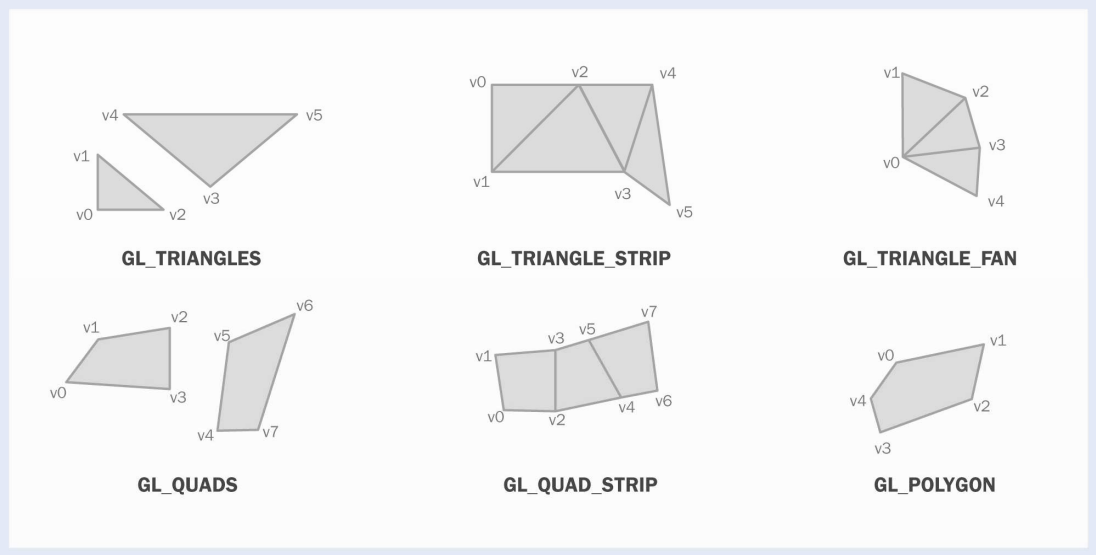

Figura 6: Otras primitivas geométricas.

11. Métricas en espacios bidimensionales

Se pueden utilizar varias métricas para expresar la distancia entre dos puntos del plano. Este concepto se utilizará más adelante para verificar si el usuario del programa pica cerca o lejos de un punto existente.

Una métrica muy divulgada es la *distancia euclidiana* o *euclídea,* que es la distancia entre dos puntos del espacio euclídeo bidimensional, equivalente a la longitud del segmento de recta que los une (entendiendo al espacio euclídeo bidimensional como -por ejemplo- una hoja de papel; es decir, el plano 2D *xy* que manejamos usualmente).

La distancia euclídea se deduce a partir del teorema de Pitágoras.[1] Es decir, en un espacio bidimensional, la distancia euclídea entre dos puntos P1 y P2,

de coordenadas (x1, y1) y (x2, y2) respectivamente, es la siguiente:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

La zona que queda definida por todos los puntos del plano que están a distancia *d* de P_1 es una circunferencia centrada en P_1 de radio *d* y que pasa por P_2.

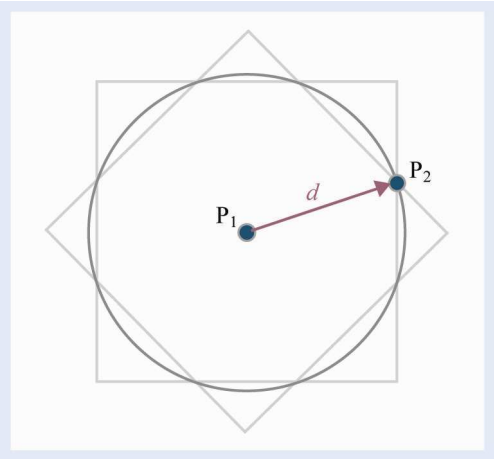

Figura 7: Zonas definidas por las distintas métricas.

En términos de eficiencia del programa, el cálculo de la distancia euclídea involucra una raíz cuadrada, que es una operación bastante costosa computacionalmente.

Otra manera de expresar la distancia entre dos puntos del plano es por medio de la denominada *métrica del rombo.* Dados dos puntos P_1 y P_2, se define como sigue:

$$d(P_1, P_2) = |x_2 - x_1| + |y_2 - y_1|$$

Donde | | definen el módulo o valor absoluto de un número.[2]

Se denomina métrica del rombo debido a que los puntos equidistantes de P1, con esta métrica, definen un rombo (en realidad, un cuadrado a 45°).

[1] El Teorema de Pitágoras establece que en un triángulo rectángulo, el cuadrado de la longitud de la hipotenusa (el lado de mayor longitud del triángulo rectángulo) es igual a la suma de los cuadrados de las longitudes de los dos catetos (los dos lados menores del triángulo, los que conforman el ángulo recto).

[2] El valor absoluto o módulo de un número real es su valor numérico sin tener en cuenta su signo, sea éste positivo (+) o negativo (-). Así, por ejemplo, 5 es el valor absoluto de 5 y de -5.

Analizando la eficiencia del programa, aquí solamente están incluidas operaciones de suma y resta, que son operaciones poco costosas computacionalmente.

Otra métrica eficiente es la siguiente:

$$d(P_1, P_2) = \text{Max } (x_2 - x_1 \ , \ y_2 - y_1)$$

Los puntos equidistantes de P_1 definen un cuadrado alineado con los ejes cartesianos alrededor de P_1.

12. Procesamiento de eventos

Las aplicaciones interactivas permiten al usuario controlar el flujo del programa de una forma natural, por ejemplo, posicionando y pulsando el mouse o mediante el uso del teclado. Dentro del programa se puede capturar la posición del mouse e identificar una tecla pulsada.

GLUT tiene incorporado su propio ciclo de proceso de eventos.[3] Para ello es necesario llamarlo con la función *glutMainLoop,* que implementa un ciclo infinito que se encarga de consultar la cola de eventos y responder a cada uno de ellos ejecutando una función. La función que se ejecuta cuando se produce un evento debe ser escrita por el programador y, además, ser registrada como una función de respuesta (denominada *callback*). De este modo, GLUT identifica a qué función llamar para responder a un determinado evento.

Existen tres tipos de eventos: de ventana, de menús y globales.

Los eventos de ventana están asociados a las ventanas y tratan cuestiones como el redibujado, el redimensionamiento y la opresión de un botón del mouse.

Los eventos de menú responden a la selección de las opciones de un menú.

Los eventos globales están asociados al control de temporizadores y acciones *idle,* que veremos en otra unidad.

Los callbacks que se declaran corresponden a los eventos que deseamos que el programa interprete. No tiene sentido tornar más lenta a la computadora por atender eventos que no se van a procesar (movimiento del mouse, por ejemplo). Con este mismo sentido, debemos cuidar la velocidad dentro de los callbacks: si no hay nada que hacer en él, hay que salir muy rápido.

Para realizar el registro de un callback, se utiliza un conjunto de funciones con sintaxis común: *void glut[Event]Func (tipo (*function) (parámetros)),* que indican que para responder al evento *Event* debe utilizarse la función *function.*

Es posible registrar la llamada a funciones de cada uno de estos eventos mediante los siguientes comandos:

- *glutMouseFunc(Mouse_cb)* registra en *Mouse_cb()* el evento que ocurre cuando se presiona un botón del mouse.

- *glutKeyboardFunc(Keyboard_cb)* registra en *Keyboard_cb()* el evento que ocurre cuando se presiona una tecla.

Luego, para interactuar con el mouse, hay que definir la función *Mouse_cb()* que toma cuatro parámetros:

[3] Un evento es un suceso que ocurre en un sistema, por ejemplo, realizar un click, presionar una tecla o minimizar una ventana, entre otros.

- *void Mouse_cb(int button, int state, int x, int y)*

Entonces, cuando se presiona el botón del mouse, el sistema llama a la función registrada con los valores de estos parámetros.

El valor de *button* podrá ser *GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON* y *GLUT_RIGHT_BUTTON,* correspondiéndole el botón izquierdo, central o derecho, respectivamente.

El valor de *state* será *GLUT_UP* y *GL_DOWN.* El valor de *x* e *y* indica la posición del mouse en el momento del evento. El valor numérico de *x* corresponde a la distancia en píxeles a partir de la izquierda de la ventana, pero el valor de *y* es la distancia en píxeles de la parte superior de la ventana.

Para interactuar con el teclado hay que definir la función *Keyboard_cb().* El prototipo de la función es el siguiente.

```
void Keyboard_cb(unsigned char key,int x,int y)
```

Donde *key* es el caracter *ASCII.* Los valores *x* e *y* devuelven el valor de la posición del mouse al momento de presionar la tecla.

Como mencionamos anteriormente, para acceder a mayores detalles sobre estas funciones de GLUT, sugerimos consultar el apéndice “Notas sobre GLUT” (Notas Unidad 1 - Manipulación de objetos en 2D.pdf), que se explicó anteriormente en el presente curso, o la especificación oficial de GLUT que se detalla al final del documento (“The OpenGL Utiliy Toolkit (GLUT) Programming Interface”).

13. Implementando un programa interactivo.

En este punto haremos un análisis del ejemplo 4, que se puede descargar desde la plataforma web.

Este programa permite construir un triángulo a partir de los puntos seleccionados por el usuario. Además, permite desplazar y eliminar los puntos creados.

El código fuente se empieza a leer desde la función *main().* Allí puede verse que el programa consiste en una etapa de inicialización, que luego entra en el loop de ejecución, del cual nunca sale.

En cuanto a la inicialización, toda la labor se realiza en una rutina especial llamada *inicializa()* (excepto la inicialización interna de GLUT, para no pasarle los argumentos requeridos).

La inicialización de GLUT provee memoria y recursos. Es importante tener en cuenta que, por cuestiones de eficiencia, se debe solicitar solamente lo que se va a utilizar. En efecto, en el ejemplo vinculado al buffer de color, no se solicita el canal Alpha porque se dibujarán figuras totalmente opacas (sólo si se usaran transparencias o alguna forma de mezcla de colores, se necesitaría el canal Alpha, a través de *GLUT_RGBA*).

El mismo criterio de eficiencia se tiene en cuanta al momento de decidir el uso de simple buffer o doble buffer.

Siempre que se pretenda trabajar en tiempo real, se requiere doble buffer. En este caso, mientras se percibe el buffer *de adelante* (front buffer), todo el dibujo se realiza en el buffer *de atrás* (back buffer) y, cuando está concluido el proceso de carga del buffer, se intercambian los buffers con *glutSwapBuffers;* el de atrás pasa al frente y se actualiza la pantalla (*glutSwapBuffers* ejecuta implícitamente a *glFlush*). Con la técnica de doble buffer se evita el parpadeo (flickering). El *flickering* se produce en las animaciones que son efectuadas con un solo buffer, debido a que se está dibujando sobre lo que se está visualizando.

En unidades posteriores explicaremos en detalle los conceptos vinculados a buffers.

Por el momento comenzaremos examinando el callback de display o dibujo.

En *Display_cb()* podemos ver cómo se arma la pantalla que se va a visualizar. Debido a que todo lo dibujamos con OpenGL, la mayoría de los comandos de dibujo de OpenGL están allí o en rutinas que son llamadas desde allí.

El redisplay no es implícito: cuando el programa modifica (edita) algo, debe llamar explícitamente al callback (o avisar la necesidad con *glutPostRedisplay).* Pero el sistema operativo también provoca llamadas implícitas, por ejemplo, al mover una ventana de otro programa por sobre la nuestra.

Descomentando la primera línea podemos ver, en la ventana de comandos, la extraordinaria cantidad de veces que se llama a esta rutina. Por lo tanto, se produce evidentemente un cuello de botella en el que hay que programar con mucho cuidado para no perder eficiencia.

El redisplay explícito (que se logra llamando a Display_cb) se produce al terminar con una serie de modificaciones. Si en una rutina se desconoce si lo que acaba de modificarse es todo lo que hay que hacer (o se sabe que no es todo), en lugar de realizar la llamada explícita, obligando a redibujar, se avisa, mediante una llamada a *glutPostRedisplay(),* que hay que hacerlo en el próximo paso por el loop de ejecución. Luego, aparecen los comandos para dibujar primitivas, que en este ejemplo son los puntos y el triángulo.

Reshape_cb() se encarga de reaccionar a los cambios de tamaño de la ventana y también es llamado cuando la misma es creada. En el *reshape_cb()* se definen el *viewport,* que es la región de la ventana donde se hará el dibujo; el sistema de coordenadas y la porción del espacio visible. En este caso, las coordenadas miden simplemente píxeles en la ventana completa.

La dificultad que esto plantea es la falta de uniformidad en el origen: para los callbacks de GLUT y para el sistema operativo (cursor, posición de la ventana), las coordenadas son siempre píxeles, con el origen ubicado arriba y a la izquierda. Para OpenGL, en cambio, el origen está abajo y a la izquierda y con *z* apuntando hacia fuera del monitor. Para solucionarlo, reemplazamos la variable y por h-y, en los callbacks asociados a las coordenadas del cursor.

Los detalles de las funciones de OpenGL que se usan en el callback de reshape los veremos en unidades posteriores, al analizar las transformaciones y sus matrices.

La definición del sistema de coordenadas y el viewport se hacen donde haga falta, normalmente en una rutina especial. Sin embargo, en este pequeño programa, se hicieron en el callback de reshape porque es en el único momento en que cambian.

Para salir del programa se utiliza el estándar <ALT>+<F4>, por lo que se declaró y definió el callback que atiende a las teclas especiales: Special_cb().

Como puede verse, sólo se sale del programa si la tecla presionada es <F4> y si, además, el modificador <ALT> está pulsado. Los modificadores son <ALT>, <CTRL> o <SHIFT> o combinaciones. Los detalles pueden verse en el manual de GLUT.

A partir de aquí, para continuar con el análisis del código fuente, seguimos cursos de acción y no callbacks.

Cuando el programa comienza no hay ningún punto definido; picando en la ventana se crean los puntos y cuando hay tres se visualiza el triángulo. Si se pica cerca de un punto previo, éste puede moverse al igual que el punto recién creado. Las teclas DEL y Backspace se programaron para borrar puntos. Como es estándar, una borra *hacia adelante* y la otra *hacia atrás* en la lista de puntos.

Las variables que almacenan los puntos y controlan la edición son globales y están definidas y explicadas al principio del archivo fuente. Sólo resta decir que, por simplicidad, los puntos se almacenan en una lista unidimensional de pares de enteros {x0,y0,x1,y1,x2,y2,x3,y3}.

Con el funcionamiento del programa en mente, se puede seguir con facilidad la acción de cada evento; sólo hay que saber qué callback maneja cada uno.

Así, Mouse_cb() se encarga de los clicks, reacciona al pulsar y al soltar algún botón del mouse(en este caso, sólo atendemos al botón izquierdo).

Motion_cb() se activa cuando se mueve el cursor con algún botón apretado; esta acción se llama *drag* (arrastre). Sólo se define como callback cuando efectivamente se pulsó algún botón y se anula (se le pasa a GLUT la dirección nula) al soltarlo. El callback de los drags sigue al cursor aún fuera de la ventana, por lo que hay que programar si se permite o no el arrastre de un punto fuera de la ventana. Puede suceder que quede invisible y, por lo tanto, no se podrá volver a picar sobre el mismo para editarlo.

Por último, Keyboard_cb() atiende a las pulsaciones del teclado estándar (ASCII); en este caso, sólo escucha DEL y Backspace.

Existen dos partes ligeramente complicadas: una consiste en averiguar si el botón izquierdo se pulsó con el cursor cerca de un punto previo y la otra es la lógica de borrar hacia delante o hacia atrás.

Esperamos que pueda entender el ejemplo interactuando con el programa, utilizando breakpoints y watches, o bien empleando algunos cout a la ventana de comando.

BIBLIOGRAFÍA

Hearn, D.; Baker, P. *Computer Graphics, C version.* Second Edition. Pearson Prentice Hall, 1997.

Kilgard, M. J. "The OpenGL Utiliy Toolkit (GLUT) Programming Interface". Silicon Graphics, Inc, 1996 [en línea] [OpenGL]
http://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf

Neider J.;Davis,T.; Woo M. "OpenGL Programming Guide: The Official Guide to Learning OpenGL". Addison-Wesley Publishing Company, 1997 [en línea] [OpenGL]
http://www.opengl.org/documentation/red_book/

OpenGL [en línea] http://www.opengl.org

The Blue Book: The OpenGL Reference manual [en línea] [OpenGL]
http://www.opengl.org/documentation/blue_book/