Introducción a la Práctica con OpenGL y GLUT

1. Herramientas a Utilizar

Las prácticas de esta materia consistirán en analizar, modificar, completar o desarrollar programas gráficos en C o C++, utilizando *OpenGL*. En general, para cada práctico, la cátedra proveerá un código base sobre el cual trabajar, para centrar el trabajo solo en los puntos que interesan específicamente para el tema de estudio del momento. Estos ejemplos fueron desarrollados con las versiones 1.x de *OpenGL* y la biblioteca *FreeGLUT*. Por esto, para poder llevar a cabo la práctica el alumno deberá contar con un entorno de desarrollo para C++ y con estas bibliotecas. Cualquier compilador/IDE puede ser configurado para utilizar estas bibliotecas, y las mismas están disponible para casi cualquier sistema operativo.

OpenGL es la biblioteca que se encarga de generar los gráficos. Define la API mediante la cual enviamos información y comandos a la GPU. Existen varias versiones de *OpenGL*, y hay marcadas diferencias entre ellas. Las versiones 3.x y 4.x son las utilizadas generalmente por la industria (siendo la 4.6 es la más actual al momento de escribir este apunte, que data de 2017). Sin embargo, para la práctica utilizaremos las versiones 1.x[1]. Se selecciona esta versión, a pesar de ser obsoleta, porque es la más simple y fácil de aprender y requiere muy poco trabajo para su puesta en marcha. En el contexto de la materia, la API gráfica no es lo importante, sino simplemente una herramienta con la cual probar los conceptos aprendidos en teoría, y experimentar. En ese sentido, *OpenGL* 1.3 tiene funcionalidades suficientes, y presenta una interfaz mucho más simple que alternativas como versiones modernas de *OpenGL*, y otras APIs como *DirectX* o *Vulkan*. Una vez finalizada la materia, habiendo ganado experiencia y comprendido los fundamentos, el alumno no debería tener mayores inconvenientes para aprender a utilizar estas otras APIs o versiones por su cuenta.

Por otro lado, complementaremos *OpenGL* con *GLUT* o alguna de sus variantes (en particular *FreeGLUT* es la implementación más usada y recomendada). *OpenGL* se encarga de generar la imagen que se mostrará en pantalla, pero de nada más. Es necesario crear un ”contexto” primero. Esto es, una ventana y un área dentro de la misma donde *OpenGL* pueda ”dibujar”. Además, para la interacción, también es necesario detectar los eventos sobre la ventana (si cambia de tamaño, si el usuario hace click o presiona una tecla, si se cierra, etc). *FreeGLUT* es la biblioteca que provee toda esa funcionalidad extra para la creación y gestión de las ventanas y sus eventos. Seleccionamos *FreeGLUT* nuevamente por ser una biblioteca muy fácil de aprender y que requiere muy poco código para su puesta en marcha (*boiler-plate-code*). Sin embargo, asociada a su simplicidad están sus limitaciones. No dispone, por ejemplo, de ningún widget/control (botones, listas, cuadros de texto, etc). Programas más complejos podrían requerir embeber el contexto *OpenGL* en ventanas de otras bibliotecas para GUIs, como pueden *wxWidgets* o *QT*.

[1] la mayoría de los ejemplos funciona con la versión 1.1 (que data de 1997). En casos excepcionales se requiere alguna funcionalidad adicional, pero siempre presente, aunque sea como extensión, en la serie 1.x. Dado que son todas versiones que llevan muchos años disponibles (la versión 2.0 se publicó en 2004), es casi imposible que un entorno de desarrollo actual no soporte estas características.

2. Modelo vs GUI

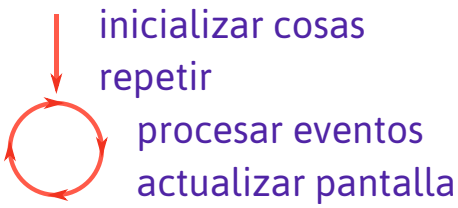
Para empezar a hablar sobre programas con interfaces gráficas, primero hay que distinguir lo que se entiende por modelo y lo que se entiende por interfaz en este contexto. Tomando, por ejemplo, un programa que gestiona los alquileres en una biblioteca, el modelo podría estar conformado por las clases Socio, Libro, Alquiler y Biblioteca. Estas clases contienen los algoritmos y estructuras de datos para manipular esa información, y en general estas cosas no dependen de una interfaz gráfica (o de cualquier otro tipo) en particular. Por ejemplo, habrá contenedores para libros, socios y alquileres, y métodos para buscar un socio, para validar si un libro está disponible, y para asentar el alquiler del mismo al socio. Por otro lado, hay una interfaz que permite consultar y modificar los datos que gestiona el modelo (las ventanas que usa el usuario). Esta interfaz despliega en pantalla los datos que guarda el modelo, y traduce los eventos del usuario en operaciones sobre los mismos.

En una aplicación gráfica, el mecanismo es el mismo. En general el modelo describe ”matemáticamente” los elementos a representar gráficamente, y la interfaz se encarga de llevarlos al monitor y de traducir los eventos en llamadas a funciones del modelo. Si un programa dibuja un segmento de recta, el modelo podría contener, por ejemplo, las coordenadas de los puntos extremos. Con esos datos, el código de la interfaz se encarga de representar el segmento en pantalla. Cuando el usuario mueve un extremo del segmento, por ejemplo arrastrándolo con el mouse, el programa debe modificar su modelo registrando las nuevas coordenadas, y luego dejar que el código de la interfaz muestre la nueva recta, de la misma forma que antes. Sería un error, pensar que el programa debe dibujar directamente la nueva recta, sin modificar los datos guardados en el modelo subyacente. Resumiendo: modificar el segmento no es dibujarlo en otra posición, sino cambiar sus coordenadas en el modelo interno; luego las rutinas de dibujo se encargarán de que se redibuje en otra posición.

Además es importante destacar que en los sistemas modernos toda la escena se reconstruye en cada frame. En los 90 era común que al mover un objeto en pantalla, el programa solo actualice la zona correspondiente a dicho objeto, por ejemplo, pintando sobre la posición vieja con el color de fondo para borrarlo, y dibujandolo luego en la nueva posición. Con el hardware actual, es más habitual y conveniente redibujar tooda la escena/pantalla desde cero en cada frame, en lugar de tratar de rastrear qué partecita cambió para actualizar solo eso. Esto también ayuda a la separación entre modelo e interfaz, e implica que no importa cual sea el cambio que hagamos en el modelo, la función que se encargue de renderizarlo lo ”considerará” automáticamente.

3. El Bucle de Eventos

Con estas ideas en mente, podemos explicar el siguiente diagrama, al cual se apegan en mayor o menor medida casi todas las aplicaciones gráficas:



1. Cuando comienza la ejecución del programa, este define un contexto inicial, que involucra tareas como crear la ventana y cargar los datos iniciales.
2. Luego comienza un bucle que se repite, generalmente hasta que finaliza el programa. Dentro de este bucle se llevan a cabo dos tareas:

    a) Procesar eventos: el programa debe reaccionar ante ciertos eventos en la interfaz gráfica. Los eventos pueden ser los clicks o movimientos del mouse, la presión de una tecla, el cambio de tamaño de la ventana, etc. En general, los eventos los genera directa o indirectamente el usuario mediante los dispositivos de entrada (aunque un evento podría ser “transcurrió x cantidad de tiempo”).
    b) Se actualiza la pantalla. El programa reacciona ante los eventos modificando sus datos, y estos cambios deben verse reflejados en la interfaz.

Los detalles de cómo se detectan los eventos, o se actualiza el contenido de la ventana, dependen de las bibliotecas que se utilicen para ello. Algunas bibliotecas tienen objetos o funciones para preguntar qué eventos están ocurriendo, y otros objetos o funciones para dibujar elementos en la pantalla. En estos casos, el programa cliente debe implementar el bucle que se mencionó anteriormente. SFML es un ejemplo de biblioteca que adopta este mecanismo. En otros casos, como en GLUT, la biblioteca implementa el bucle y el programador no puede modificarlo. Pero entonces, ¿cómo hace el programa cliente para reaccionar ante eventos en el bucle si no puede modificarlo? La respuesta es utilizando *callbacks*.

4. Callbacks y GLUT

Se llama *callback* a una función/método (A) que es pasado a otra función/método de una biblioteca (B) para que, más tarde, B pueda invocar a A. En C/C++, un *callback* es generalmente un puntero a función o método. Las bibliotecas gráficas tienen métodos/funciones para ”registrar” *callbacks*. Esto es, para decir a qué funciones queremos que invoquen. En general, hay *callbacks* específicos para cada tipo de evento, que la biblioteca utilizará en el bucle de eventos para cederle el control al programa cliente cuando ocurran.

La biblioteca *GLUT* trabaja de esta forma. En la etapa de inicialización, al construir la ventana, el programa cliente debe indicar qué función asociar a cada tipo de eventos (a qué función llamar cuando alguien hace click, a qué función llamar cuando alguien cambia el tamaño de la ventana, etc). Luego de que el programa realiza todas las inicializaciones necesarias, le cede el control a la biblioteca, y partir de allí la biblioteca ejecuta ese bucle esperando a que ocurra algún evento de interés, y cuando esto ocurre invoca a la función asociada a ese tipo de evento.

Por ejemplo, un programa que utilice *GLUT* podría definir ciertas funciones:

```
void my_reshape_cb (int w, int h) {
    /*...que hacer cuando la ventana cambia de tamaña...*/
}
void my_mouse_cb (int but, int st, int x, int y); {
     /*...que hacer cuando el usuario hace click...*/
}
void my_keyboard_cb (char key, int x, int y); {
    /*...que hacer cuando se presiona una tecla...*/
}
```

e incluir en el `main` :

```
/*...algunas inicializaciones...*/
glutReshapeFunc (my_reshape_cb); // si cambia de tamaño la ventana, llamar a my_reshape_cb
glutMouseFunc (my_mouse_cb); // si el usuario hace click, llamar a my_mouse_cb
glutKeyboardFunc (my_keyboard_cb); // si presiona una tecla, llamar a my_keyboard_cb
/*...otras inicializaciones...*/
glutMainLoop(); // entrar en el loop principal de la biblioteca
```

Hay que notar un detalle importante: las funciones `glutXxxFunc` no dicen para cual ventana están registrando el *callback* (el programa podría tener varias ventanas). Esto es porque *GLUT* trabaja con un estado global. Todo lo que una función no reciba como argumento, lo toma de ese ”estado”. Es decir, la biblioteca guarda una configuración interna (el estado), y usa eso en cada llamada. Ese estado dice, entre otras cosas, qué ventana utilizar. Y esta política se aplica en todos los aspectos. Por ejemplo, al crear una ventana, la función que lo hace no recibe entre sus argumentos ni el tamaño ni la posición de la ventana, sino que solo el nombre. Entonces ¿cómo definimos tamaño y posición? Hay funciones para modificar ese estado interno. Debemos invocar estas funciones antes de crear la ventana para que al crearla, el estado contenga la posición y el tamaño que queremos:

```
glutInitWindowSize (640,480); // la próx. ventana tendrá tamaño 64x480
glutInitWindowPosition (100,100); // la próx. ventana estará en la posición 100,100
glutCreateWindow ("VEjemplo"); // crear una ventana nueva con título "VEjemplo"
```

Si creásemos otra ventana inmediatamente a continuación, también se colocaría en la misma posición y con el mismo tamaño, ya que el estado se mantiene, a menos que lo volvamos a modificar antes con otras llamadas a `glutInitWindow`. Siguiendo la misma lógica, al llamar a `glutCreateWindow` definimos como ventana actual en el estado a esa nueva ventana, y todas las siguientes llamadas a `glutXxxFunc` registraran callbacks para esa ventana, hasta que creemos una nueva, o cambiemos la ventana ”actual” en el estado con `glutSetWindow`.

Esta forma de definir las cosas, utilizando [una máquina de] estados en lugar de utilizar métodos con muchísimos argumentos, u argumentos de tipos complejos (structs con muchísimos campos), *GLUT* la copia de *OpenGL*.

5. OpenGL

Como aclaramos anteriormente, *GLUT* solo gestiona la creación de la ventana y la detección de sus eventos, pero no tiene funciones para dibujar dentro de la misma. Para dibujar, utilizamos las funciones de *OpenGL* (que en contraparte, solo tiene funciones relacionadas al dibujo, pero nada relacionado a la creación de ventanas o el manejo de eventos). *OpenGL* utiliza una máquina de estados, y su estado es también global[2]. Al dibujar un punto en OpenGL, decimos solo sus coordenadas, pero el color y el tamaño se toman del estado. Igualmente, cuando en OpenGL decimos que vamos a dibujar con rojo, todo lo que siga hasta que se ejecute el próximo cambio de color, sea un solo punto o sean un millón de triángulos, va a ser rojo. Más aún el tipo de primitiva que uno dibuja también es parte del estado:

```
glColor3f(1.0,0.0,0.0); // vamos a dibujar en rojo (rgb=1;0;0)
glBegin(GL_TRIANGLES); // vamos a dibujar triángulos
// ahora, todos los puntos que enviemos, serán considerados vértices de
// triángulos; OpenGL dibujará un triángulos por cada tres puntos
for(int i=0;i<n;i++) {
    glVertex2i(triang[i].x0,triang[i].y0);
    glVertex2i(triang[i].x1,triang[i].y1);
    glVertex2i(triang[i].x2,triang[i].y2);
}
glEnd(); // fin de los triangulos
```

Se puede notar que los nombres de todas las funciones de *GLUT* comienzan con el prefijo `glut`, mientras que todas las de *OpenGL* con el prefijo `gl`, y a continuación una o pocas palabras que indican qué hace (todo escrito en formato *camelCase*[3]). Algunas funciones de *OpenGL* tienen variantes para distintos tipos y cantidades de datos y esto también se manifiesta en la convención de nombres[4]. Por ejemplo, un punto se puede definir con 2, 3 o 4 dimensiones, cada coordenada puede ser `int`, `float`, `double`, etc, y además se pueden pasar cada una en un argumento diferente, o todas juntas en un arreglo. Entonces, para estos casos, las funciones de *OpenGL* agregan al final un sufijo que indica la cantidad de argumentos primero (un número), el tipo luego (un carácter), y si es un vector (la letra v) o no (nada):

$$\texttt{glAlgunaFuncion}\texttt{N}\texttt{t}\texttt{v}(\dots)$$

- `N` es la dimensión, puede ser `2`, `3` o `4`
- `t` indica el tipo:

[2] Generalmente es un error utilizar un estado "global" al diseñar una API. OpenGL es así por razones históricas, y esto también hace que en algunos sentidos sea más fácil de utilizar. Sin embargo, APIs modernas como *Vulkan* han abandonado este concepto.

[3] *Camel case* hace referencia al modo de componer un identificador mediante varias palabras. En este formato la palabras van pegadas, utilizando iniciales en mayúsculas para marcar dónde termina una palabra y comienza otra. Ej: `glutMainLoop`, `glutCreateWindow`, `glGetError`, `glLoadMatrix` etc. C++, en cambio, utiliza *Snake case*, que consiste en separar con guión bajo: `max_element`, `stable_sort`, `shared_ptr`, `priority_queue`, etc.

[4] *GLUT* y *OpenGL* son APIs diseñadas para C, no para C++. Dado que en C no existía la sobrecarga de funciones, necesariamente se debían utilizar distintos nombres de función para distintos argumentos.

  - `b`: byte (`signed char`)
  - `s`: `short`
  - `d`: `int`
  - `f`: `float`
  - `d`: `double`
  - si antes de la letra se agrega una `u`, se utiliza la versión `unsigned` del mismo tipo

- `v`: indica que se pasa un arreglo (puntero)
  - para pasar los `N` argumentos individualmente en lugar de usar un arreglo simplemente se omite la `v` (no va otra letra a cambio).

- Todas estas ”partes” son opcionales, y no se incluyen cuando no son necesarias (cuando hay una sola versión de la función).
- Ejemplos:

```
double r=1.0, g=0.5, b=0.7;
glColor3d(r,g,b); // 3 doubles
```

```
float s=0.25f, t=0.50f;
glTexCoord2f(s,t); // 2 floats
```

```
int p[2] = { 10, 20 };
glVertex2iv(p); // un puntero a 2 ints
```

```
glScaled(2.0,1.0,1.0); // solo se aclara el tipo, todas las versiones reciben 3 argumentos
```

6. Estructura mínima de un TP

Analizaremos ahora la mínima estructura presente y habitual en todos programa los programas que entrega la cátedra para realizar los trabajos prácticos.

6.1. La función `main`

Todo programa C/C++ comienza por la función `main`:

```
int main(int argc, char *argv[]) {
    glutInit(&argc,argv);
    initialize();
    glutMainLoop();
}
```

- `glutInit` es una función de la biblioteca *GLUT*, y es necesario invocarla antes de utilizar cualquier otra función de la misma.
- `initialize` es una función del propia del programa, allí colocamos usualmente todo lo relacionado a la inicialización del programa y su estado (modelo inicial, ventana, *callbacks*, estado OpenGL inicial, etc).
- `glutMainLoop` es el punto en que le cedemos el control a la biblioteca. La biblioteca esperará a que ocurran eventos e invocará a nuestros *callbacks* cuando esto pase. En algún sentido este loop puede verse como infinito, en general la aplicación se mantienen allí hasta que termina (hasta que se cierra su ventana).

6.2. Inicialización

Veamos ahora un ejemplo de función `initialize`

```
void initialize() {
    // 1. crear la ventana
    glutInitDisplayMode(GLUT_DEPTH|GLUT_RGBA|GLUT_DOUBLE);
    glutInitWindowSize(w,h); glutInitWindowPosition(50,50);
    glutCreateWindow("Ejemplo GLUT+OpenGL");
    // 2. definir los callbacks
    glutDisplayFunc(display_cb);
    glutReshapeFunc(reshape_cb);
    glutKeyboardFunc(keyboard_cb);
    glutMotionFunc(motion_cb);
    glutMouseFunc(mouse_cb);
    // 3. estado inicial de OpenGL
    glClearColor(1,1,1,1);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);
    ...
}
```

1. Primero se crea la ventana. Las dos primeras lineas preparan el estado (tamaño, posición, formato del *framebuffer*) para luego invocar a `glutCreateWindow`.
2. Luego asociamos *callbacks* para la ventana recién creada. `glutDisplayFunc` (callback para cuando hay que redibujar) y `glutReshapeFunc` (para cuando se define o cambia el tamaño de la ventana) son los mínimos necesarios para poder dibujar. Además, usualmente habrá otros para los eventos que genere el usuario. Los identificadores que se le pasan (`xxx_cb`) son nombres de funciones que definirá el programa. Los prototipos de estas funciones no son libres, sino que están especificados por *GLUT*.
3. Finalmente se configura el estado inicial de OpenGL, especialmente aquel que no vallamos a modificar durante el redibujado (una vez configurado, permanece en ese ”estado”).

6.3. Modelo y estado propio

Dado que los callbacks de *GLUT* deben tener un prototipo predefinido, y que este solo incluye información sobre el evento en cuestión y no deja espacio para información adicional que quiera agregar el programador, no queda otra opción más que definir el estado mediante variables globales (en general esto no es una buena práctica, pero sí es la forma más fácil, y de algún modo *GLUT* nos obliga a hacerlo). Para este ejemplo, supongamos que vamos a dibujar segmentos en 2D:

```
struct Segmento { int x0, y0, x1, y1; };
std::vector<Segmento> vsegmentos;
int seg_sel = -1, ext_sel = -1;
```

Si queremos que la aplicación nos permita interactuar con los segmentos (por ejemplo, mover sus extremos), necesitaremos además de los mismos, información para saber dentro de un evento si había un segmento seleccionado (y si lo había, cual de sus extremos estamos queriendo mover, para ello son `seg_sel` y `ext_sel`).

6.4. Callback de dibujo

```
void display_cb() {
    glClear(GL_COLOR_BUFFER_BIT); // "borrar" todo
    glColor3f(1,1,0); glLineWidth(3); // definir estilo de linea
    glBegin(GL_LINES); // dibujar lineas
    for(Segmento &s:vsegmentos) {
        glVertex2i(s.x0, s.y0);
        glVertex2i(s.x1, s.y1);
    }
    glEnd();
    if (seg_sel!=-1) { // si hay un punto seleccionado...
        glColor3f(1,0,0); glPointSize(5); // definir estilo de puntos
        glBegin(GL_POINTS); // dibujar punto seleccionado
        if (ext_sel==0) glVertex2i(vsegmentos[seg_sel].x0, vsegmentos[seg_sel].y0);
        else            glVertex2i(vsegmentos[seg_sel].x1, vsegmentos[seg_sel].y1);
        glEnd();
    }
    glutSwapBuffers(); // mostrar resultado
}
```

La función que dibuja normalmente redibuja toda la escena. Por ello primero ”borra” (repinta todo de ”blanco”), y luego dibuja todo nuevamente, siguiendo la lógica de máquina de estado que se describió antes. El dibujo no se hace directamente sobre la pantalla, para que no se pueda llegar a ver el proceso (que no se vea cómo se va armando la imagen si es compleja), sino que se hace en un *buffer* (espacio de memoria) auxiliar y cuando está listo se intercambia (*swap*) con el que se muestra en pantalla. Este mecanismo se conoce como *double-buffering*.

6.5. Otros callbacks

Dijimos que era necesario definir un *callback* para el *reshape.* Este define el espacio/sistema de coordenadas a utilizar. Dejaremos los detalles de su implementación para la unidad de transformaciones. Veamos a continuación los que generan la interacción con el usuario (los que responden a teclado y mouse). Por ejemplo, si queremos seleccionar un punto al hacer click, y moverlo al arrastrarlo:

```
void mouse_cb(int button, int state, int x, int y) { // click
    y = win_h-y;
    if (button==GLUT_LEFT_BUTTON) { // boton derecho...
        if (state==GLUT_DOWN) { // si apreta el botón...
            // seleccionar el extremo más cercano
            double min_dist = std::numeric_limits<double>::max();
            for(size_t i=0; i<vsegmentos.size(); ++i) {
                double d0 = distancia(x,y,vsegmentos[i].x0,vsegmentos[i].y0);
                if (d0<min_dist) { min_dist=d0; seg_sel=i; ext_sel=0; }
                double d1 = distancia(x,y,vsegmentos[i].x1vsegmentos[i].y1);
                if (d1<min_dist) { min_dist=d1; seg_sel=i; ext_sel=1; }
            }
        } else { // si suelta el botón...
            seg_sel = ext_sel = -1;
        }
        glutPostRedisplay(); // actualizar pantalla
    }
}
```

```
void motion_cb(int x, int y) { // drag
    y = win_h-y;
    if (seg_sel!=-1) {
        if (ext_sel==0) { vsegmentos[seg_sel].x0=x; vsegmentos[seg_sel].y0=y; }
        else            { vsegmentos[seg_sel].x1=x; vsegmentos[seg_sel].y1=y; }
        glutPostRedisplay(); // actualizar pantalla
    }
}
```

La linea `y = win_h-y` presente habitualmente en *callbacks* del mouse se debe a que los sistemas de coordenadas del modelo y de la imagen suelen tener los ejes y opuestos. En las imágenes *raster* es habitual que el $(0,0)$ se encuentre arriba a la izquierda, mientras que en el modelo vectorial lo consideramos abajo a la izquierda; de aquí la necesidad de invertirlo considerando el alto de la ventana (que se asume guardado en alguna variable global `win_h`).

Es importante notar además que el *motion*, luego de modificar los datos del segmento, invoca a `glutPostRedisplay` para que se genere un evento de redibujado, y entonces se ejecute la función `display_cb` para que los cambios se vean reflejados en pantalla [5].

[5] si bien algunos ejemplos invocan directamente a `display_cb`, es mejor hacerlo mediante la función de *GLUT*. Esta función invocará al redibujado "cuando pueda" y no inmediatamente, ya que si el redibujado es muy costoso, puede que sea necesario procesar varios eventos entre un redibujado y otro para que la aplicación responda más rápido.

7. Referencias

- OpenGL:
  - https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/
  - https://www.glprogramming.com/red/
- GLUT/FreeGLUT:
  - https://www.opengl.org/resources/libraries/glut/spec3/node1.html
  - http://freeglut.sourceforge.net/docs/api.php