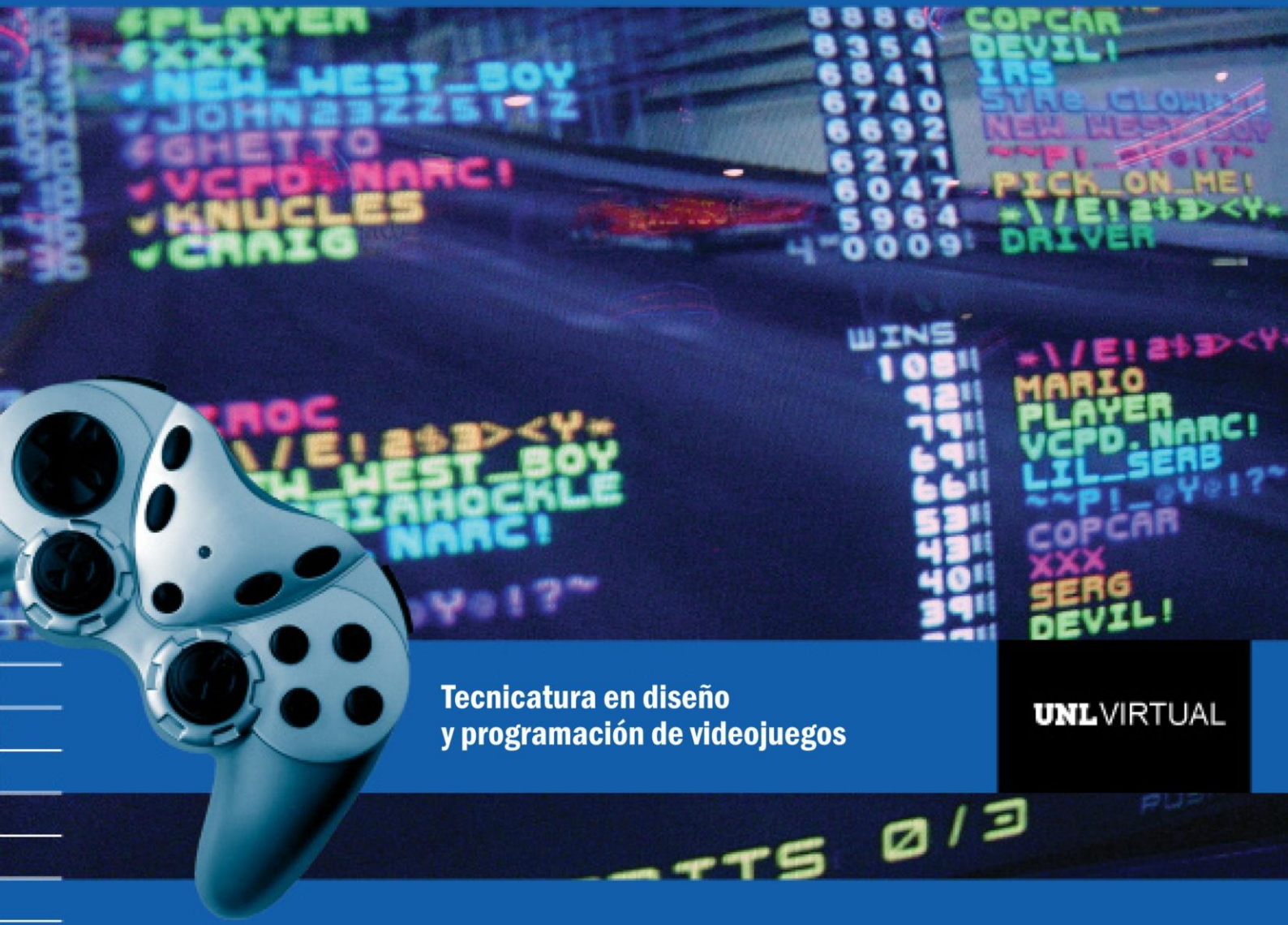
UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS



Tecnicatura en diseño
y programación de videojuegos

UNLVIRTUAL

Modelos y algoritmos para videojuegos I

Unidad 4

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

4. ¿HAS VISTO AL CONEJO BLANCO?

4.1. Introducción

Hasta aquí hemos visto algunas nociones básicas de renderizado y manipulación de imágenes, pero esto no es todo lo que necesitamos a la hora de programar nuestro videojuego. En realidad, como vimos anteriormente, es sólo una de las partes.

En esta unidad vamos a concentrarnos en la interactividad de nuestras aplicaciones, es decir, en cómo poder reaccionar ante las acciones del jugador.

El jugador interactúa con el videojuego mediante los llamados dispositivos de entrada, entre los cuales encontramos el teclado, mouse y el siempre bien ponderado joystick. Si bien existen otros dispositivos, no vamos a ocuparnos detalladamente de los mismos en esta unidad.

La captura de los estados de dichos dispositivos varían de plataforma a plataforma, pero existen conceptos que son aplicables a todas ellas.

En esta unidad veremos dichos conceptos, y luego, cómo se traducen en SFML.

4.2. Buffered vs. Unbuffered

Existen, principalmente, dos formas de trabajar con los dispositivos de entrada: *buffered* y *unbuffered*, que significa con y sin buffers, respectivamente.

Definamos, ante todo, qué es un buffer:

Buffer

Espacio en memoria donde se almacena temporalmente información mientras la misma espera ser procesada.

En el caso de los dispositivos de entrada, estos buffers contienen información sobre las teclas o botones presionados.

Imaginemos que en nuestra casa tenemos un timbre y cada vez que alguien llega toca el timbre para que lo atendamos. De esta forma, no vamos a dejar de atender a nadie. Y si llegan varias personas, van a ir tocando timbre en el orden en que llegan, por lo que los atenderemos en ese orden. Esto es lo que se llama *buffered*. Es decir, las personas que necesitan ser atendidas se van agrupando en un buffer y avisando que debemos atender dicho buffer tocando el timbre. Si el buffer esta vació no sonará el timbre y no debemos preocuparnos por ello.

Por otro lado, supongamos ahora que el timbre se nos rompió y que la única forma de saber si hay alguien afuera para ser atendido es saliendo a mirar regularmente. Si alguien llega, no tiene forma de avisarnos. Puede suceder que alguien llegue y, como no salimos a mirar, se vaya. Nosotros sólo sabemos si hay alguien afuera en el instante en que salimos a mirar. No sabemos si vino alguien en el período en que no miramos; sólo sabemos si en ese preciso momento hay alguien. Esto es lo que se denomina *unbuffered*. No hay buffer de personas; las personas no hacen cola, porque no tienen forma de avisar que vinieron. Simplemente pueden esperar que salgamos justo en el momento en que ellos están. Si se van, nunca nos vamos a enterar que vinieron.

Unbuffered está relacionado con el estado actual de los dispositivos de entrada, mientras que *buffered* tiene en cuenta también lo que sucedió en el transcurso del tiempo en que no salimos a mirar.

Ambas formas de trabajar se utilizan actualmente y la elección de una u otra depende del tipo de juego con el que estemos trabajando.

Por ejemplo, si trabajamos en un videojuego de estrategia manejado por el mouse, es importante capturar todos los clicks que hace el usuario. Por ello, deberíamos usar buffered. De no ser así, podría suceder que el usuario haga un click y no nos enteremos, por lo que la unidad comandada en el juego no se moverá. Para que no ocurra esto, el usuario tendría que hacer click y mantenerlo lo suficiente para que nuestro programa chequee el estado y lo procese.

En cambio, si estuviésemos haciendo un First Person Shoter (FPS), es mejor administrar de manera unbuffered el movimiento de nuestro personaje por teclado, ya que sólo importa si en determinado momento el usuario está apretando hacia adelante. Y por las características del juego, es una tecla que se mantendrá apretada por el tiempo en que el usuario desee avanzar. Si utilizáramos buffered, estaríamos recibiendo miles de eventos inútilmente, que sobrecargarían nuestro videojuego.

Esto último es un detalle importante para mencionar acerca de las entradas buffered. Pues, para que existan las mismas, debe haber una arquitectura de eventos que le comunique al desarrollador los sucesos que han ocurrido.

Esta arquitectura tiene su costo de procesamiento asociado, que es necesario en ciertas aplicaciones, pero que en otras puede tornar a nuestro videojuego en algo imposible de jugar.

Evento

Suceso ocurrido en el sistema. Puede ser generado por el usuario o por el mismo sistema operativo.

4.3. De eventos y algo más

Antes de adentrarnos en la práctica, debemos entender qué es un evento en el contexto del desarrollo de videojuegos.

Los eventos van más allá de la arquitectura para entradas (input). Un evento es un mensaje que nos avisa que algo sucedió y nos da la posibilidad de reaccionar ante tal suceso. Es decir, es una forma de recibir notificaciones de sucesos externos e internos de nuestra aplicación. A medida que van ocurriendo, los eventos se ubican en un buffer, el cual luego es procesado, de modo que ningún evento se pierde. Por ello, esta arquitectura de eventos resulta la opción natural para hacer *buffered input*.

Los eventos pueden ser de distinto tipo y la mayoría de lo que ocurre en nuestra aplicación es comunicada mediante eventos.

Por ejemplo, si el usuario cierra la ventana, se producirá un evento avisando que hizo click en la X; o si la ventana cambió de tamaño, se generará un evento indicando los nuevos tamaños.

La forma de manejar los eventos varía de librería a librería. Aquí nos enfocaremos en cómo lo hace SFML, recurriendo a un ejemplo.

4.3.1. Eventos cercanos del primer tipo

En SFML, los eventos se van acumulando en una cola y en cada ciclo de nuestro programa debemos recuperarlos para procesarlos.

Veamos cómo luce un loop de procesamiento de eventos típico:

```
sf::Event Event;
while (App.GetEvent(Event)){

	// Procesar el evento

	}
```

Allí se observa que declaramos un objeto de tipo *Event*. El mismo es el que utiliza la aplicación para devolvernos información de un evento. `App.GetEvent()` es la función que se fija si existe un evento y, de ser así, devuelve *true* y pone en su argumento Event información sobre el mismo.

Esta función debe ser llamada hasta que no queden eventos en el buffer que describíamos antes. Pero, ¿cómo hacemos para saber cuántos eventos faltan procesar?

Es sencillo: si `App.GetEvent()` devuelve *false,* es porque no hay más eventos y podemos continuar. De no ser así, esto es, mientras devuelva *true*, tenemos que seguir preguntando si quedan eventos. Por esta razón, el loop se realiza mientras queden eventos. Si no los procesamos a todos en cada ciclo, nuestra aplicación seguramente va a comportarse mal, ya que no nos estamos enterando de los comandos del usuario ni del sistema.

En secciones anteriores, cuando creábamos nuestras ventanas de aplicación, veíamos que no podíamos cerrarlas por más que hiciéramos click en el botón *cerrar*. Esto ocurría porque no estábamos procesando los eventos y nuestra ventana nunca se enteraba de que debía cerrarse.

Ahora veamos cómo quedaría el código teniendo en cuenta dicha consideración:

Main.cpp

```
////////////////////////////////////////////////////////////
/// Punto de entrada a la aplicación
////////////////////////////////////////////////////////////
int main()
{

    //Creamos la ventana de la aplicacion
    sf::RenderWindow App(sf::VideoMode(800, 600, 32), "Sprites");

       while (App.IsOpened())
       {
              sf::Event Event;
              while (App.GetEvent(Event))
              {
                     // Si se cerró la ventana
                     if (Event.Type == sf::Event::Closed)
                            App.Close();

              }

              App.Display();
       }

    return 0;
}
```

El código es idéntico al utilizado anteriormente, sólo que ahora hemos agregado el loop de eventos ya descrito.

Dentro de ese loop debemos fijarnos si el usuario cerró la ventana y, de ser así, salir de la aplicación (eso es lo que realizan las líneas):

```
                    // Si se cerró la ventana
                    if (Event.Type == sf::Event::Closed)
                          App.Close();
```

El objeto Event tiene una propiedad llamada *type,* que almacena el tipo de evento que sucedió.

Los tipos de eventos son miembros del namespace *sf* y miembros de la clase *Event*. Por ello debemos utilizar el nombre completo `sf::Event::Closed`. Si el tipo de evento coincide con el de cerrar la ventana, entonces lo hacemos utilizando el método `App.Close()`.

Si compilamos y ejecutamos el código tendremos una ventana que, si tratamos de cerrar, ahora sí responderá al evento que genera el usuario.

En SFML, los tipos de evento que podemos encontrar son los siguientes:

- `Closed`
- `Resized`
- `LostFocus`
- `GainedFocus`

Estos eventos son de ventana y nos indican si se cerró la aplicación, se cambió su tamaño o hubo algún cambio en el foco.

Luego tenemos los eventos de teclado:

- `TextEntered`
- `KeyPressed`
- `KeyReleased`

De mouse:

- `MouseWheelMoved`
- `MouseButtonPressed`
- `MouseButtonReleased`
- `MouseMoved`
- `MouseEntered`
- `MouseLeft`

Y por último, los eventos de joystick:

- `JoyButtonPressed`
- `JoyButtonReleased`
- `JoyMoved`

En los próximos parágrafos, brindaremos más detalles sobre cada uno de ellos.

4.4. Sprite arriba, sprite abajo

Vamos a ver ahora cómo trabajamos con el teclado.

Primero, exploraremos el método buffered y luego veremos el unbuffered.

4.4.1. Buffer si

Como dijimos anteriormente, en algunos casos es útil trabajar con un buffer para el teclado.

Un ejemplo de esto es cuando queremos que nuestra aplicación termine en el momento en que el usuario presiona *escape*. Para ello, debemos recibir siempre que el usuario accione la tecla y no podemos permitirnos perder dicho evento, por lo que utilizaremos buffers.

Dentro de los eventos de teclado tenemos:

- `TextEntered`
- `KeyPressed`
- `KeyReleased`

De los cuales los últimos dos son los que utilizaremos.

- `KeyPressed:` nos indica que una tecla fue presionada.
- `KeyReleased:` nos indica que una tecla fue soltada.

Como dijimos, el objeto Event nos brinda más información sobre el evento. En el caso de estos dos, la información que recibimos es la siguiente:

- `Event.Key.Code:` contiene el código de la tecla que fue presionada o soltada.
- `Event.Key.Alt`: indica si *alt* se encuentra presionada.
- `Event.Key.Control`: indica si *control* se encuentra presionada.
- `Event.Key.Shift`: indica si *shift* se encuentra presionada.

Es decir, leyendo dichos atributos del objeto Event, podemos obtener esa información.

Veamos, por ejemplo, cómo haríamos si queremos que nuestra aplicación salga cuando presionamos *escape*.

Lo primero que debemos hacer es agregar al loop de eventos un *if*, preguntando si existió un evento de tipo *Keypress*.

```
    if (Event.Type == sf::Event::KeyPressed)
```

Si esta sentencia es verdadera, entonces debemos fijarnos si se ha presionado la tecla que nos interesa, es decir, *escape*. Para hacer esto, debemos comparar `Event.Key.Code`, que contiene el código de la tecla apretada, con el código que deseamos.

```
       if(Event.Key.Code == sf::Key::Escape)
                                App.Close();
```

Por sencillez de programación, existe un *enum* definido en el namespace *sf,* que contiene todas las teclas posibles.

Para acceder al mismo, debemos utilizar el nombre del namespace y el nombre del enum de la siguiente manera `sf::Key::[Nombre tecla]`.

Si coincide, debemos cerrar la aplicación y lo realizamos de la misma manera que lo hicimos anteriormente.

4.4.2. Buffer no

Lo expuesto anteriormente tiene una aplicación acotada debido al *delay* (retardo) en la entrega de los eventos. La alternativa es utilizar *input* sin buffers (esta técnica que veremos también se conoce como *pooling*).

SFML nos da la posibilidad de saber si en un determinado momento una tecla se encuentra presionada o no. Para ello se utiliza el objeto *Input*. El mismo nos da la información en cualquier momento sobre el estado de los dispositivos de entrada.

```
const sf::Input& in = App.GetInput();
```

Para obtener este objeto, utilizaremos el método `App.GetInput()`. Luego, emplearemos los métodos de Input para obtener la información.

Por ejemplo, si queremos saber si una tecla se encuentra presionada en determinado momento, nos devuelve *true* si la tecla se encuentra apretada.:

```
Input.IsKeyDown(sf::Key::[Tecla]);
```

Ahora veamos como quedaría el ejemplo del *escape* para este caso:

Main.cpp

```
////////////////////////////////////////////////////////////
/// Punto de entrada a la aplicación
////////////////////////////////////////////////////////////
int main()
{

       sf::Event Event;
    //Creamos la ventana de la aplicacion
    sf::RenderWindow App(sf::VideoMode(800, 600, 32), "Input");

       const sf::Input& in = App.GetInput();

       while (App.IsOpened())
       {

       while (App.GetEvent(Event)){

             // Cerrar ventana
             if (Event.Type == sf::Event::Closed)
                 App.Close();

             }

              // Cerrar ventana si presiono escape
              if(in.IsKeyDown(sf::Key::Escape)==true)
                    App.Close();

              App.Display();
       }

    return EXIT_SUCCESS;
}
```

Nótese que el loop de eventos también está en este fragmento de código. Esto sucede porque siempre debemos atender los eventos de la aplicación aunque no utilicemos los de input. No obstante, el evento de cerrar siempre debe estar.

En este ejemplo también podemos ver claramente cómo conviven los eventos con la entrada sin buffers.

4. 5. Oh oh, no ese punto rojo de nuevo

El mouse es principalmente lo que diferencia a la plataforma PC de las demás plataformas de videojuegos; es un periférico que hace sencilla la interfaz de selección y comando. Por ello, los juegos de estrategia son naturales en PC y no ha habido demasiada proliferación en otras plataformas.

Lo mismo ocurrió con los FPS, que primero existieron en PC y luego se fueron extendiendo a otras plataformas, aunque con algunos cambios para suplir la falta de mouse en las consolas de sobremesa. Por ejemplo, el *auto aiming* (auto apuntar) del *Mass Effect*. Esto también implicó grandes dificultades a la hora de portar un videojuego de PC a las consolas, que hiciera uso extensivo del mouse.

4.5.1. El mouse y los buffers

Como dijimos anteriormente, el mouse es uno de los mayores usuarios del buffer, especialmente para los botones.

Veamos los eventos que utilizaremos durante el curso:

- `MouseWheelMoved` : cuando se mueve la rueda
- `MouseButtonPressed` : cuando se aprieta un botón
- `MouseButtonReleased`: cuando se suelta un botón
- `MouseMoved`: cuando se mueve el mouse

Imaginemos ahora que tenemos un videojuego en el que necesitamos un cursor para seleccionar y que ese cursor debe seguir la posición del mouse. ¿Cómo hacemos esto con buffered input?

Primero, debemos crear un sprite, que hará las veces de cursor, y cargar su material desde el disco. Luego, necesitamos trabajar sobre el loop de eventos.

En primer lugar, tenemos que ver si se produjo un evento de mouse. Para el cursor utilizaremos el último de los eventos listados, que nos indica si se movió y dónde se encuentra, para poder dibujar el sprite en dicha posición. Luego de cada iteración hay que obtener la nueva posición del mouse y actualizar la del sprite.

El evento MouseMoved nos brinda la siguiente información:

- `Event.MouseMove.X` : contiene la posición en x en coordenadas locales.
- `Event.MouseMove.Y` : contiene la posición en x en coordenadas locales.

Con dicha información vamos a actualizar la posición del sprite. El código queda:

```
////////////////////////////////////////////////////////////
// Librerías
////////////////////////////////////////////////////////////
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>


////////////////////////////////////////////////////////////
/// Variables
////////////////////////////////////////////////////////////

sf::Sprite cursor;
sf::Image mat_cursor;


////////////////////////////////////////////////////////////
/// Punto de entrada a la aplicación
////////////////////////////////////////////////////////////
int main()
{


    sf::Event Event;

    //Creamos el sprite para el cursor
    mat_cursor.LoadFromFile("..\\img\\cursor.png");
    cursor.SetImage(mat_cursor);
    cursor.SetPosition(0,0);

    //Creamos la ventana de la aplicacion
    sf::RenderWindow App(sf::VideoMode(800, 600, 32), "Input");
    //Ocultamos el cursor del sistema
    App.ShowMouseCursor(false);

       while (App.IsOpened())
       {

                while (App.GetEvent(Event))
        {
            // Close window : exit
            if (Event.Type == sf::Event::Closed)
                App.Close();

                    if(Event.Type==sf::Event::MouseMoved)
       //Actualizamos la posicion del sprite
       //con la informacion del evento del mouse
                        cursor.SetPosition(Event.MouseMove.X,
        Event.MouseMove.Y);

        }

             App.Clear();
             App.Draw(cursor);
             App.Display();
       }

    return EXIT_SUCCESS;
}
```

Veamos ahora, en particular, algunos fragmentos del código. Lo primero que debemos notar es la línea:

```
    //Ocultamos el cursor del sistema
    App.ShowMouseCursor(false);
```

Lo que hacemos aquí es indicarle al sistema operativo que no muestre el cursor, porque de lo contrario veríamos los dos superpuestos, el del sistema y el que dibujamos nosotros.

Luego, actualizamos la posición del sprite en cada ciclo, utilizando la información del objeto Event:

```
                    if(Event.Type==sf::Event::MouseMoved)
//Actualizamos la posicion del sprite
//con la informacion del evento del mouse
                            cursor.SetPosition(Event.MouseMove.X,
 Event.MouseMove.Y);
```

Y finalmente, limpiamos la pantalla y dibujamos el sprite.

Si ejecutamos dicho código podremos ver nuestro cursor sobre la ventana.

Intentá no ocultar el mouse del sistema.

¿Te parece que podría resultar útil no ocultarlo en alguna aplicación?

4. 5.2. El mouse y nadie más

Ahora veamos cómo luce nuestro código sin utilizar los buffers.

Para acceder al estado del mouse se utiliza el mismo objeto que empleamos para el teclado *sf::Input*.

El código es el mismo; sólo varía el loop principal para utilizar el objeto Input.

```
const sf::Input& in = App.GetInput();

       while (App.IsOpened())
       {

                while (App.GetEvent(Event))
         {
              // cerrar ventana
              if (Event.Type == sf::Event::Closed)
                  App.Close();
         }

               cursor.SetPosition(in.GetMouseX(),in.GetMouseY());

               App.Clear();
               App.Draw(cursor);
               App.Display();
       }
```

El objeto Input posee los métodos `GetMouseX()` **y** `GetMouseY()` para obtener las posiciones en coordenadas locales de ventana del mouse. Luego, simplemente actualizamos la posición del sprite, como en el caso anterior.

4.6. Oh... bonita comodidad

Por último, en este capítulo vamos a ver algunas cuestiones relacionadas a otros dispositivos de entrada, denominados de manera general *joysticks*.

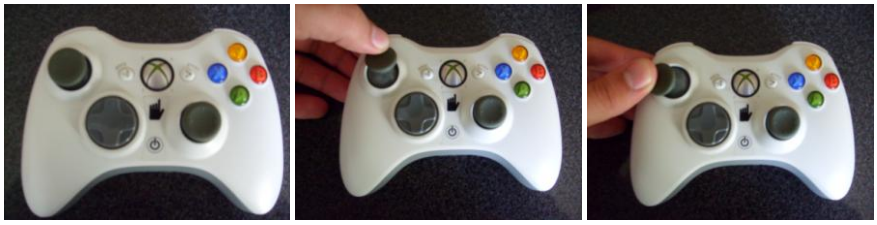
Al igual que en los casos anteriores, los joysticks pueden funcionar con o sin buffers y se aplican las mismas consideraciones.

En general, en los joysticks tenemos dos tipos de elementos: botones digitales y ejes analógicos.

Los botones pueden tener sólo dos estados: presionados o no. Estos botones se comportan del mismo modo que las teclas del teclado.

Por otro lado, los ejes analógicos pueden tomar una gran cantidad de valores entre la posición de equilibrio y los extremos. Por ello no es posible representarlos con un solo valor, sino que se utiliza un flotante que va desde 0, cuando el joystick se encuentra en el centro, hasta 1/-1, cuando el joystick se encuentra en el extremo.

Un ejemplo de esto es un joystick de Playstation® o de Xbox®, donde cada uno de los sticks analógicos poseen dos ejes, uno en *x* y otro en *y*. Cuando el stick está en el centro, ambos ejes se encuentran en 0, y al moverlo, este valor varía.



En estas imágenes se puede apreciar un joystick de Xbox® 360 ejemplificando lo explicado.

En la primera, el stick izquierdo se encuentra en el centro, por lo que las coordenadas del eje *x* e *y* son (0,0).

En la segunda imagen podemos ver el stick inclinado hacia arriba. En este estado, el eje *y* se encuentra en 1 y el *x*, en 0.

Lo opuesto sucede en la imagen de la derecha, donde el eje *y* se encuentra en 0 y el eje *x* se encuentra en -1. Los valores extremos de posición suelen depender de la librería que estemos utilizando.

Las configuraciones de los joysticks pueden ser muy dispares, al igual que su forma, pero siempre se reducen a un conjunto de botones digitales y ejes analógicos. Por lo tanto, lo que veremos aquí se aplica a cualquier dispositivo.

Si trabajamos con buffers, los eventos que podemos recibir del joystick son los siguientes:

- `JoyButtonPressed` : se presionó un botón
- `JoyButtonReleased` : se soltó un botón
- `JoyMoved` : se movió algún eje

Con sólo estos tres eventos podemos obtener toda la información que necesitemos del joystick.

En el caso los dos primeros eventos, la información que podemos recuperar del objeto Event es la siguiente:

- `Event.JoyButton.JoystickId` : el índice del joystick que fue accionado. Este valor puede ser 0 o 1, dependiendo de si fue el primer o segundo joystick conectado.
- `Event.JoyButton.Button` : el número de botón que fue presionado.

Si, en cambio, el evento es *JoyMoved,* la información que podemos recuperar del objeto Event es la siguiente:

- `Event.JoyMove.JoystickId` : el índice del joystick.
- `Event.JoyMove.Axis` : contiene información sobre qué eje fue movido.
- `Event.JoyMove.Position` : contiene la posición del eje, que en el caso de SFML varía entre -100 y 100. para los ejes tradicionales. y entre 0 y 360 para el POV. que es un eje especial utilizado en los joysticks para simuladores de vuelo.

Por otro lado, si trabajamos sin buffers. para obtener información del estado del joystick utilizamos –al igual que en las secciones anteriores– el objeto Input y sus métodos, entre los cuales podemos destacar:

```
Input.IsJoystickButtonDown(indice_joystick, indice_boton)
```

Éste nos devuelve verdadero si el botón *indice_boton* del joystick *indice_joystick* se encuentra presionado.

Para obtener la información de los ejes utilizamos la función:

```
Input.GetJoystickAxis(indice_joystick, enum_eje);
```

Aquí, enum_eje es una enumeración definida en *sf::Joy,* que posee los posibles ejes.

A modo de ejemplo, transcribimos los posibles valores de dicha enumeración:

```
enum Axis
     {
       AxisX,
       AxisY,
      AxisZ,
AxisR,
       AxisU,
      AxisV,
       AxisPOV,
       Count // For internal use
     };
```

BIBLIOGRAFÍA

Alonso, Marcelo; Finn, Edward. *Física: Vol. I: Mecánica*, Fondo Educativo Interamericano, 1970.

Altman, Silvia; Comparatone, Claudia; Kurzrok, Liliana. *Matemática/ Funciones*. Buenos Aires, Editorial Longseller, 2002.

Botto, Juan; González, Nélida; Muñoz, Juan C. *Fís Física*. Buenos Aires, Editorial tinta fresca, 2007.

Díaz Lozano, María Elina. *Elementos de Matemática*. Apuntes de matemática para las carreras de Tecnicaturas a distancia de UNL, Santa Fe, 2007.

Gettys, Edward; Séller, Frederick. J.; Skove, Malcolm. *Física Clásica y Moderna*. Madrid, McGraw-Hill Inc., 1991.

Lemarchand, Guillermo; Navas, Claudio; Negroti, Pablo; Rodriguez Usé, Ma. Gabriela; Vásquez, Stella M. *Física Activa*. Buenos Aires, Editorial Puerto de Palos, 2001.

Sears Francis; Zemansky, Mark; Young, Hugh; Freedman, Roger. *Física Universitaria*. Vol. 1, Addison Wesley Longman, 1998.

SFML Documentation, http://www.sfml-dev.org/documentation/

Stroustrup, Bjarne. The Design and Evolution of C++. Addison Wesley, Reading, MA. 1994.

Stroustrup, Bjarne. *TheC++ Programming Language*. Addison Wesley Longman, Reading, MA. 1997. 3° ed.

Wikipedia, http://www.wikipedia.org