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 2
Un mundo de papel

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

2. UN MUNDO DE PAPEL

2.1. Un lugar en el mundo

Lo primero que debemos aprender para crear un mundo virtual es cómo medir y representar unívocamente cantidades y posiciones. De esta forma podremos, por ejemplo, ubicar a nuestro personaje en la pantalla e indicarle que se mueva en determinada dirección con una velocidad específica. Para esto vamos a aprender unas nociones básicas de física y álgebra que son aplicadas constantemente en la programación de videojuegos.

2.1.1. Sistema Internacional de Unidades (SI)

Este sistema se basa en la versión modernizada del sistema métrico.

En Francia, hacia finales del 1700, existía una gran variedad de medidas que obligó a la Academia de Ciencias de París a proponer *la creación de un sistema de medidas decimal perdurable e indestructible* que tomara como base al metro como la unidad de longitud.

Este sistema se llamó *sistema métrico decimal* y permitía que en cualquier momento y lugar, con los medios disponibles para cualquier persona, las medidas se pudieran reproducir con exactitud.

Al pasar los años, este sistema se hizo universal y, en 1960, la Conferencia General de Pesas y Medidas, cuya oficina internacional funciona en Sèvres (área suburbana de París, Francia), lo denominó *Sistema Internacional de Unidades* (SI).

La longitud, la masa y el tiempo son magnitudes físicas fundamentales que describen numerosos objetos y fenómenos. El sistema de unidades usado por los científicos para representar éstas y otras magnitudes se basa en el sistema métrico.

El SI –la versión modernizada del mencionado sistema– incluye magnitudes fundamentales y derivadas, que se describen como unidades fundamentales y derivadas.

Tiene siete magnitudes y unidades fundamentales, que es el menor número para una descripción completa de cualquier cosa que se observe o se mida en la naturaleza.

Asimismo, existen dos unidades suplementarias para la medición de ángulos: el ángulo plano de dos dimensiones y el ángulo sólido de tres dimensiones, pero no ha habido acuerdo para incluirlas como fundamentales o derivadas.

Sin embargo, la mayoría de las magnitudes físicas son *derivadas* porque –como la palabra lo indica– se derivan de otras por medio de ecuaciones matemáticas. Así, la densidad, **ρ**, se deriva de la masa y el volumen (por definición $\rho = m / V$).

En la tabla siguiente se pueden observar las unidades fundamentales, las suplementarias y algunas derivadas que integran el SI:

UNIDADES FUNDAMENTALES		
Magnitud	**Unidad**	**Símbolo**
longitud	metro	m
masa	kilogramo	kg
tiempo	segundo	s
intensidad de corriente eléctrica	ampere	A
temperatura termodinámica	kelvin	K
intensidad luminosa	candela	cd
cantidad de materia	mol	mol

UNIDADES SUPLEMENTARIAS		
Magnitud	**Unidad**	**Símbolo**
ángulo plano	radian	rad
ángulo sólido	estéreo-radian	sr
UNIDADES DERIVADAS		
Magnitud	**Unidad**	**Símbolo**
área	metro cuadrado	m^2
volumen	metro cúbico	m^3
velocidad	metro por segundo	m/s
aceleración	metro por segundo al cuadrado	m/s^2
fuerza	newton	N

2.1.2. Magnitudes escalares y vectoriales

Una *magnitud* es una variable física usada para describir la situación de un sistema particular. Esto significa que se pueda atribuir a la variable un valor numérico y, más precisamente, que se pueda medir. Por ejemplo, cuando expresamos coloquialmente: ¡Qué calor!, está relacionado a la sensibilidad de cada persona. En cambio, si la temperatura ambiente es de 40ºC, todos percibirán lo mismo, independientemente de la sensación que les produzca. Por lo tanto, la temperatura es una magnitud que se puede medir directamente.

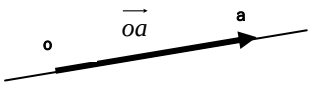
Ahora bien, las magnitudes pueden ser escalares y vectoriales.

Las escalares están totalmente definidas y se indican con un valor numérico y su unidad. Por ejemplo: volumen, peso específico, tiempo, longitud, etc.

Una magnitud vectorial es aquella que se define por un vector y se compone de: punto de aplicación, módulo, dirección y sentido en que actúa. Ejemplos de este tipo de magnitudes son la velocidad, aceleración, fuerza, etc. (no es lo mismo ejercer una fuerza sobre un cuerpo hacia la derecha que hacerlo hacia la izquierda).

Vector
Geométricamente, es un segmento orientado.

Se lo suele indicar con un par de letras ordenadas, que representan origen y extremo con una flechita arriba, $\overrightarrow{oa}$, o también con una letra minúscula con una flechita arriba, $\overrightarrow{v}$. En general, un vector oa con origen en el punto **o** y extremo en el punto *a* posee:



a) un punto de aplicación, que es el origen del vector (punto o);

b) una dirección que está indicada por la recta que lo contiene o cualquier paralela a ella; $\overleftrightarrow{oa}$

c) un sentido, indica hacia a dónde apunta el extremo del vector (cada dirección tiene dos sentidos posibles);

d) un módulo, intensidad o valor. Indica la medida del segmento y se escribe $\left|\overrightarrow{v}\right|$ o $\left|\overrightarrow{oa}\right|$

2.1.3. Sistema de referencia

Para localizar un submarino hay que indicar tres referencias: latitud, longitud, profundidad. Éstos, obviamente, son números que remiten a la posición del submarino. Para la longitud se tiene en cuenta el meridiano de Greenwich, considerando el este o el oeste del mismo. Para la latitud, se considera el Ecuador y se establece la posición al

norte o al sur del mismo, y para la profundidad se utilizan números negativos, ya que se fija como cero el nivel de la superficie.

Lo contrario ocurre con la altura de una montaña. Ésta se expresa con números positivos por considerarse por encima del nivel del mar.

Para determinar la posición de un cuerpo es necesario establecer un sistema de referencia, que puede ser establecido a partir de la posición de otro cuerpo (punto de referencia).

Una forma muy útil y simple de expresar la posición de un cuerpo es mediante un sistema de coordenadas cartesianas, que presenta las siguientes características:

a) los ejes de referencia son perpendiculares entre sí;

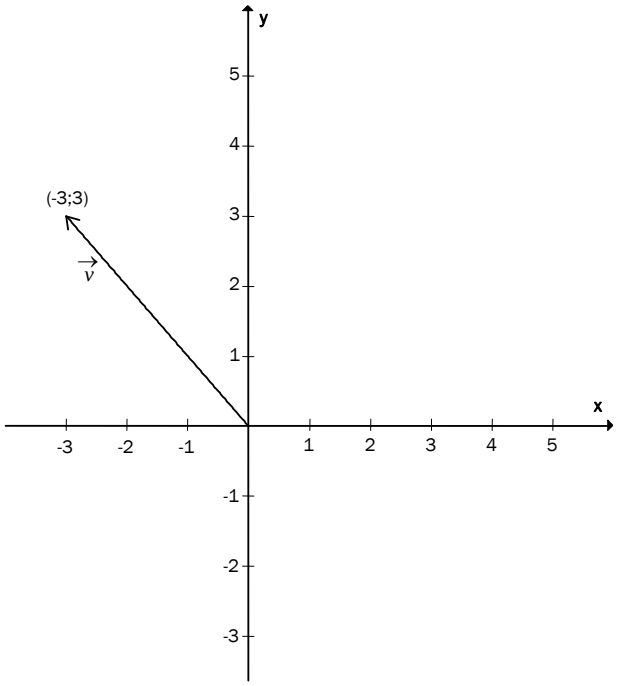
b) puede ser bidimensional (2D), con coordenadas genéricas (x;y) en el plano.

c) puede ser tridimensional (3D), con coordenadas genéricas (x;y;z) en el espacio.

El punto de intersección de los ejes cartesianos se llama *origen de coordenadas* y se indica como (0,0), en el caso de 2D, y (0, 0,0) para el caso de 3D. Por convención, desde el origen de coordenadas hacia la derecha se toma el eje como positivo y, hacia la izquierda, como negativo.

Sistema de coordenadas bidimensional: vector posición

La *posición* es una magnitud vectorial y la misma queda descripta por el vector posición, que comienza en el origen de coordenadas y termina en un punto. Las coordenadas del vector posición son las coordenadas del punto.



Es decir:

$$\vec{v} = (x; y; z)$$

La distancia del punto al origen de coordenadas es el módulo de este vector y se calcula:

$$\vec{v} = (x^2 + y^2 + z^2)^{1/2}$$

Si estamos trabajando en el plano, la coordenada *z* se anula y sólo se trabaja en dos dimensiones en (*x*, *y*), como lo indica la figura.

Observación

La elección de un sistema de referencia surge de considerar un determinado objeto en reposo. Ejemplo: cuando se estudia el movimiento de un avión, se indica la ubicación en cierto instante, pero con respecto a otro objeto que está en reposo, que en nuestro caso es la Tierra. Sin embargo, esto es relativo porque, como sabemos, la tierra no está en reposo, sino que posee un movimiento de rotación.

2.2. Un lugar en nuestro monitor

Así como se utiliza el sistema cartesiano para representar la posición de un objeto en el mundo real, lo mismo sucede en nuestro mundo virtual. En particular, en esta materia trabajaremos con videojuegos 2D, por lo que nuestros sistemas serán sistemas cartesianos en *x* e *y*, como el último presentado en la sección anterior.

2.2.1. Píxeles más, píxeles menos

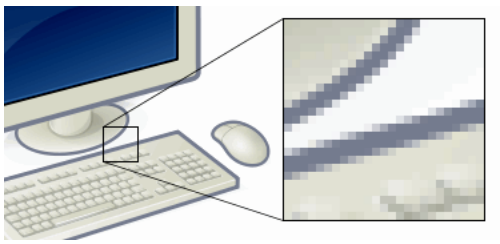
Las imágenes digitales están compuestas de píxeles. Un píxel se puede definir de la siguiente manera:

Píxel

Es la menor unidad homogénea en color que forma parte de una imagen digital, ya sea ésta una fotografía, un fotograma de video o un gráfico.

Dicho de una manera más sencilla, es la menor parte de una imagen que no puede poseer más de un color.

Al conjunto de píxeles podríamos pensarlo como pequeñas baldosas que se ubican una al lado de otra.



Si observamos desde lejos la figura que antecede, varios efectos ópticos del ojo nos hacen ver una imagen de color continuo y no pequeños píxeles, uno al lado del otro. Pero si hacemos un zoom, podemos ver claramente los píxeles que componen esa imagen.

La cantidad de píxeles de una imagen en la dirección *x* e *y* se conoce con el nombre de *resolución de la imagen*.

Por ejemplo, una imagen de 800x600 quiere decir que posee 800 píxeles en la dirección *x* y 600 en *y*. De esta forma, podemos medir la calidad de una imagen: cuanto mayor sea la resolución, menos se pixelará. Es decir, podremos hacerle más zoom antes de que empiecen a distinguirse los píxeles.

Hasta aquí, entonces, hemos utilizado píxeles para referirnos a la mínima parte de una imagen, pero nuestros monitores también están conformados por píxeles.

Los monitores, sin importar la tecnología, se conforman como una matriz de pequeños elementos que pueden representar sólo un color y que, al combinarse, forman las imágenes, tal como ocurre con la imagen digital.

Por lo tanto, también podemos hablar de la resolución que posee nuestro monitor o televisor. La misma está determinada por la disposición física y la tecnología con que fue construido. Por ejemplo, un monitor que soporte una resolución de 1024x768 significa que posee 1024 píxeles en *x* y 768 en *y*.

Las resoluciones de las imágenes y del monitor son independientes entre sí. Esto es, podemos ver imágenes de 800x600 en un monitor que trabaje en una resolución de 640x480. Sin embargo, nosotros veremos la imagen con la resolución del monitor.

Si, en cambio, observamos una imagen de menor resolución en un monitor con mayor resolución, la veremos pixelada debido al menor tamaño que posee.

En la guía práctica retomaremos estos conceptos para comprenderlos mejor.

Por último, es importante mencionar la resolución de la placa de video del sistema sobre el que estemos trabajando.

La placa de video emite la señal a una determinada resolución, que será vista en el monitor. En este caso, la resolución de video debe ser compatible con la del monitor. Esto quiere decir que debe ser igual o menor, ya que hay una correspondencia directa entre la resolución de video y la resolución física del monitor. Si la resolución de video es menor, hay técnicas para adaptarla. En cambio, si la resolución de video es mayor, la imagen no se podrá visualizar.

2.2.2. Todos somos imágenes

En el caso de nuestros videojuegos 2D, siempre trabajaremos con imágenes y con sus distintas resoluciones. Para ello debemos entender mejor, de la manera más sencilla posible, cómo es el proceso de generación de la imagen que vemos en pantalla.

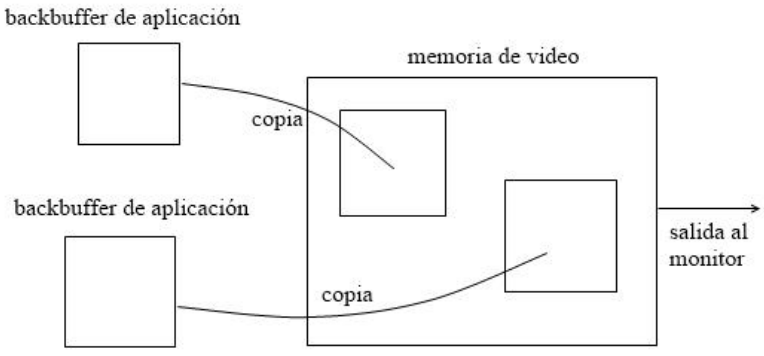
Lo que se muestra en la misma es una región de memoria de nuestra placa de video sobre la cual se dibuja todo el contenido que queremos mostrar. Esta área de memoria, que llamaremos *framebuffer* de video, puede pensarse como una imagen con una resolución determinada.

Todo lo que se muestra en pantalla debe ser dibujado en esta imagen: las ventanas, los botones, el cursor del mouse, etc. La resolución de esta imagen es la que se utiliza para definir la resolución de video y debe ser compatible con la de nuestro monitor, es decir, igual o menor.

Entonces, para poder mostrar nuestro videojuego en pantalla, debemos dibujar el contenido sobre este framebuffer.

Para simplificar esta tarea, cada aplicación posee un framebuffer privado sobre el cual dibuja lo que necesita, y luego, este framebuffer de aplicación se copia sobre el framebuffer de video.

Esto se ilustra en la siguiente figura de manera muy superficial:



Cada framebuffer de aplicación representa una aplicación (ventana). Nosotros vamos a trabajar sobre estos framebuffers de aplicación y dibujaremos todo sobre esta memoria.

El primer parámetro que utilizaremos para crear nuestra aplicación es, justamente, el tamaño de este framebuffer privado, también llamado *resolución de ventana.*

2.2.3. El más feo de los aspectos

Razón de aspecto o *aspect ratio* (en inglés) es un término muy común y utilizado últimamente por la proliferación de LCDs en el mercado. Cuando hablamos de *aspecto* nos referimos a:

Aspect Ratio / Relación de aspecto
Es la proporción entre la anchura y la altura de una imagen o pantalla.

Esta relación normalmente se expresa como *X*:*Y*. La relación de aspecto más común antes de la era de los LCDs era 4:3, que es la utilizada por la normal PAL de televisión.

En cambio, los monitores/televisores denominados *wide* poseen un aspect ratio de 16:9. Es por ello que cuando reproducimos un DVD, que suelen tener una razón 16:9, en un televisor 4:3 vemos dos franjas negras, una arriba y otra abajo, para compensar la diferencia entre los dos aspectos. En las siguientes imágenes se puede observar un ejemplo de una misma imagen vista con distintas razones:

16:9

4:3

Como se puede observar en la imagen 16:9, se posee un mayor ángulo de visión horizontal, pero la escena se ve menos en los extremos superior e inferior, es decir, verticalmente. Por ejemplo, son resoluciones de pantalla 4:3 las siguientes:

- 800x600
- 640x480
- 1024x768

Como ejemplo de resoluciones 16:9, podemos citar:

- 1366x768
- 1280x720– conocido como 720p
- 1920x1080 – conocido como 1080p de la norma de televisión en alta definición.
- 2048x1152

A la hora de crear nuestra ventana de juego vamos a tener que indicar el tamaño de la misma, teniendo en cuenta que un mismo aspecto puede verse bien en algunos televisores, pero mal en otros. En particular, lo que indicamos es el aspecto del framebuffer que introdujimos anteriormente.

2.2.4. Profundidad de color

Más conocido como bits por píxel, la profundidad de color es la cantidad de bits que se utiliza para representar el color de un píxel y esto está directamente ligado a la cantidad de colores que se pueden representar.

Una imagen con más bpp (bits por píxel) nos brinda mayor calidad de imagen, pero ocupa más espacio, ya que necesita guardar más información. Así, una imagen de 8 bpp puede almacenar solamente 256 colores distintos, por lo que es imposible representar, por ejemplo, una foto correctamente.

Las profunidades comunmente usadas hoy en día son de 24 y 32 bits. En la de 24 se utilizan 8 bits por cada canal: rojo, verde y azul. Mezclando estos tres colores se pueden obtener aproximadamente 16 millones de colores distintos, lo cual es suficiente para la mayoría de las aplicaciones debido a las limitaciones del ojo humano para percibir los colores.

En caso de los 32 bits, la cantidad de colores es la misma que con 24, ya que utiliza 8 para cada canal. La diferencia es que los 8 extras que posee son utilizados para *canal alpha*.

Este canal indica el grado de transparencia de la imagen en un determinado píxel. De esta forma, podemos tener imágenes transparentes o con fondo transparente. Esto es fundamental en el desarrollo de videojuegos porque todas las imágenes son cuadradas y, para poder tener personajes que no lo sean, éstos deben dibujarse sobre un fondo transparente. Así, a la hora de dibujar un personaje en pantalla, las áreas de la imagen cuadrada que no se deben dibujar se indican con el canal alpha.

En esta figura se pueden apreciar dos imágenes. La superior derecha posee canal alpha y el fondo se marcó como transparente (por ello que es posible ver el patrón de ajedrez que hay detrás).

Por otro lado, la imagen que se encuentra en la esquina inferior izquierda no posee alpha, por lo que se puede ver el fondo blanco de la misma.

Ambas imágenes son cuadradas, pero la única diferencia es que una tiene alpha y la otra no.

En los videojuegos, todo lo que observamos en pantalla y no se ve cuadrado utiliza esta técnica.

2.2.5. Coordenadas de pantalla

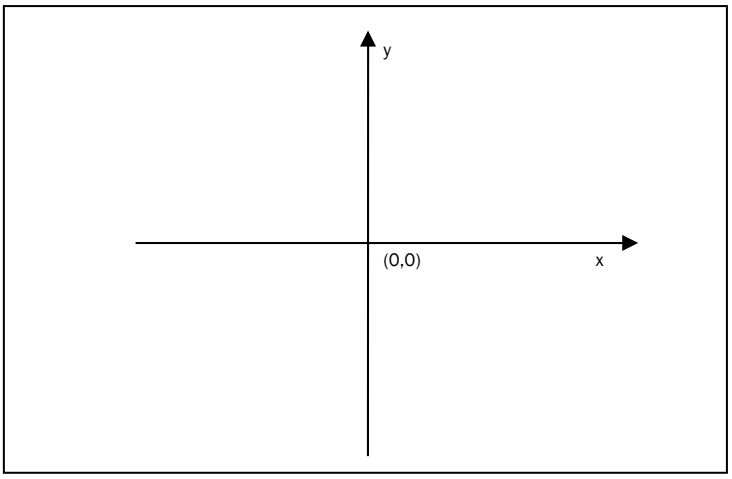
Las coordenadas de la pantalla en un juego en 2D, como veremos en la materia, se especifica utilizando un sistema de referencia cartesiano en *x* e *y*.

Con este sistema ya nos alcanza para posicionar un objeto a lo largo y alto de todo el dispositivo de visualización (monitor, televisor, proyector, etc.).

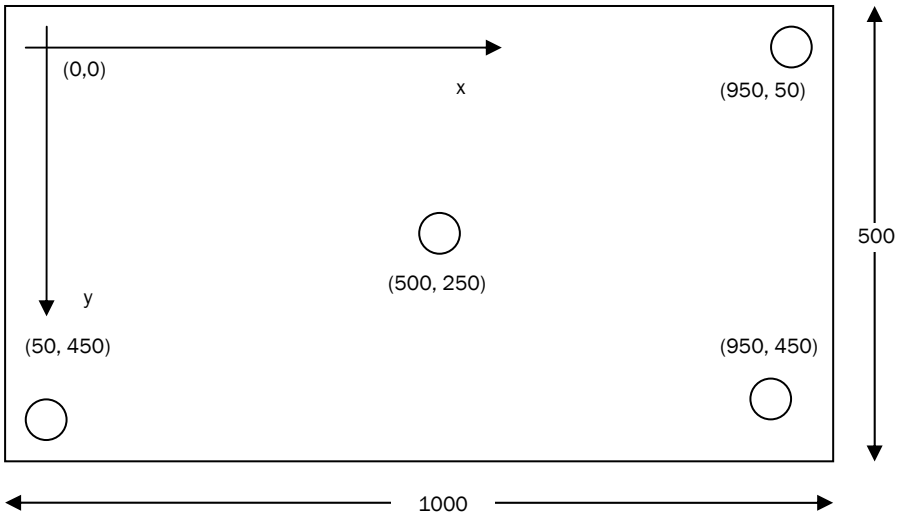
Lo primero que debemos tener en cuenta, como se dijo anteriormente, es cómo ubicamos este sistema cartesiano. Es decir, dónde se posiciona el centro, ya que éste podría estar en diferentes partes y el sistema seguir siendo válido y útil para representar las magnitudes.

La elección del origen del sistema de referencias depende de la biblioteca con la que estemos trabajando. No en todas se encuentra en el mismo lugar, aunque existe una convención que es respetada por la mayoría.

Lo primero que se nos ocurriría seguramente es dibujar los ejes como uno lo suele hacer en papel, es decir, con el origen en el centro de la pantalla/ventana:



Sin embargo, el origen generalmente no se ubica en el centro, sino en la esquina superior izquierda, y la dirección de crecimiento de la coordenada *y* es hacia abajo y no hacia arriba, como en el ejemplo anterior. Es decir:



En este caso, si posicionamos un objeto en el (0,0), éste aparecerá en la esquina superior izquierda de la pantalla. En el dibujo se pueden observar las coordenadas que tendrían algunos objetos colocados en dicho sistema.

Esto, en la práctica, se traduce de la siguiente manera: si queremos que nuestro personaje suba en la pantalla, deberíamos disminuir su posición en *y*.

Esta convención puede parecer un poco antinatural, pero veremos que no es muy difícil acostumbrarse.

2.2.6. Código, código, al fin código

Finalmente, luego de esta breve introducción de los conceptos básicos, vamos a adentrarnos con el código. Para ello debemos crear un proyecto configurado como se describió en la introducción.

El archivo main.cpp es muy sencillo y es listado a continuación. Luego explicaremos cada parte del mismo.

Main.cpp

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


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

	// Creamos la ventana
	sf::RenderWindow App(sf::VideoMode(800, 600, 32),
	 "Que ventana horrible");

	// Loop principal
	while (App.IsOpened())
	{

		// Limpiamos la ventana
		App.Clear();

		//Dibujamos la escena

		// Mostramos la ventana
		App.Display();
	}

	return 0;
}
```

Si copiamos y compilamos este código en nuestro proyecto, el mismo no debería devolvernos ningún error, siempre y cuando todo esté configurado correctamente. De no ser así, hay que rever las configuraciones explicadas con anterioridad.

Una vez compilado el proyecto, podemos ejecutarlo y veremos que nos muestra una ventana completamente negra.

Ahora vamos a observar por parte qué es cada cosa y explicar un poco los fundamentos.

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

En primer lugar y como en todo programa, debemos indicar qué librerías queremos utilizar.

SFML es de diseño modular, lo cual implica que podemos incorporar sólo algunas partes de manera independiente. Por eso, los archivos de encabezado están separados.

En este caso, vamos a utilizar la librería `Window.hpp`, que contiene funcionalidad para crear ventanas y administrarlas. También necesitamos incluir `Graphics.hpp`, que contiene funcionalidad para administrar imágenes y crear el framebuffer. Como veremos a continuación, este proceso es transparente para el programador.

La ventana donde se mostrará nuestro videojuego es lo primero que debe ser creado en toda aplicación SFML.

Todas las clases son miembros del namespace sf, por lo que debemos anteponerlo para la correcta resolución de nombres. La clase que representa una ventana de renderizado es RenderWindow, que posee funcionalidad para crear la ventana/framebuffer, setear sus parámetros, borrar su contenido, dibujarla, mostrarla, etc.

```
// Creamos la ventana
       sf::RenderWindow App(sf::VideoMode(800, 600, 32),
 "Que ventana horrible");
```

El constructor recibe dos parámetros. El primero es un VideoMode, que es un objeto que describe un modo de video y provee funciones para determinar qué modos soporta el dispositivo actual.

Modo de Video
Descripción del ancho, alto, profundidad de color y frecuencia de un dispositivo de visualización.

Cuando decimos *frecuencia* nos referimos a la cantidad de veces que el hardware muestra una imagen por segundo. Este valor debe estar por arriba de los 60hz, para que el usuario no note parpadeos incómodos. Hay mucho para decir sobre este tema, pero lo pospondremos para más adelante.

En este caso, al objeto VideoMode lo estamos inicializando con una resolución de 800x600 y una profunidad de color de 32 bpp.

 ¿Qué sucede si cambiamos los parámetros del constructor? ¿Observamos algún cambio?

El segundo parámetro que recibe el constructor es el título que figurará en la ventana que creamos. Luego de crearla, nos introduciremos en el *loop* o bucle principal del programa.

```
       while (App.IsOpened())
       {

              // Limpiamos la ventana
              App.Clear();

              //Dibujamos la escena

              // Mostramos la ventana
              App.Display();
       }
```

La idea de este bucle es que, mientras la aplicación se encuentre abierta, se ejecuten el dibujo y la lógica.

En este caso, `App.IsOpened()` es un método de la ventana que devuelve *true* o *false*, dependiendo de si la misma fue o no cerrada. Si la ventana está abierta, entonces hay que dibujar su contenido. La primera línea del loop `App.Clear()` borra el contenido del framebuffer y lo deja listo para empezar a dibujar sobre él.

Antes de salir del loop debemos indicar a la ventana que ya terminamos de dibujar la escena para que pueda procesarla y mostrarla en pantalla. Esto es lo que realiza `App.Display()`.

Seguramente han notado que la aplicación que se abre no puede cerrarse haciendo click en el botón correspondiente. Sólo puede lograrse desde el IDE (Integrated Development Environment), mediante un click en el botón de detener la ejecución. Esto se debe a que nuestra aplicación no está *escuchando* eventos.

No obstante, retomaremos este tema en las próximas unidades, ya que momentaneamente nos apegaremos a la estructura más sencilla posible.

Este loop que hemos visto se repite a lo largo de las librerías y las distintas plataformas. Por lo tanto, lo que aprenderemos es aplicable a otras librerías, además de SFML, ya que –desde el punto de vista lógico– estos pasos siempre deben estar.

2.2.7. ¿Uno, dos... tres... qué?

En el parágrafo anterior hablamos de posicionamiento, pero en ningún momento dijimos en qué unidades está expresada dicha posición.

En el caso de nuestros juegos en 2D, estas cordenadas de posición se deben encontrar dentro de los límites de nuestro framebuffer o ventana. Por ejemplo, si nuestra ventana es de 300x200, nuestras coordenadas en x debe cumplir $0 <= x < 300$, y en y $0 <= y < 200$.

Dentro de este rango, la variación no se da con saltos enteros, sino que podemos indicar posiciones con números decimales. Esto quiere decir que una imagen que dibujamos en nuestra ventana no tiene que coincidir necesariamente con los píxeles del frambuffer.

Imaginemos que estamos haciendo una nueva versión del clásico *pong*, donde una pequeña pelota rebota contra las paletas que se encuentran en los extremos horizontales de la pantalla, es decir, una se encuentra en x cercano a 0 y la otra en x cercano a 300, si utilizamos la resolución citada anteriormente. La pelota recorre la pantalla y rebota contra estas paletas.

Lo que nos interesa en este momento es cómo se posiciona la pelota en la pantalla. Para simplifcar vamos a considerar que sólo se desplaza en x con y constante (50 por ejemplo), es decir, horizontalmente.

En un principio, la pelota se ubica en la coordenada (10,50) y queremos moverla hacia la derecha. Entonces, una pregunta válida es: ¿Cuánto es lo mínimo que puedo desplazarla hacia la derecha?

Aquí, otra vez, lo más natural sería pensar que lo mínimo que puedo desplazar la pelota es un píxel, debido a la naturaleza discreta de las imágenes. Sin embargo, veremos que alguna librerías, como SFML, no dibuja directamente las imágenes –la pelota en nuestro caso– sobre el framebuffer, sino que son sometidas a una serie de pasos previos.

Esto nos permite indicarle posiciones que no son enteras, esto es, que se mueva menos de un píxel a la derecha. El resultado de esto depende de la librería y de esos *pasos previos*, pero no ahondaremos por ahora en este tema. Lo importante es que podemos indicarle las posiciones con números flotantes dentro del rango descrito anteriormente.

Como vimos hasta aquí, siempre hablamos de cantidades, pero no de unidades.

Si bien hay una correspondencia entre posición y píxeles, esto sólo se da en casos particulares, por ejemplo, cuando la pantalla de juego no se desplaza y no se realizan acercamientos. Pero si esto sucede, ya no resulta válido dicho mapeo, por lo que –para ser más generales– no lo utilizaremos.

Dentro de nuestro juego, las unidades son adimensionales, esto es, pueden ser las que nosotros deseemos. Por ejemplo, si el objeto se encuentra en x=10, podemos decir que x se encuentra a 10 cm o a 10 m del origen. En los dispositivos gráficos no existen unidades, sino sólo las que nosotros deseemos utilizar.

En los cálculos físicos y modelos es importante que, una vez elegido un conjunto de unidades para las magnitudes básicas, éstas sean mantenidas y siempre se trabaje con las mismas, a fin de mantener coherencia física. Por eso normalmente se utilizan las unidades del SI. Es decir, generalmente se piensa que en la pantalla nuestro mundo figura en metros, que al tiempo lo medimos en segundos y a las fuerzas, en Newtons. Las magnitudes que no se encuentren en estas unidades deberán ser convertidas antes de ser utilizadas.

En las actividades se explorará un poco más este apartado.

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