UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Modelos y algoritmos para videojuegos I

Unidad 5

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

5. UN POCO DE PEGAMENTO

5.1 La primera aplicación orientada a objetos

Hasta el momento hemos trabajado con ejemplos y ejercicios donde no hicimos mucho hincapié en la orientación a objetos. En esta unidad veremos una forma de estructurar nuestra aplicación, utilizando los conceptos de orientación a objetos que dimos anteriormente. También introduciremos algunas nociones claves para el desarrollo de videojuegos.

La estructura que se presentará responde a fines didácticos y no pretende ser la mejor ni la más eficiente, sino un punto de partida sobre el cual puedan construir, corregir y criticar.

Ahora sí, entonces, entremos de lleno a nuestro esqueleto de aplicación.

5.2. El esqueleto en el armario

En primer lugar, vamos a definir una clase Game, que será la de mayor jerarquía y se encargará de llevar adelante la actualización de todos los demás componentes que intervengan en nuestra aplicación.

Pero antes debemos aprender un par de cosas más... Comencemos con el manejo del tiempo.

5.2.1. Conviviendo con el tiempo

En todo videojuego es muy importante tener algún medio para medir el tiempo transcurrido entre cada loop de actualización del estado del juego. Esto nos sirve principalmente para dos cosas: medir la tasa de cuadros por segundo (framerate) y asegurarnos de que nuestro videojuego se ejecute a la misma velocidad en todas las máquinas.

El framerate es un número que nos da una idea del rendimiento de nuestro videojuego. Generalmente, en los videojuegos se apunta a tener un framerate de 60 cuadros por segundo (60 fps), aunque en muchos casos hay que conformarse con 30 debido a limitaciones técnicas. Justamente, los fps son una forma de medir la cantidad de recursos que nos consume nuestra aplicación o, dicho de otra forma, qué tan lento es nuestro videojuego.

Para medir los fps necesitamos conocer cuanto tiempo se demora nuestra aplicación en dibujar un frame y luego, en base a esto, estimamos cuántos frames podemos dibujar en un segundo.

Así, sea T_f el tiempo que demora en dibujar un frame, entonces podemos calcular los fps como:

$$fps = \frac{1}{T_f}$$

Es decir, dividimos 1 segundo por el tiempo de frame.

Por otro lado, dijimos que el tiempo es también importante para regular la velocidad del videojuego.

Supongamos que tenemos nuestro loop principal de aplicación donde en cada vuelta aumentamos la posición del sprite en uno:

Main.cpp

```
////////////////////////////////////////////////////////////
/// Variables
////////////////////////////////////////////////////////////
sf::Sprite cursor;
sf::Image mat_cursor;


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


       sf::Event Event;
       mat_cursor.LoadFromFile("cursor.png");
       cursor.SetImage(mat_cursor);
       cursor.SetPosition(0,0);

       int x=0;
    //Creamos la ventana de la aplicacion
      sf::RenderWindow App(sf::VideoMode(800, 600, 32),
      Input");
      App.ShowMouseCursor(false);
      while (App.IsOpened())
       {

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

              x=x+1;
              cursor.SetPosition(x,x);

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

       return EXIT_SUCCESS;
}
```

Esta aplicación provoca que el sprite llamado cursor se mueva en diagonal por la pantalla. La velocidad con que se mueve depende de la rapidez de la PC para ejecutar el loop de rendering. Así, cuanto más rápido lo haga, más veloz será el movimiento. Para solucionar esto, se utiliza el tiempo transcurrido desde el último frame. Es decir, se multiplica el incremento del desplazamiento por el tiempo transcurrido.

Por lo tanto, si Tf es el tiempo desde el último frame, el código quedaría de la siguiente manera:

main.cpp

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

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

              x=x+1*Tf;
              cursor.SetPosition(x,x);
...
```

Aquí podemos ver que se multiplica el incremento de la posición (1) por el tiempo transcurrido.

De esta manera, si la máquina es rápida, lo cual permitirá realizar más ciclos en un segundo, cada desplazamiento será menor que en una máquina lenta, donde habrá menos ciclos, pero donde cada desplazamiento será mayor debido a que Tf será mayor. En ambas máquinas, el incremento será de una unidad cuando Tf sea de un segundo. Es decir, nuestro objeto tendrá una velocidad de un píxel por segundo.

Ahora que ya conocemos la importancia del tiempo en nuestro videojuego, veamos cómo hacemos con SFML para poder medirlo.

Obtener el tiempo desde el último frame se puede hacer de dos formas.

La primera consiste utilizar el método GetFrameTime() de la clase RenderWindow:

main.cpp

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

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

              Tf=App.GetFrameTime();
              x=x+1*Tf;
              cursor.SetPosition(x,x);
...
```

La segunda alternativa consiste en utilizar la clase Clock, que posee dos métodos Reset() para poner a cero el reloj, y GetElapsedTime(), que nos devuelve la cantidad de tiempo que transcurrió desde la última vez que llamamos a Reset(). Ahora, el mismo ejemplo anterior nos quedaría así:

main.cpp

```
Clock reloj;

       while (App.IsOpened())
       {

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

              Tf=reloj.GetElapsedTime();
              reloj.Reset();

              x=x+1*Tf;
              cursor.SetPosition(x,x);
...
```

En este caso, cada vuelta leemos el tiempo transcurrido y reseteamos el reloj para la próxima medición.

Otra función muy útil para manejar el tiempo es SetFrameRateLimit(int). La misma limita automáticamente la cantidad de cuadros que se dibujan por segundo. De esta manera, no debemos preocuparnos si nuestro videojuego se ejecuta más rápidamente en una máquina que en otra. Por ejemplo, si queremos que se mantenga la tasa de cuadros por segundo en 60, hacemos:

```
App.SetFramerateLimit(60);
```

Si pasamos un 0 como argumento, el framerate no se limita y la aplicación se ejecuta tan rápido como sea posible.

5.3. Unas líneas de texto, si es tan amable

En las aplicaciones de consola resulta sencillo generar salidas de tipo texto debido a la naturaleza de la consola. Sin embargo, no es tan trivial realizarlo en una ventana gráfica, ya que todo lo que podemos dibujar allí son imágenes. En estos casos, por lo tanto, el texto debe pasar por un proceso previo para convertirse en una imagen, y SFML nos provee facilidades para lograr dicha tarea.

Lo primero que debemos hacer en SFML es definir la fuente que utilizaremos para dibujar en pantalla. Para ello emplearemos el objeto Font, que provee el método para cargar una fuente de diferentes orígenes (el más común permite cargarla desde disco).

La función LoadFromFile() recibe como argumento el nombre del archivo de fuente que deseamos cargar y devuelve un boolean indicando si la cargó con éxito o no.

A continuación, veamos un ejemplo:

```
sf::Font MyFont;

if (!MyFont.LoadFromFile("arial.ttf"))
{
    // Error si no se pudo cargar la fuente
}
```

Luego, ya podemos definir objetos del tipo sf::String (no confundir con std::string), que son los que finalmente serán dibujados en pantalla.

Dichos objetos poseen propiedades para configurar la posición, tamaño, color, etc. del texto.

Por ejemplo:

```
sf::String Text;

Text.SetText("Bazzinga");
Text.SetFont(MyFont);
Text.SetSize(50);

Text.SetColor(sf::Color(128, 128, 0));
Text.SetRotation(90.f);
Text.SetScale(2.f, 2.f);
Text.Move(100.f, 200.f);
```

Las tres primeras propiedades (el texto a mostrar, la fuente a utilizar y el tamaño) son obligatorias para cualquier String que se cree. Luego, para dibujar dicho String, simplemente llamamos el método Draw de RenderWindow:

```
sf::RenderWindow App;
```

Para mayor información sobre todas las propiedades disponibles de la clase String, los invitamos a leer la documentación oficial, donde podrán encontrar con más detalles todas las posibilidades que ofrece.

5.4. El esqueleto al fin

En este parágrafo vamos a proponer un esqueleto de aplicación y explicar un poco sus partes. Luego, les quedará como actividad completar la tarea, aplicando todo lo que hemos visto hasta ahora. También podrán utilizar esta clase como base del proyecto final de la materia.

La clase principal es la Game, que contiene métodos y propiedades para inicializar el juego, actualizarlo y dibujarlo. La declaración de la misma es:

Game.h

```
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include <string>
using namespace sf;
using namespace std;
class Game
{
private:
	RenderWindow *pWnd;
	void ProcessEvent(Event &evt);
	void DrawGame();
	void UpdateGame();
	void ProcessCollisions();
public:
	Game(int alto, int ancho, string titulo);
	~Game(void);

	void Go();
};
```

Como vemos, contiene como propiedad el RenderWindow, que utilizaremos para dibujar, y los siguientes métodos privados:

a) *ProcessEvent()* Este método recibe un evento y se encarga de procesarlo.

b) *DrawGame()* Este método se encarga de dibujar todos los elementos que intervienen en nuestro videojuego.

c) *UpdateGame()* Este método se encarga de actualizar las posiciones, estados, etc. de los elementos de nuestro videojuego.

d) *ProcessCollisions()* En este método debemos chequear las colisiones que se produjeron y reaccionar acorde a las mismas.

Además, posee un constructor donde recibimos el ancho, alto y título de la venta de juego.

Por último, el loop principal de la aplicación se encuentra en el método Go que veremos a continuación:

```
void Game::Go(){

	//objeto para recibir eventos
	Event evt;

	while(pWnd->IsOpened()){

		//procesar eventos
		while (pWnd->GetEvent(evt))
			ProcessEvent(evt);

		//procesar colisiones
		ProcessCollisions();

		//actualizar estados de objetos
		UpdateGame();

		pWnd->Clear();
		DrawGame();
		pWnd->Display();

	}
}
```

En este método se puede observar el tradicional loop del juego y las llamadas a los métodos propuestos anteriormente, que se encargan de las distintas tareas también descriptas.

En general, la mayoría de los videojuegos se estructuran de esta forma, obviamente con variantes y soluciones adaptadas a cada género.

Desde luego, no es obligatorio respetar esta estructura. La intención fue mostrarles una posible implementación de un videojuego orientado a objetos para que podamos ver cómo se diferencia con el loop que realizábamos antes en el mismo main.

Ahora, entonces, para ejecutar nuestro videojuego, deberíamos hacer lo siguiente:

```
#include "Game.h"

void main(int argc, char* argv[])
{

	Game juego;
	juego.Go()
	return;
}
```

Esto invocaría el método que se encarga de manejar la aplicación.

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