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 3
¿Qué hago acá?

Docentes
Sebastián Rojas Fredini
Patricia Schapschuk

CONTENIDOS

3. ¿QUÉ HAGO ACÁ? 2

3.1. (Sprite != Imagen) && (Material==Imagen) 2

3.1.1. Materiales + Imágenes + Sprites=? 2

3.1.2. De vuelta al mundo real 4

3.1.3. El mundo real a escala 6

BIBLIOGRAFÍA 11

3. ¿QUÉ HAGO ACÁ?

3.1. (Sprite != Imagen) && (Material==Imagen)

Una simple ventana no alcanza para el objetivo que tenemos en mente, pero es una pieza fundamental para su consecución.

Seguramente, a esta altura de la materia, estén ansiosos por poner *ya* un personaje en la pantalla.

Entonces, en esta unidad, haremos un alto en los conceptos físicos y nos introduciremos en el mundo de los sprites, imágenes y materiales, y veremos cómo posicionarlos en nuestra ventana.

No obstante, para ello, primero necesitamos conocer algunos conceptos y diferenciarlos correctamente.

Ahora, sin más parafernalia, prosigamos con lo más interesante: definamos qué es una imágen y qué es un sprite.

3.1.1. Materiales + Imágenes + Sprites=?

Si bien a simple vista pueden parecer la misma cosa, es muy importante dominar estos tres conceptos y, principalmente, reconocer en qué difieren.

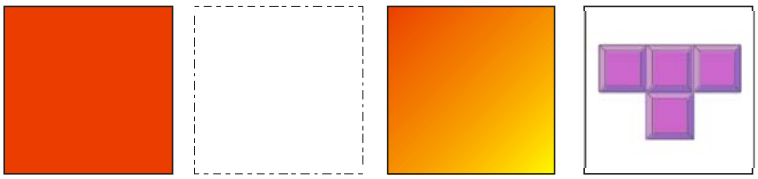
Las definiciones que consideraremos aquí son las más utilizadas actualmente en los motores gráficos, aunque a veces se les da distinto significado de acuerdo al contexto. Nosotros, en cambio, hablaremos de esto de la forma más general posible y luego veremos cómo se traducen los conceptos en la práctica.

Arranquemos con los materiales. ¿En qué pensamos cuando decimos *materiales*? Podríamos pensar en la materia prima de algo. Por ejemplo, si vamos a construir una casa, si la haremos con cemento y ladrillos, o con madera. En ambos casos, tendremos una casa, pero se verán radicalmente distintas, ya que los materiales utilizados –ladrillo y madera– son diferentes. También podemos tener dos casas de madera, pero pintadas de distinto color, es decir, el mismo objeto en ambos casos, pero con uno de sus materiales –la pintura– diferente. Como vemos, estos materiales le dan el aspecto y la forma a la casa.

En videojuegos, material se suele llamar a la descripción de cómo se dibuja en pantalla una superficie, que puede ser en 3D o 2D; nosotros nos concentraremos en esta última.

El material determina cómo se ve el objeto en pantalla. Por ejemplo, si queremos dibujar un rectángulo en pantalla, lo primero que debemos preguntarnos es cómo deseamos que se vea. ¿Que sea de un color sólido? ¿Que el borde sea punteado? ¿Que la pintura sea en degradé? ¿Que en la zona del rectángulo se dibuje una imagen que tenemos guardada?...

En la siguiente figura se muestran algunos ejemplos de esto.



El objeto es, en todos los casos, un cuadrado, pero con distintos materiales. Notemos, en especial, el último de ellos donde, en lugar de rellenarse con un color determinado, se le dibujó encima una imagen de un bloque del popular videojuego *Tetris*®. En este caso, el material indica directamente qué imagen es la que se debe dibujar sobre el rectángulo.

Esto nos lleva al segundo concepto que buscamos aclarar: imagen, que puede definirse de la siguiente manera:

Imagen
Un arreglo de píxeles en dos dimensiones. Es decir que en informática la definiremos como una matriz de píxeles.

Una imagen es una de las partes que compone un material, pero éste abarca mucho más e inclusive podría o no contener imágenes.

De esta forma, entonces, hemos establecido la relación entre imágenes y materiales. En este sentido cabe destacar que, si no tuviésemos un material, no veríamos directamente lo que intentamos dibujar, ya que su apariencia visual no estaría determinada.

Perfecto. Hasta aquí tenemos materiales, pero no hemos aclarado sobre qué van aplicados los mismos. Es decir, ¿a qué objeto los aplicamos? Por ejemplo, el cuadrado del ejemplo anterior… ¿Qué es? ¿A qué cosas podemos aplicarles materiales?

Las respuestas a estas preguntas dependen en gran medida de si estamos trabajando en 2D o 3D. Como siempre, nosotros nos vamos a concentrar en el mundo de papel, así que vamos a ver a qué objeto le aplicamos los materiales allí.

En 2D, los materiales se aplican sobre superficies, esto es, porciones rectangulares de nuestra pantalla que luego son dibujadas en la misma. Estas superficies se denominan *sprites*.

Un *sprite* es un objeto que posee representación gráfica dada por el material, pero además tiene otras propiedades como posición, rotación, etc. Cuando programamos nuestro videojuego trabajamos con sprites y modificamos sus propiedades para darles movimiento y animación.

Sprite
Superficie con determinadas propiedades para su visualización y comportamiento, que es integrada a una escena.

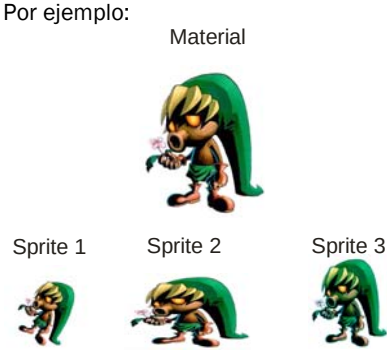
Esta definición difiere un poco de la clásica, según la cual un sprite es una imagen que se utiliza para componer una escena con varias imágenes. Esto supone que un sprite es una imagen y viceversa. Es decir, que ambos son términos intercambiables. Sin embargo, la evolución del hardware, software y pensamiento hicieron necesario dividir y separar estos conceptos, de la forma en que fueron presentados en esta unidad.

En los albores de los videojuegos, una imagen estática, que sólo podía ser desplazada, alcanzaba para representar un personaje y un objeto. Es más, era la única forma de hacerlo. Con el paso del tiempo, una imagen ya no bastó, sino que fue necesario animarla, escalarla, rotarla, etc.

Varios años más tarde, se desarrolló la capacidad de poder generar efectos sobre esa imagen, cambiarle el tono, modularla con el fondo, etc. Todas estas acciones adicionales ya no podían ser descriptas simplemente por la imagen, que conformaba un sprite en esos tiempos, sino que eran necesarias algunas estructuras extras para representarlas.

Es aquí donde cobra sentido la división entre sprite, imagen y material. Mientras que la imagen era simplemente eso, una imagen, particularmente las propiedades de posición, tamaño, rotación y animación fueron a parar al objeto sprite, y todo lo referido a la visualización terminó en lo que conocemos como material. Esto permitió, por ejemplo, que dos sprites, con la misma imagen asociada, se dibujen de diferentes maneras.

Por ejemplo:



En esta imagen podemos observar a Link, de *Zelda*; *A Link to the past*, convertido en Deku. En este caso, tenemos 3 sprites que comparten el mismo material. Sin embargo, cada sprite se dibuja dependiendo de la rotación (Sprite 1), tamaño (Sprite 2) y modulación tonal (Sprite 3).

Si nos permitimos dejar la formalidad un poco de lado, podemos ver a un sprite como la representación de una imagen dentro de nuestro videojuego.

Ahora ya podemos definir más formalmente que es un material:

Material
Descripción de cómo se ha de dibujar la superficie de un sprite.

Es importante recordar que, como en el ejemplo anterior, un material puede ser aplicado a muchos sprites. Y en memoria sólo existe una instancia del material y no una por cada sprite.

Por último, cerrando la pregunta del título de la unidad. ¿Qué tenemos con todo esto combinado? Nada más y nada menos que un personaje en pantalla.

3.1.2. De vuelta al mundo real

La división que presentamos en la sección anterior no siempre es respetada por los motores gráficos. Generalmente se rompen las barreras entre material e imagen, o imagen y sprite. Sin embargo, la generalidad con la que tratamos estos conceptos nos va a permitir adaptarnos a cualquier invento maquiavélico de la tecnología.

SFML, en particular, no posee el concepto de material, sino que funde material e imagen en un solo objeto. Es decir, tenemos sprites por un lado, e imágenes por el otro, siendo la imagen la descripción de cómo debe verse un sprite. Vamos a ver un ejemplo sencillo de cómo dibujar nuestro primer sprite:

Main.cpp

```
      //Cargamos la imagen del archivo
      image.LoadFromFile("..\\img\\un-spacy.jpg");

      //Configuramos el material del sprite
      sprite.SetImage(image);

      //Seteamos la posicion
      //sprite.SetY(50.4f);

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

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

            App.Clear();
            //Dibujamos el sprite
            App.Draw(sprite);

 App.Display();
 }

 return 0;
}
```

Las librerías que utilizamos son las mismas que empleamos para la unidad anterior, por lo que no vemos cambios en esa sección. Por el contrario, en la declaración de variables, observamos la aparición de dos elementos nuevos: una variable de tipo Sprite y otra de tipo Image.

```
////////////////////////////////////////////////////////////
/// Variables
////////////////////////////////////////////////////////////
sf::Image image;
sf::Sprite sprite;
```

Image hace las veces de material, como decíamos anteriormente; tiene funcionalidad para cargar imágenes de diferentes formatos (*.png, *.jpg, etc.) y almacenarlas, entre otras cosas.

Sprite, en cambio, sí responde a la descripción de la sección anterior y nos sirve como superficie para asignarle el material.

Una vez dentro del punto de entrada de la aplicación *main,* lo primero que debemos hacer es cargar la imagen del dispositivo de almacenamiento:

```
//Cargamos la imagen del archivo
	image.LoadFromFile("..\\img\\un-spacy.jpg");

//Configuramos el material del sprite
	sprite.SetImage(image);
```

Esto es lo que hace la primera línea, su argumento es justamente el *path* donde reside dicho archivo.

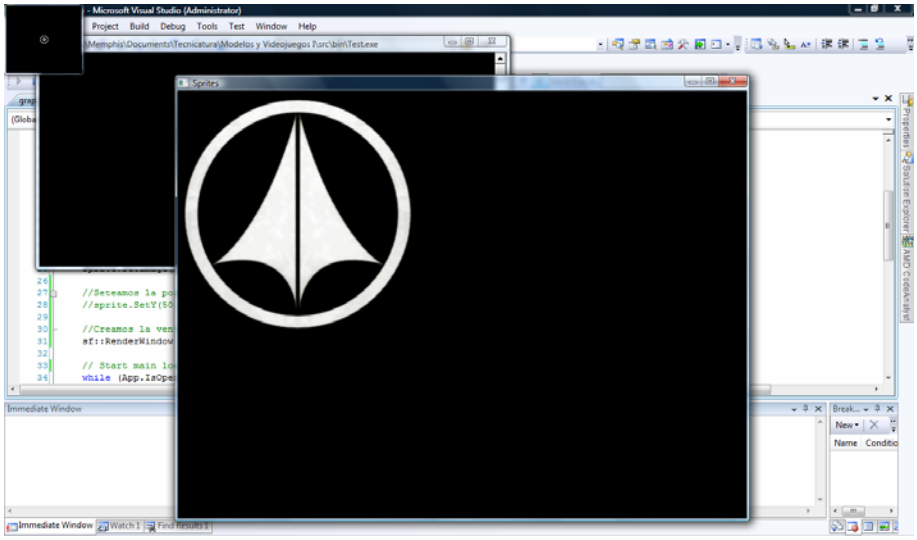
Luego de cargar la imagen, ya podemos asignarla a nuestro sprite para ser dibujado en pantalla. Para ello, utilizamos el método *SetImage,* como en el ejemplo. Desde este punto en adelante, generalmente se trabaja sobre el objeto Sprite y no sobre el Image, ya que Sprite es el que define dónde se posiciona, su rotación, etc.

Luego, dentro del loop principal del programa que vimos anteriormente, debemos indicar que dibuje el sprite. Esto lo hacemos luego de haber llamado *Clear* para limpiar la pantalla y antes de llamar *Display* que, como recordarán, se encarga de mostrar lo que dibujamos en el *backbuffer*.

```
	App.Clear();
		//Dibujamos el sprite
		App.Draw(sprite);

		App.Display();
```

Para hacerlo, utilizamos el método *Draw* del objeto RenderWindow, que recibe como parámetro el sprite a dibujar.

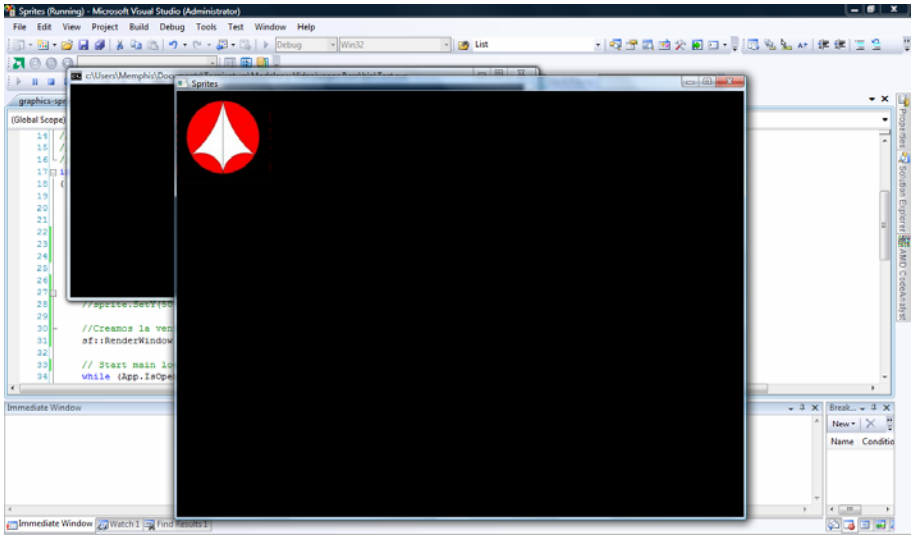
Si ejecutamos el programa, veremos algo similar a lo siguiente:



Aquí observamos nuestro sprite sobre el fondo negro, mostrando el logo de la *UN Spacy,* de la popular serie Macross®, en blanco y negro.

Ahora veremos qué sucede si le cambiamos el material, pero sin alterar el sprite. Para ello, vamos a modificar la línea donde seteamos la imagen un-spacy.jpg por un-spacy2.gif, que es una versión en color del mismo logo.

Si compilamos y ejecutamos, observaremos el cambio que hicimos:



Como vemos, el logo aparece en color, pero es más pequeño.

Ahora bien, ¿por qué sucede esto?

En primer lugar, se podría argumentar correctamente que las imágenes un-spacy.png y un-spacy2.png poseen distinto tamaño, por lo que la última se ve más pequeña. Sin embargo, lo que determina el tamaño de lo que estamos viendo en pantalla no es la imagen, que es simplemente el material, sino el sprite.

Entonces, ¿por qué los logos se ven de distinto tamaño?

La respuesta es sencilla: porque en ningún momento definimos el tamaño del sprite. Por lo tanto, en estos casos, el sprite asume que su tamaño será igual al de la imagen que tiene asignada.

En el próximo parágrafo veremos cómo podemos hacer para que los dos logos tengan el mismo tamaño.

3.1.3. El mundo real a escala

Para poder comparar el tamaño de los sprites, vamos a dibujar ambos sprites en una sola ventana. Para ello, debemos agregar un nuevo objeto sprite y un nuevo objeto imagen y configurarlo de la misma manera que hicimos con el primero. En este código se cambia el nombre a las variables para identificar con mayor claridad cuál es color y cuál es blanco y negro.

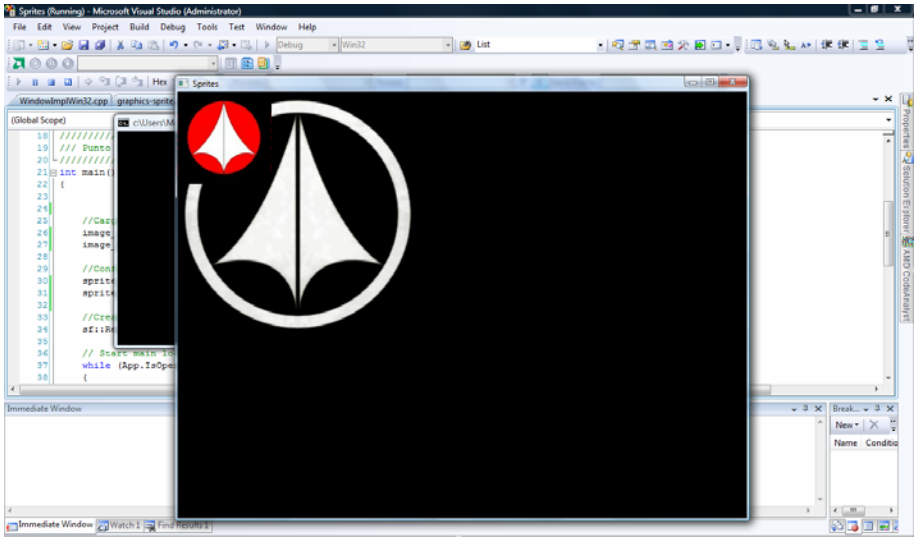
```
////////////////////////////////////////////////////////////
/// Variables
////////////////////////////////////////////////////////////
sf::Image image_color;
sf::Image image_bw;

sf::Sprite sprite_color;
sf::Sprite sprite_bw;
////////////////////////////////////////////////////////////
/// Punto de entrada a la aplicación
```

```
////////////////////////////////////////////////////////////
int main()
{
       //Cargamos las imagenes del archivo
       image_color.LoadFromFile("..\\img\\un-spacy2.jpg");
       image_bw.LoadFromFile("..\\img\\un-spacy.jpg");

       //Configuramos los materiales color y blanco y negro
       sprite_color.SetImage(image_color);
       sprite_bw.SetImage(image_bw);

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

 while (App.IsOpened())
 {
             App.Clear();

             App.Draw(sprite_bw);
             App.Draw(sprite_color);

        App.Display();
 }

 return EXIT_SUCCESS;
}
```

Como pueden observar, el código es idéntico al anterior pero con una variable más.

Hasta aquí no hemos introducido nada nuevo. Si ejecutamos este código, ¿qué veremos?

La salida debería ser similar a ésta:



Aquí se pueden apreciar las dos imágenes, una sobre la otra. El orden en que aparecen solapadas depende del orden en que se llama el método Draw del RenderWindow.

¿Esto es realmente así? ¿Qué sucede si cambiás el orden de dibujo?

Ya tenemos ambas imágenes en la pantalla, pero aún no hemos terminado porque, para compararlas visualmente, no nos sirve que estén encimadas, Esto, en el caso de que tengan el mismo tamaño, provocaría que sólo veamos una.

Para solucionar este problema hay que cambiar la posición de una de ellas. La posición se configura con el método *SetPosition(),* que recibe como argumentos la nueva posición en *x* e *y* para el sprite. Dicha posición es relativa al vértice superior izquierdo

de la imagen. En nuestro caso, con moverlo un poco a la derecha nos alcanza, por lo que agregando la siguiente línea solucionamos nuestro problema:

```
sprite_color.SetPosition(350,0);
```

Esto indica que el sprite está en 350 en la dirección *x,* y 0 en la dirección *y*.

Ahora ya estamos en condiciones de ver cómo hacemos para que tengan el mismo tamaño.

En SFML uno no puede setearle directamente el tamaño en píxeles al sprite, sino que debe setear un factor de escala para la imagen. Es decir, el tamaño del sprite es siempre proporcional al de la imagen.

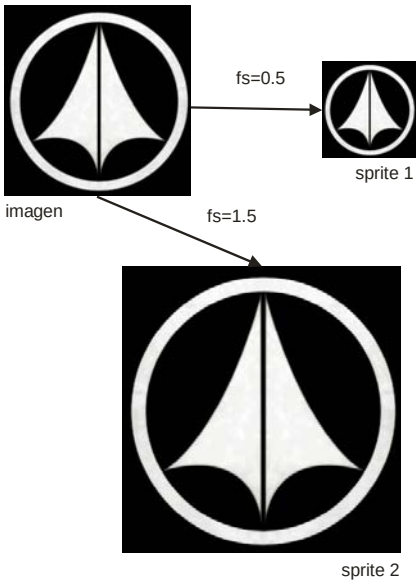
Por ejemplo, si nuestra imagen es de 256x256 y el factor de escala del sprite es de 1.0, el tamaño del sprite será de 256x256. En cambio, si el factor de escala fuese 2, nuestro sprite tendría un tamaño de 512x512. Este factor de escala varía desde 0 al número flotante máximo que podemos tener en el sistema. Si usamos una escala menor a 1, entonces estamos achicando la imagen.

Pasando en limpio, el tamaño del sprite en cada dimensión (*x* e *y*) es:

Tamaño Sprite
Ts = fs * Ti

donde Ts es el tamaño del sprite, fs es el factor de escala y Ti es el tamaño de la imagen.

Veamos unos ejemplos gráficos donde variamos el factor de escala y vemos el tamaño de sprite resultante:



Los factores de escala pueden ser los mismos o distintos para *x* e *y*.

En el caso de este ejemplo, se tomó igual la escala para que las imágenes no pierdan su relación de aspecto.

Si queremos que los dos sprites tengan el mismo tamaño, esta forma de trabajar nos obliga a calcular, mediante una división entre los tamaños en cada dimensión de las imágenes, cuál es el factor de escala que las relaciona a las mismas.

Por ejemplo, dadas dos imágenes a las cuales llamaremos I_1 e I_2, vamos a ver cuál es el factor por el que debemos multiplicar I_1 para que sea igual a I_2. Dicho factor se denomina $f_{1\to 2}$ y se define de la siguiente manera:

$$f_{1\to 2} = \frac{I_2}{I_1}$$

Este factor de escala se emplea para una coordenada determinada.

Como se dijo anteriormente, tanto en *x* como en y, dicho factor puede ser distinto. No obstante, su cálculo es idéntico, teniendo en cuenta que I_1 e I_2 son el ancho de la imagen en la coordenada *x* y el alto en la coordenada y.

Veamos ahora cómo modificamos el código para conseguir que ambas imágenes tengan el mismo tamaño. Para ello, debemos definir nuevas variables en la sección de declaraciones:

```
float escalaX;
float escalaY;
float height_color, width_color;
float height_bw , width_bw;
```

A las dos primeras las usaremos para guardar el factor de escala en cada dimensión. Luego, contaremos con pares de variables para guardar el alto y ancho de cada una de las imágenes.

Una vez cargadas las imágenes en nuestro código, podemos calcular el factor de escala:

```
height_color=(float)image_color.GetHeight();
height_bw=(float) image_bw.GetHeight();
width_color=(float)image_color.GetWidth();
width_bw=(float) image_bw.GetWidth();

escalaY= height_bw/height_color;
escalaX= width_bw/width_color;
```

Para ello, primero debemos obtener los tamaños de la imagen en cada dimensión. En *y* lo hacemos con el método *GetHeight(),* mientras que en *x* recurrimos a *GetWidth()*. Ambos métodos devuelven un valor entero, por lo que para poder dividirlos y obtener un factor flotante, debemos convertirlos a *float*. Por ello se coloca el operador de *casting* en las cuatro primeras líneas.

Luego, aplicamos la fórmula dada anteriormente y obtenemos los factores de escala para que la imagen en color tenga el mismo tamaño que la que está en blanco y negro.

Posteriormente, escalamos el sprite en color con dicho factor:

```
sprite_color.SetScale(escalaX,escalaY);
```

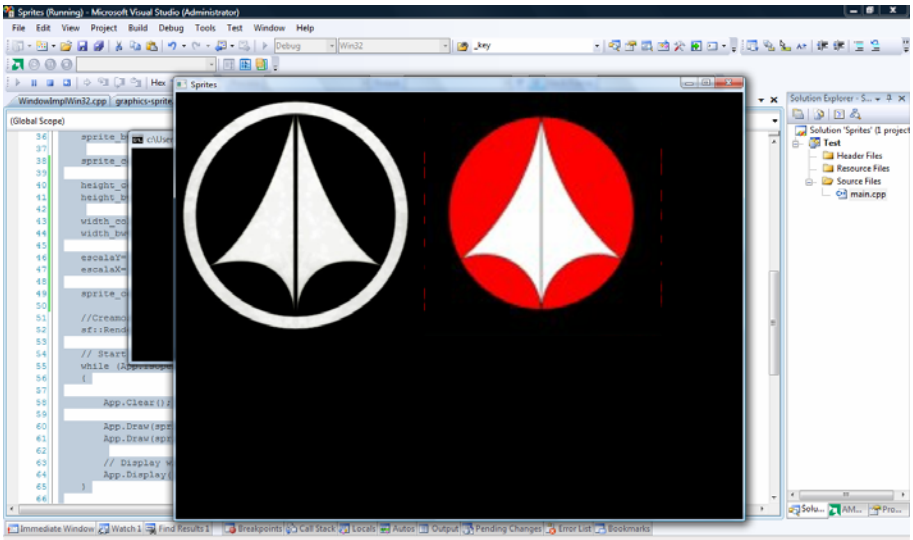
A continuación, se lista como quedaría nuestro código fuente:

Main.cpp

```
////////////////////////////////////////////////////////////
/// Variables
////////////////////////////////////////////////////////////

sf::Image image_color;
sf::Image image_bw;

sf::Sprite sprite_color;
sf::Sprite sprite_bw;

float escalaX;
float escalaY;
float height_color, width_color;
float height_bw , width_bw;
////////////////////////////////////////////////////////////
/// Punto de entrada a la aplicación
////////////////////////////////////////////////////////////
int main()
{
```

```
        //Cargamos la imagen del archivo
        image_color.LoadFromFile("..\\img\\un-spacy2.jpg");
        image_bw.LoadFromFile("..\\img\\un-spacy.jpg");

        //Configuramos el material del sprite
        sprite_color.SetImage(image_color);
        sprite_bw.SetImage(image_bw);

        sprite_color.SetPosition(350,0);

        height_color=(float)image_color.GetHeight();
        height_bw=(float) image_bw.GetHeight();

        width_color=(float)image_color.GetWidth();
        width_bw=(float) image_bw.GetWidth();

        escalaY= height_bw/height_color;
        escalaX= width_bw/width_color;

        sprite_color.SetScale(escalaX,escalaY);

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

 // Start main loop
 while (App.IsOpened())
 {

              App.Clear();

              App.Draw(sprite_bw);
              App.Draw(sprite_color);

              // Display window on screen
 App.Display();
 }

 return EXIT_SUCCESS;
}
```

Si compilamos dicho programa y lo ejecutamos, deberíamos ver lo siguiente:



Aquí, efectivamente, se observa el mismo tamaño en ambas imágenes.

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