UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación para videojuegos II

Unidad 2
Animaciones y modelado de personajes

Docente
Pablo Abratte

CONTENIDOS

1. Manejo eficiente de imágenes

En Modelos y Algoritmos para Videojuegos I aprendimos la diferencia entre una imagen y un sprite. Mientras que una *imagen* es un arreglo bidimensional de píxeles, un *sprite* es un objeto cuya representación gráfica está dada por una imagen, pero además posee otras propiedades como posición, rotación, escala, etc. Dicho de manera más sencilla, un sprite es una forma de desplegar una imagen en pantalla.

La biblioteca SFML, que utilizaremos durante este curso, también realiza esta distinción y nos provee de las clases *sf::Image* y *sf::Sprite* para representar imágenes y sprites, respectivamente.

Las imágenes son recursos que ocupan gran cantidad de memoria y cuya carga y copia son operaciones computacionalmente costosas, por lo que se deben tener ciertos cuidados a la hora de manipularlas.

Un error que se comete a menudo es utilizar una instancia de *sf::Image* para cada *sprite*, ya que es la forma más simple de dibujar algo en pantalla. Sin embargo, esto generalmente no es una buena idea, dado que se desperdicia memoria al tener múltiples copias de la misma imagen. El siguiente ejemplo ilustra esta situación:

```
1. #include <SFML/Graphics/Sprite.hpp>
2.
3. class Misil: public sf::Sprite{
4.     private:
5.     sf::Image imagenMisil;
6.
7.     public:
8.     Misil(){
9.         // cada misil guarda su propia copia
10.        // de la imagen
11.        imagenMisil.LoadFromFile("data/misil.png");
12.        SetImage(imagenMisil);
13.    }
14. };
```

La ventaja de diferenciar *sprites* e imágenes en SFML radica en que estas últimas son pesadas y computacionalmente costosas de manipular, mientras que los *sprites* son livianos, ya que sólo guardan información sobre la representación de una imagen, pero no la imagen en sí.

En la mayoría de los casos, lo correcto es compartir una única imagen entre muchos *sprites*.

> Como regla general debería existir una sola instancia de *sf::Image* por cada archivo de imagen que utilicemos.

Una forma posible de realizar esto, en el caso de una clase cuyas instancias utilizarán la misma imagen, es hacer lo siguiente:

```
1. #include <SFML/Graphics/Sprite.hpp>
2.
3. class Misil: public sf::Sprite{
4.     private:
5.     static sf::Image imagenMisil;
6.     static void Init(){
7.         imagenMisil.LoadFromFile("data/misil.png");
8.     }
9.
10.    public:
11.    Misil(){
12.        // todos los misiles comparten la misma imagen
13.        Sprite.SetImage(imagenMisil);
14.    }
15. };
```

En el ejemplo se crea una única imagen que es compartida por todos los objetos de la clase y que es necesario inicializar llamando a la función estática *Init()* antes de crear algún objeto.

En un diseño más avanzado, las imágenes pueden ser cargadas y accedidas automáticamente mediante un *manejador de imágenes* o *image manager,* permitiendo una utilización más sencilla y genérica de estos recursos. La idea es que el manejador guarde cada imagen junto con un nombre asociado para que, si alguna es solicitada más de una vez, la misma instancia sea siempre devuelta. El siguiente ejemplo ilustra su utilización:

Manejador de imágenes o image manager
Es una clase que gestiona imágenes de manera eficiente, simplificando el uso de dichos recursos.

```
#include <SFML/Graphics/Sprite.hpp>
#include "ImageManager.h"

class Misil: public sf::Sprite{
    public:
    Misil(){
        // se solicita la imagen al ImageManager, que devuelve
        // siempre la misma instancia para el nombre dado
        Sprite.SetImage(ImageManager.GetImage("data/misil.png"));
    }
};
```

En siguiente parágrafo analizaremos la implementación de un manejador de imágenes sencillo.

1.1. Implementación de un manejador de imágenes

A continuación, abordaremos la implementación de una clase sencilla que nos facilite la carga y utilización de imágenes. Esta clase, que denominaremos *ImageManager,* se encargará de almacenar cada imagen e identificarla con su correspondiente nombre de archivo. El usuario podrá interactuar con un objeto de esta clase, solicitándole una imagen que el manejador buscará en su memoria y devolverá, o cargará desde un archivo en caso de no tenerla aún.

El manejador guardará las imágenes en un objeto tipo *map<string, sf::Image>*. Su funcionalidad principal será el método *GetImage(),* que recibirá el nombre de un archivo y devolverá una referencia a la imagen buscada. Además, existirán otras funciones auxiliares que nos simplificarán la utilización de la clase.

En el siguiente código podemos observar el prototipo de la clase:

```
#ifndef IMAGEMANAGER_H
#define IMAGEMANAGER_H

#include <iostream>
#include <map>
#include <SFML/Graphics.hpp>
using namespace std;

class ImageManager {
private:
    map<string, sf::Image> images;

public:
    sf::Image &GetImage(string filename);
    sf::Image &operator[](string filename);
    unsigned Size();
};

#endif

```

La función *Size()* nos permitirá conocer cuantas imágenes guarda el *ImageManager* en su memoria. La sobrecarga del operador de subscripción (*[]*) nos permitirá obtener la

misma funcionalidad que *GetImage(),* pero de una forma más cómoda, como veremos más adelante.

```
1. #include "ImageManager.h"
2.
3. Image &ImageManager::GetImage(string filename){
4.     // buscamos la imagen pedida
5.     map<string, sf::Image>::iterator p;
6.     p=images.find(filename);
7.     // si no la encontramos
8.     if(p==images.end()){
9.         // la cargamos desde el archivo
10.        sf::Image img;
11.        img.LoadFromFile(filename);
12.        // y la insertamos en el mapa
13.        pair< map<string, sf::Image>::iterator, bool > q;
14.        q=images.insert(pair<string, sf::Image>(filename, img));
15.        // hacemos que p apunte a la imagen insertada
16.        p=q.first;
17.    }
18.    // devolvemos la imagen
19.    return p->second;
20. }
21.
22.
23. Image &ImageManager::operator[](string filename){
24.    return GetImage(filename);
25. }
26.
27.
28. unsigned ImageManager::Size(){
29.    return images.size();
30. }
31.
```

Analizaremos, ahora, la implementación de las funciones de la clase.

El funcionamiento de la función *Size()* y del operador de subscripción es sencillo y directo, mientras que el código de mayor complejidad se encuentra en la función *GetImage(),* que analizaremos a continuación.

En la línea 5 se declara un iterador que apuntará a la entrada del mapeo que contenga la imagen deseada. Utilizamos la función *find()* para buscar la imagen con el nombre especificado. En caso de encontrarla, no se entrará al condicional de la línea 8 y simplemente se devolverá el segundo campo del par apuntado por *p* (la imagen) en la línea 19.

En caso de no encontrar la imagen, esta será cargada desde el disco e insertada en el mapeo. Recordemos que la función *insert()* devuelve un par, constituido por un iterador, a otro par y un booleano. Este último valor no nos interesará, ya que sabemos de antemano que la imagen no se encuentra en el mapeo, por lo que simplemente igualamos *p* al primer campo de este par (el iterador a la entrada insertada) para luego retornar la imagen.

En el siguiente fragmento de código ejemplificamos la utilización del *ImageManager*:

```
1. #include <SFML/Graphics/Sprite.hpp>
2. #include "ImageManager.h"
3.
4. int main(int argc, char *argv[]){
5.     ImageManager im;
6.     Sprite misil1, misil2, misil3;
7.
8.     misil1.SetImage(im.GetImage("data/misil.png"));
9.     misil2.SetImage(im.GetImage("data/misil.png"));
10.    // esta forma permite lograr la misma funcion
11.    // pero de forma mas corta
12.    misil3.SetImage(im["data/misil.png"]);
```

```
13.
14.     // loop del juego...
15.
16.     return 0;
17. }
18.
```

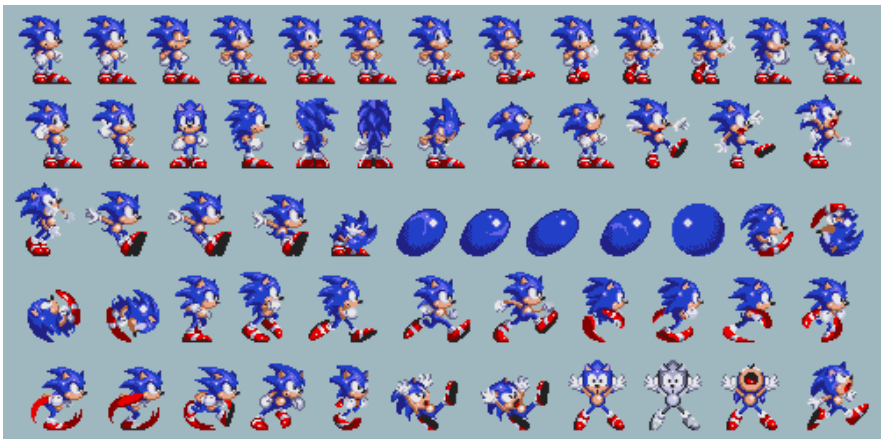
La clase puede utilizarse creando un único objeto para manejar todas las imágenes de un juego, o una instancia para cada clase que utilice imágenes diferentes. Otro buen diseño consistiría en modificar la clase para que los miembros y atributos sean estáticos y poder utilizarla sin necesidad de crear ningún objeto.

La herramienta propuesta es simplemente un ejemplo y de ninguna manera es la única ni mejor forma de manejar imágenes. Cada programador podrá utilizar las bibliotecas que considere más adecuadas, tanto propias como ajenas que se encuentren disponibles, en base a los requerimientos del videojuego que desarrolle.

Finalmente, las técnicas aquí aplicadas para imágenes también pueden ser aprovechadas con recursos de otros tipos como sonidos, *shaders*, música, etc.

1.2. Manejo de sprite sheets o grillas de imágenes

Muchas veces nos encontraremos con un único archivo que posee en su interior muchas imágenes, tal como se muestra en la siguiente figura:



Este tipo de imágenes se conoce como *sprite sheets* u hojas de sprites, las cuales contienen todas las texturas posibles para un sprite, es decir, los cuadros de todas las animaciones de un personaje o entidad. Un caso similar son los *tile sets,* que poseen la totalidad de las partes constitutivas de un escenario, generalmente denominado *tile map*.

Sprite sheets u hojas de sprites
Son imágenes que contienen todas las texturas posibles para un *sprite*, es decir, los cuadros de todas las animaciones de un personaje o entidad.

Es posible conseguir sprite sheets de muchos juegos clásicos de consolas en páginas como http://spritedatabase.net/. Algunas vienen acompañadas por archivos que especifican la posición de cada cuadro y la sucesión de los mismos en cada animación. Existen motores de videojuegos que están incluso optimizados para aprovechar este tipo de representación.

Podríamos adaptar nuestro *ImageManager* para trabajar también con sprite sheets. Sin embargo, éste identifica las imágenes por su nombre de archivo, lo cual complica su utilización al ser necesario especificar algún tipo de regla para nombrar a cada una de las imágenes presentes en un archivo.

Por lo tanto, en lugar de utilizar nuestro *ImageManager,* implementaremos una nueva clase que maneje específicamente sprite sheets, permitiendo acceder a las imágenes mediante un índice y no a través de un nombre.

Este nuevo manejador, que denominaremos *SpriteSheetManager,* será muy similar al anterior en cuanto a sencillez y funcionalidad. A continuación, veremos su implementación.

```
#ifndef SPRITESHEETMANAGER_H
#define SPRITESHEETMANAGER_H
#include <vector>
#include <SFML/Graphics.hpp>
using namespace std;

class SpriteSheetManager{
private:
    vector<sf::Image> images;

public:
    // carga un sprite sheet
    void Load(string filename, unsigned nCols, unsigned nRows=1);
    // devuelve la imagen en la posicion i
    sf::Image &GetImage(unsigned i);
    sf::Image &operator[](unsigned i);
    // devuelve la cantidad de imagenes almacenadas
    unsigned Size();
};

#endif

```

```
#include "SpriteSheetManager.h"

void SpriteSheetManager::Load( string filename,
                               unsigned nCols,
                              unsigned nRows){
    sf::Image sheet;
    // cargamos
    sheet.LoadFromFile(filename);
    // segun la cantidad de filas/columnas calculamos el tamanio
    // de las subimagenes
    unsigned subimgw, subimgh;
    subimgw=sheet.GetWidth()/nCols;
    subimgh=sheet.GetHeight()/nRows;

    unsigned i, j;
    // r contendra la region de la imagen que nos interesa
    sf::IntRect r;
    // en temp copiaremos la parte de la imagen que buscamos
    sf::Image temp(subimgw, subimgh);

    // descomentar la linea siguiente si no quieren interpolacion
    //temp.SetSmooth(false);

    for(i=0; i<nRows; i++){
        for(j=0; j<nCols; j++){
            // cargamos r con los datos del rectangulo
            r.Left=subimgw*j;
            r.Top=subimgh*i;
            r.Right=r.Left+subimgw;
            r.Bottom=r.Top+subimgh;
            // copiamos una porcion de la imagen a temp
            temp.Copy(sheet, 0, 0, r);
            // e insertamos una copia de temp en el vector
            images.push_back(temp);
        }
    }
}

sf::Image &SpriteSheetManager::GetImage(unsigned i){
    return images[i];
}

sf::Image &SpriteSheetManager::operator[](unsigned i){
    return images[i];
}

unsigned SpriteSheetManager::Size(){
```

```
48.     return images.size();
49. }
50.
```

En esta ocasión, utilizaremos un vector para almacenar las imágenes. Será necesario llamar a la función *Load()* para cargarlas antes de utilizarlas. Dicha función realiza la mayor parte del trabajo, dividiendo una imagen en subimágenes más pequeñas, las cuales tendrán el mismo tamaño, que será calculado en base a la cantidad de filas y columnas en la grilla, pasadas como parámetro a la función.

En el siguiente parágrafo haremos uso de esta clase para crear animaciones, cambiando la imagen de un *sprite* con el transcurso del tiempo.

2. Animaciones

Animación
Cambio de textura o imagen de un sprite a medida que transcurre el tiempo.

Definiremos animación como el cambio de textura o imagen de un sprite a medida que transcurre el tiempo.

Figura 1. Sucesión de imágenes que conforman una de las animaciones de un sprite.

A continuación, mostraremos un código que anima esta grilla de imágenes:

```
 1. #include <SFML/Window.hpp>
 2. #include <SFML/Graphics.hpp>
 3. #include "SpriteSheetManager.h"
 4. using namespace sf;
 5.
 6. // cuanto se muestra cada frame
 7. const float animVel=0.1;
 8.
 9. int main(int argc, char *argv[]){
10.     // creamos nuestra ventana y definimos el area visible
11.     RenderWindow w(VideoMode(300,300),"Ejemplo Animacion 1");
12.     w.SetView(View(FloatRect(0,0,150,150)));
13.
14.     // creamos nuestro manager de imagenes
15.     SpriteSheetManager sm;
16.     sm.Load("hulk.png", 8);
17.
18.     // creamos el sprite
19.     Sprite hulk;
20.     hulk.SetPosition(0,0);
21.
22.     // el nro de cuadro que estamos mostrando
23.     int iFrame;
24.     // y la cantidad de tiempo que ya lo mostramos
25.     float frameElapsed=0;
26.
27.     // el reloj para contar el paso del tiempo
28.     Clock clk;
29.
30.     // el bucle del juego...
31.     while(w.IsOpened()) {
32.         sf::Event e;
33.         while(w.GetEvent(e)) {
34.             if(e.Type == e.Closed)
35.                 w.Close();
36.         }
37.
```

```
38.         // limpiamos el buffer de pantalla
39.         w.Clear(Color(0,0,0,255));
40.
41.         // calculamos el tiempo que paso
42.         frameElapsed+=clk.GetElapsedTime();
43.         clk.Reset();
44.
45.         if(frameElapsed>animVel){
46.             // actualizamos el frame y el tiempo
47.             iFrame++;
48.             frameElapsed-=animVel;
49.             // cuidamos que el nro de frame no se pase
50.             if(iFrame>7) iFrame=0;
51.         }
52.
53.         // seteamos la imagen del sprite
54.         hulk.SetImage(sm[iFrame]);
55.
56.         // y lo dibujamos
57.         w.Draw(hulk);
58.
59.         // actualizamos la pantalla
60.         w.Display();
61.     }
62.     return 0;
63. }
64.
```

En las líneas 11 y 12 se crea la ventana y se asigna una vista para visualizar solamente la porción del plano donde se dibuja el *sprite*. En las líneas 15 y 16, creamos una instancia del *SpriteSheetManager*, que desarrollamos anteriormente, y cargamos la grilla de imágenes (8 columnas y 1 fila).

Para realizar la animación, utilizaremos dos variables: *iFrame* indica el número de *frame* actual en la animación, mientras que *frameElapsed* cuenta el tiempo que dicho *frame* lleva siendo mostrado.

En la constante *animVel*, declarada en la línea 7, especificamos por cuántos segundos será mostrado cada cuadro. En la línea 42 se acumula, en la variable *frameElapsed*, el tiempo transcurrido desde la ultima actualización. Luego, en la línea 45, se evalúa si el *frame* actual ha sido mostrado por suficiente tiempo y, de ser así, se pasa al siguiente cuadro de la animación y se actualiza la variable *frameElapsed*.

Por último, asignamos la imagen al *sprite* y lo mostramos en pantalla.

Un personaje puede tener muchas animaciones y sería complicado realizar el procedimiento ejemplificado para manejar las animaciones de todas las entidades intervinientes en un juego. Por esta razón, desarrollaremos una clase que se encargue de realizar este trabajo por nosotros y nos permita utilizar animaciones de manera sencilla y genérica.

Antes de comenzar debemos preguntarnos: ¿qué características necesitamos que posea nuestra clase?

Algunas funcionalidades que nos serán de utilidad al trabajar con animaciones son las siguientes:

- Utilizar *frames* que tengan distinta duración.
- Elegir si al terminar la animación debe realizar un *loop* o quedar en el último cuadro.
- En caso de no realizar un *loop*, saber si la animación terminó.
- Poder reiniciar la animación.
- Permitir que nuestra clase maneje animaciones de un único cuadro para poseer flexibilidad.

A continuación, mostramos el prototipo de la clase propuesta en base a estas necesidades:

```
#ifndef ANIMATION_H
#define ANIMATION_H
#include <vector>
#include <SFML/Graphics/Sprite.hpp>
using namespace std;
using namespace sf;


class Animation{
private:
    // punteros a las imagenes
    vector<Image *> frames;
    // duracion de cada frame
    vector<float> frameTimes;
    // tiempo que ha sido mostrado el frame actual
    float currentFrameElapsed;
    // nro de frame actual y nro total de frames
    int iframe, nFrames;
    // banderas para saber si la animacion debe
    // ciclar o de lo contrario si ya finalizo
    bool loop, end;

public:
    Animation(bool loop=true);
    // permite agregar un nuevo frame con su duracion
    void AddFrame(Image &, float lenght);

    // permite avanzar la animacion un tiempo dt
    // y devolver la imagen actual
    Image &Animate(float dt);
    Image &GetCurrentFrame();

    // algunas funciones para controlar la animacion
    void SetLoop(bool loop);
    bool GetLoop();
    void Reset();
    bool Ended();

    // otras funciones para tener mayor control
    float GetCurrentFrameElapsedTime();
    void SetCurrentFrameNum(int i);
    int GetCurrentFrameNum();
};

#endif

```

La clase se encarga de gestionar una animación, guardando dos vectores con punteros a imágenes y el tiempo en segundos que cada una debe ser mostrada. Posee, al igual que en el ejemplo anterior, variables que contienen el número de *frame* actual y el tiempo que éste ha sido desplegado. La variable *nFrames* cuenta la totalidad de cuadros en la animación; la variable *loop* sirve para controlar si la animación es cíclica y, en caso de no serlo, la bandera *end* indicará si la misma llegó a su fin.

La función *Animate()* realiza la funcionalidad expuesta en el ejemplo anterior. Recibe el tiempo transcurrido, actualiza el estado de la animación y devuelve un puntero a la imagen que debe mostrarse.

Ahora veremos los detalles de la implementación de esta clase:

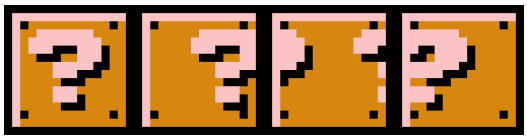
```
#include "Animation.h"

// construye la animacion inicializando todas las variables
Animation::Animation(bool loop){
    this->loop=loop;
    Reset();
}
```

```
 8.
 9. // agrega un nuevo cuadro con su respectiva duracion
10. void Animation::AddFrame(Image &i, float frameLenght){
11.     frames.push_back(&i);
12.     frameTimes.push_back(frameLenght);
13.     nFrames++;
14. }
15.
16. // avanza la animacion en base al tiempo transcurrido
17. // tiene en cuenta si debe hacer un loop o finalizar
18. // ademas devuelve un puntero a la imagen actual de
19. // la animacion para un uso mas comodo
20. Image &Animation::Animate(float dt){
21.     currentFrameElapsed+=dt;
22.     if(currentFrameElapsed>frameTimes[iframe]){
23.         currentFrameElapsed-=frameTimes[iframe];
24.         iframe++;
25.         if(iframe>=frames.size()){
26.             if(loop)
27.                 // volvemos al principio
28.                 iframe=0;
29.             else{
30.                 // la animacion finalizo
31.                 iframe--;
32.                 end=true;
33.             }
34.         }
35.     }
36.     return *frames[iframe];
37. }
38.
39. // devuelve un puntero a la imagen actual de la animacion
40. Image &Animation::GetCurrentFrame(){
41.     return *frames[iframe];
42. }
43.
44. // estas 2 funciones permiten controlar si se debe
45. // o no hacer loop
46. void Animation::SetLoop(bool loop){
47.     this->loop=loop;
48. }
49.
50. bool Animation::GetLoop(){
51.     return loop;
52. }
53.
54. // permite saber si la animacion termino
55. // (en caso de que no haga loop)
56. bool Animation::Ended(){
57.     return end;
58. }
59.
60. // reinicializa la animacion
61. void Animation::Reset(){
62.     iframe=0; currentFrameElapsed=0; end=false;
63. }
64.
65.
66. // estas funciones permiten un mayor control
67. // nos van a ser utiles mas adelante
68. void Animation::SetCurrentFrameNum(int i){
69.     iframe=i;
70. }
71.
72. int Animation::GetCurrentFrameNum(){
73.     return iframe;
74. }
75.
76. float Animation::GetCurrentFrameElapsedTime(){
77.     return currentFrameElapsed;
78. }
79.
```

Nuestra clase no guarda las imágenes de la animación, sino punteros a las mismas, por lo que puede ser utilizada en conjunto con alguno de los manejadores de imágenes desarrollados anteriormente.

Mostraremos un ejemplo de la utilización de la clase desarrollada, realizando diversas animaciones con la siguiente grilla de imágenes:



```cpp
#include <SFML/Window.hpp>
#include <SFML/Graphics.hpp>
#include "SpriteSheetManager.h"
#include "Animation.h"
using namespace sf;

// cuanto se muestra cada frame
const float animVel=0.1;

int main(int argc, char *argv[]){
    // creamos nuestra ventana y definimos el area visible
    RenderWindow w(VideoMode(300,300),"Ejemplo Animacion 2");
    w.SetView(View(FloatRect(0,0,60,60)));

    // creamos nuestro manager de imagenes
    SpriteSheetManager sm;
    sm.Load("box_sprites.png", 4);

    // creamos los sprites
    Sprite box1, box2, box3;
    box1.SetPosition(0,0);
    box2.SetPosition(20,0);
    box3.SetPosition(40,0);

    // creamos las animaciones
    Animation anim1, anim2, anim3;
    anim1.AddFrame(sm[0], 0.25);
    anim1.AddFrame(sm[1], 0.25);
    anim1.AddFrame(sm[2], 0.25);
    anim1.AddFrame(sm[3], 0.25);

    anim2.AddFrame(sm[0], 1);
    anim2.AddFrame(sm[1], 0.25);
    anim2.AddFrame(sm[2], 0.25);
    anim2.AddFrame(sm[3], 0.25);
    anim2.AddFrame(sm[0], 1);
    anim2.AddFrame(sm[3], 0.25);
    anim2.AddFrame(sm[2], 0.25);
    anim2.AddFrame(sm[1], 0.25);

    // tiene 1 solo frame por lo que sera estatica
    anim3.AddFrame(sm[0], 1);

    // el reloj para contar el paso del tiempo
    Clock clk;
    float dt;

    // el bucle del juego...
    while(w.IsOpened()) {
        sf::Event e;
        while(w.GetEvent(e)) {
            if(e.Type == e.Closed)
                w.Close();
        }

        // calculamos el tiempo transcurrido
        dt=clk.GetElapsedTime();
        clk.Reset();
```

```
59.
60.         // limpiamos el buffer de pantalla
61.         w.Clear(Color(0,0,0,255));
62.
63.         box1.SetImage(anim1.Animate(dt));
64.         box2.SetImage(anim2.Animate(dt));
65.         box3.SetImage(anim3.Animate(dt));
66.
67.         // dibujamos los sprites
68.         w.Draw(box1);
69.         w.Draw(box2);
70.         w.Draw(box3);
71.
72.         // actualizamos la pantalla
73.         w.Display();
74.     }
75.     return 0;
76. }
77.
```

3. Un paso más allá

Como comentamos anteriormente, las clases desarrolladas para manipular imágenes y animaciones constituyen un ejemplo sencillo y distan aún de lo que podría considerarse una herramienta óptima para el desarrollo de videojuegos.

Por eso, en este punto de la unidad nos plantearemos cuáles son las mejoras posibles para las clases desarrolladas. Expondremos algunas ideas y desarrollos potenciales respecto de las mismas.

La forma en que cargamos las *sprite sheets* es sencilla, dado que asumimos que todos los frames tienen igual tamaño. Sin embargo, de esta manera los cuadros pequeños tendrán el mismo tamaño que los grandes, desperdiciando no sólo memoria, sino también ciclos de GPU para copiar en pantalla una gran cantidad de píxeles transparentes que no son necesarios.

Sería, entonces, de gran utilidad desarrollar un programa que permita cargar una *sprite sheet* y seleccionar la región que ocupa cada cuadro junto con su centro. Este software podría permitir también la edición y previsualización de animaciones.

Otra mejora posible es evitar la tarea de dividir una imagen en partes más pequeñas y utilizar la función *SetSubRect()* de la clase *sf::Sprite,* obteniendo un pequeño aumento en el rendimiento.

Hasta ahora sólo hemos considerado una animación como el cambio de textura de un sprite con el transcurso de tiempo. Una definición más general de animación debería incluir el cambio de otras propiedades como escala, modo de blending y transparencia... Sería apropiado adecuar nuestra clase *Animation* para contemplar, también, estas características.

4. Modelado de un personaje

Las clases desarrolladas en los parágrafos anteriores nos simplifican la tarea de cargar imágenes y crear animaciones. Estas animaciones nos permiten representar el movimiento o evolución de personajes u otras entidades de un videojuego. Dichas entidades pueden llegar a ser muy complejas, teniendo un número muy grande de acciones y animaciones posibles.

A continuación, presentaremos el concepto de máquina de estado finito, que nos permitirá sortear la complejidad del modelado de estas entidades. Mostraremos, luego, su aplicación mediante un ejemplo.

4.1. Máquinas de estado finito

Máquina de estado finito
También conocida como autómata de estado finito, es una representación abstracta de un sistema con un conjunto de estados, cuya salida depende no sólo de la entrada, sino también de su estado actual.

Una *máquina de estado finito (FSM, del inglés Finite State Machine),* también conocida como *autómata de estado finito,* es una representación abstracta de un sistema con un conjunto de estados, cuya salida depende no sólo de la entrada, sino también de su estado actual.

Este tipo de modelo es de muchísima utilidad en un sinnúmero de áreas, entre ellas, la electrónica y la inteligencia artificial. En nuestro caso particular, nos servirá para modelar el comportamiento de programas y entidades en videojuegos.

Formalmente, una máquina de estado finito M está conformada por:

- Un conjunto finito I de símbolos de entrada.
- Un conjunto finito O de símbolos de salida.
- Un conjunto finito S de estados.
- Un estado inicial S_0.
- Una función $f(S,I) \rightarrow S$, que determina el siguiente estado en base al estado y entrada actuales.
- Una función $g(S,I) \rightarrow O$, que determina la siguiente salida en base al estado y entrada actuales.

Esto se escribe $M = \{I, O, S, S_0, f, g\}$.

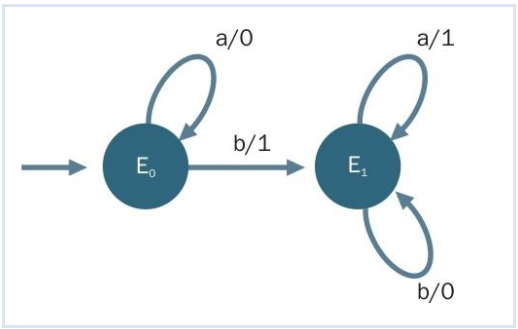
Veamos un ejemplo:

Dados los conjuntos $A = \{a, b\}$ de símbolos de entrada, $B = \{0, 1\}$ de símbolos de salida, $E = \{E_0, E_1\}$ de estados, el estado inicial E_0 y las funciones $h(E, A)$ e $i(E, A)$ que determinan respectivamente las transiciones de estado y salidas, y se definen de la siguiente forma:

$$\begin{array}{ll} f(E_0, 0) = E_0 & g(E_0, 0) = 0 \\ f(E_0, 1) = E_1 & g(E_0, 1) = 1 \\ f(E_1, 0) = E_1 & g(E_1, 0) = 1 \\ f(E_1, 1) = E_1 & g(E_1, 1) = 0 \end{array}$$

Entonces decimos que $N = \{A, B, E, E_0, f, g\}$ es una máquina de estado finito.

N puede ser representada de diversas formas; una de ellas es mediante un grafo, como se ilustra a continuación:



Los nodos del grafo representan los estados de N, mientras que los arcos indican las transiciones entre estados. Cada arco está etiquetado con la entrada que produjo la transición y la salida del sistema. La flecha que llega desde la izquierda señala el estado inicial, que a veces es representado alternativamente mediante un doble círculo.

El funcionamiento del sistema es el siguiente: comienza en el estado E_0,; a partir de allí puede recibir como entradas a o b. Si la entrada es a, lanzará una salida de 0 y seguirá en el mismo estado, mientras que si es b, el sistema dará una salida de 1 y cambiará al

estado E_1. En este nuevo estado, la salida será 1 o 0, según la entrada sea a o b, pero ya no ocurrirán cambios de estado en ningún caso.

Otra forma conveniente de representar una *FSM* es describiendo las funciones f y g a través de tablas. En este caso, la tabla que especifica a f se conoce como *tabla de transición de estados*.

Tabla de transición de estados
Es una tabla que especifica el estado al cual deberá moverse una FSM en función de su estado y entrada actuales.

A continuación, mostramos dichas tablas para la máquina ejemplificada anteriormente:

$h(E, A)$		
Entrada/Estado	E_0	E_1
a	E_0	E_1
b	E_1	E_1

$i(E, A)$		
Entrada/Estado	E_0	E_1
a	0	1
b	1	0

En el siguiente parágrafo veremos cómo puede utilizarse este modelo para simplificar la representación de un personaje en un videojuego.

4.2 Modelado de un personaje como una FSM

Podemos aprovechar el concepto de máquina de estado finito para reducir la complejidad del modelado de un personaje en un videojuego.

Si consideramos al personaje o entidad como una *FSM,* debemos definir un conjunto de entradas, salidas y estados para luego especificar los mapeos entre dichos conjuntos.

Consideraremos como entradas del sistema a todos los eventos que afecten al estado del personaje, como interacciones con el entorno (escenario, otros personajes, ítems, etc.), controles del usuario o eventos temporales (terminó de disparar, expiró un ítem de invencibilidad, etc.).

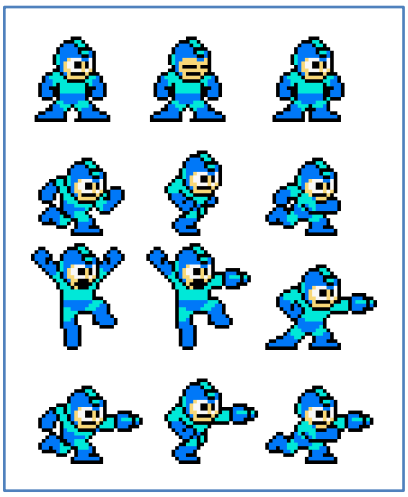
Por el momento, no nos preocuparemos por las salidas, que serán las acciones de nuestro personaje en pantalla.

Por otro lado, enfocaremos nuestro interés en los estados y sus transiciones, ya que en función de los mismos actualizaremos las características de nuestro personaje.

A modo de ejemplo, modelaremos el comportamiento del legendario *Megaman.*

Definiremos los siguientes estados para nuestro personaje:

- PARADO: cuando no hace nada.
- SALTANDO: moviéndose sólo verticalmente, es decir, saltando en el lugar.
- CORRIENDO: moviéndose hacia alguno de los costados.
- DISPARANDO: disparando mientras está parado, es decir, en el lugar.
- CORRIENDO Y DISPARANDO: disparando mientras se mueve a alguno de los costados.
- SALTANDO Y DISPARANDO: disparando mientras está saltando en el lugar.
- SALTANDO Y MOVIÉNDOSE: moviéndose hacia alguno de los costados en medio de un salto.
- SALTANDO MOVIÉNDOSE Y DISPARANDO: moviéndose hacia alguno de los costados mientras está en el aire y disparando.



Las entradas que podrán afectar al estado de Megaman son las siguientes:

- El usuario presiona la flecha hacia adelante.
- El usuario presiona la flecha hacia atrás.
- El usuario presiona el botón *A* (saltar).
- El usuario presiona el botón *B* (disparar).
- El usuario suelta la flecha hacia adelante.
- El usuario suelta la flecha hacia atrás.
- La secuencia de disparo finalizó.
- Megaman colisionó con el suelo.

Antes de continuar, debemos notar que el modelado del personaje será parcial, ya que se han omitido muchos comportamientos que existen en el juego original, como cuando es golpeado por un enemigo o utiliza poderes especiales. Cuanto más complejo sea un personaje, más serán los estados y las entradas que necesitaremos para modelarlo, y por consiguiente, más complejas se tornarán las funciones de mapeo entre entradas y estados.

Dada la gran cantidad de estados y entradas, es impráctico expresar el funcionamiento de la FSM mediante un grafo porque resulta tedioso interpretarlo. Utilizaremos, en cambio, la tabla de transición de estados que exhibimos a continuación:

Estados - Eventos	PARADO	CORRIENDO	SALTANDO	DISPARANDO	CORRIENDO Y DISPARANDO	SALTANDO Y DISPARANDO	SALTANDO Y MOVIÉNDOSE	SALTANDO MOVIÉNDOSE Y DISPARANDO
Presionar Adelante	CORRIENDO		SALTANDO Y MOVIÉNDOSE	CORRIENDO Y DISPARANDO		SALTANDO MOVIÉNDOSE Y DISPARANDO		
Presionar Atrás	CORRIENDO		SALTANDO Y MOVIÉNDOSE	CORRIENDO Y DISPARANDO		SALTANDO MOVIÉNDOSE Y DISPARANDO		
Botón *A*	SALTANDO	SALTANDO Y MOVIÉNDOSE		SALTANDO Y DISPARANDO	SALTANDO MOVIÉNDOSE Y DISPARANDO			
Botón *B*	DISPARANDO	CORRIENDO Y DISPARANDO	SALTANDO Y DISPARANDO				SALTANDO MOVIÉNDOSE Y DISPARANDO	
Soltar Adelante		PARADO			PARADO		SALTANDO	SALTANDO Y DISPARANDO
Soltar Atrás		PARADO			PARADO		SALTANDO	SALTANDO Y DISPARANDO
Secuencia de disparo finalizada				PARADO	CORRIENDO	SALTANDO		SALTANDO Y MOVIÉNDOSE
Colisión con el suelo			PARADO			PARADO	PARADO	PARADO

La tabla nos indica cuál será el nuevo estado según el evento recibido (si es que el evento significa algo para ese estado). Por ejemplo, si el estado actual es PARADO, se pasará al estado CORRIENDO si es presionada alguna de las flechas de dirección hacia adelante o atrás; se pasará al estado SALTANDO si se presiona el botón *A,* o al estado DISPARANDO si el botón presionado es el *B*. Los demás eventos no tienen sentido en este estado. El estado inicial es, por supuesto, estando PARADO.

Una vez que se conoce el estado actual de nuestro personaje, es fácil determinar lo que debemos hacer. Generalmente tendremos una animación por cada estado del personaje. Según el estado, también realizaremos determinadas acciones. Así, si el estado es CORRIENDO o CORRIENDO Y DISPARANDO, actualizaremos la posición horizontal; en cambio, si el estado es SALTANDO Y MOVIÉNDOSE, deberemos actualizar tanto la posición horizontal, como su posición y velocidad vertical.

Una vez que el modelo fue correctamente planteado mediante una tabla de transición de estados, su codificación es directa y se simplifica la depuración y corrección de errores en el comportamiento del modelo. Primero, necesitaremos desarrollar funciones o variables que indiquen la ocurrencia de los eventos que serán la entrada de nuestra máquina. Luego, en base a dicha entrada y al estado actual, determinaremos el estado siguiente y realizaremos las acciones pertinentes.

Veamos algunos fragmentos de código de la implementación de Megaman:

```
#include "Megaman.h"
#include "Animation.h"
#include "SpriteSheetManager.h"

class Megaman: public Sprite{
    private:
    enum Estado{
        PARADO,
        CORRIENDO,
        DISPARANDO,
        SALTANDO,
        DISPARANDO_Y_CORRIENDO,
        SALTANDO_Y_DISPARANDO,
        SALTANDO_Y_MOVIENDOSE,
        SALTANDO_MOVIENDOSE_Y_DISPARANDO
    };
    Estado estado;
    int direccion;
    Animation animaciones[8];

    void CambiarEstado(Estado nuevoEstado);

    // funciones que determinan eventos de entrada
    bool SecuenciaDisparoFinalizada();
    bool ColisionaConSuelo();

    // algunas acciones
    void Saltar();
    void Disparar(float x, float y);

    public:
    Megaman();
    void Mover_y_Animar(Joystick j, float dt);
};
```

Como se puede observar, hemos declarado los estados del personaje mediante una enumeración (cuyos nombres simbolizan valores enteros) y utilizado la variable *estado* para indicar el actual. Como mencionamos anteriormente, a cada estado corresponde una animación, por lo que tenemos que utilizar un arreglo de animaciones al que accedemos en la posición correspondiente al estado actual.

En el siguiente fragmento de código utilizaremos una estructura *Joystick* como eventos de entrada, la cual se pasará como parámetro al método *Mover_y_Animar()* e indicará las teclas o botones que el usuario presionó. En tanto, para la interacción de Megaman con el entorno, utilizaremos las funciones *SecuenciaDisparoFinalizada()* y *ColisionaConSuelo()* a fin de saber si estas condiciones se cumplen.

Finalmente, utilizaremos algunas funciones auxiliares como *Saltar()* y *Disparar(),* las cuales iniciarán secuencias de acciones al cumplirse ciertas transiciones entre estados. Podríamos interpretarlas como algunas de las salidas de nuestra máquina de estado finito.

El código que implementa la transición de estados se encontrará, como es de esperarse, en la función *Mover_y_Animar().* El siguiente código ilustra algunos fragmentos de dicha función. A pesar de tratarse de un código largo y complejo, el modelado realizado a priori permitirá mantenerlo ordenado según los distintos estados de nuestro personaje, facilitando la identificación de errores.

Una vez aplicada la lógica de transición de estados, podrán ajustarse las propiedades de nuestro personaje (posición, textura, etc.) en función del estado actual.

```
void Megaman::Mover_y_Animar(Joystick j, float dt){
    switch(estado){
        case PARADO:
            if(j.left){
```

```
                CambiarEstado(CORRIENDO);
                direccion=-1;
            }else if(j.right){
                CambiarEstado(CORRIENDO);
                direccion=1;
            }else if(j.a){
                CambiarEstado(DISPARANDO);
                Disparar(GetPosition().x+8*direccion, GetPosition().y);
            }else if(j.b){
                CambiarEstado(SALTANDO);
                Saltar();
            }
        break;

        case CORRIENDO:
            if(!j.left && !j.right){
                CambiarEstado(PARADO);
            }else if(j.b){
                CambiarEstado(SALTANDO_Y_MOVIENDOSE);
                Saltar();
            }else if(j.a){
                CambiarEstado(DISPARANDO_Y_CORRIENDO, true);
                Disparar(GetPosition().x+8*direccion, GetPosition().y);
            }
        break;

        ...

    }

    if(estado==CORRIENDO || estado==SALTANDO_Y_MOVIENDOSE ||
       estado==DISPARANDO_Y_CORRIENDO ||
       estado==SALTANDO_MOVIENDOSE_Y_DISPARANDO){
        Move(direccion*dx*dt, 0);
    }

    if(estado==SALTANDO || estado==SALTANDO_Y_MOVIENDOSE ||
       estado==SALTANDO_Y_DISPARANDO ||
       estado==SALTANDO_MOVIENDOSE_Y_DISPARANDO){
        vy+=gravity*dt;
        Move(0, vy*dt);
    }

    SetImage(animaciones[estado].Animate(dt));
}
```

Les sugerimos analizar minuciosamente el código completo para comprender en profundidad los detalles de la implementación, ya que se trata de un ejemplo fundamental.

Bibliografía

Simple and Fast Multimedia Library [en línea] http://www.sfml-dev.org/tutorials/

Wikipedia [en línea] http://en.wikipedia.org

JOHNSONBAUGH, R. *Matemáticas Discretas*. Pearson Education, 2005, 6° ed.